The 1.9 Millisecond Puzzle

Investigating the disproportionate cost of creating a near-empty object.

1.9ms

The Problem Constructing an `MGraph__Index()` object, containing only empty data structures, takes approximately 1.9 milliseconds.

The Discrepancy A 73:1 ratio of construction overhead versus actual work performed.

The Impact With 6 indexes, this results in a cumulative 11.4ms of unnecessary overhead.

The Hypothesis Recursive auto-initialisation of nested `Type_Safe` objects is creating a cascade of over 100 unwanted instances.

The Goal: Reduce construction time from 1.9ms to under 200µs – a 10x improvement.

NotebookLM

The High-Fidelity Development Environment

This investigation was conducted in an environment that allowed for rapid, hypothesis-driven experimentation directly against the production codebase.

Live Code Execution

Created and ran over 8 distinct Python files to test, benchmark, and debug.

Full File System Access

Read source code, created test files, and organised outputs seamlessly.

Package Installation

Installed the production `osbot-utils` package directly via `pip`.

Real Source Code Access

Worked with a 340KB digest of 177 relevant `Type_Safe` source files.

Nanosecond-Precision Benchmarking

Used the `Performance_Measure__Session` tool for rigorous validation.

Large Context Handling

Managed ~68,000 tokens of source code via automated context compression.

NotebookLM

A Blueprint for Success: The Anatomy of an Effective Brief

The session's velocity was enabled by a comprehensive brief that eliminated ambiguity and allowed for immediate problem replication.

1 **Clear Problem Statement**
`MGraph__Index()` construction is slow (~1.9ms) despite being logically empty.

2 **Quantified Impact**
A 73:1 overhead-to-work ratio, leading to significant cumulative delays.

3 **Root Cause Hypothesis**
Uncontrolled recursive initialisation of nested `Type_Safe` objects.

4 **Object Tree Visualization**
A complete dependency graph showing all 100+ nested objects and duplicates.

5 **Unambiguous Success Criteria**
A measurable target of <200µs construction time.

6 **Suggested Starting Points**
Six potential solution strategies to investigate.

A well-structured brief transforms the task from exploration into a targeted, measurable engineering problem.

NotebookLM

Phase 1: Establishing Ground Truth

The investigation followed a hypothesis-driven methodology, creating isolated test files to validate each assumption and measure every change.

Experiment 1: Baseline Measurement

Hypothesis

The performance figures and object count described in the brief are accurate and reproducible.

Action

A new script, `benchmark_type_safe.py`, was created to simulate the `MGraph__Index` object hierarchy.

Result

Confirmed.

Data

Construction Time: **~1.8ms**
Objects Created: **47**

This baseline provides the ground truth for all subsequent experiments.

NotebookLM

First Dead End: An Attempt to Outsmart the Framework

Experiment 2: Solution Attempt v1 (`Type_Safe_Lazy`)

Hypothesis

We can intercept object creation *before* `__init__` by overriding the `__cls_kwargs__` calculation method.

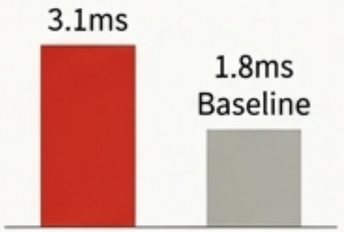
Action

Implemented `lazy_type_safe_solution.py` with a new `Type_Safe_Lazy` class.

Result

FAILED. The solution was significantly slower than the baseline.

New Construction Time: **3.1ms** (vs. 1.8ms baseline)



Analysis

The override added its own overhead without successfully preventing the parent class's object creation logic. The framework's internal caching mechanisms were not being correctly influenced.

Lesson Learned: Intercepting framework behaviour at unconventional points is fragile. The most robust solutions leverage existing, well-defined mechanisms.

NotebookLM

A Deeper Mystery: The Objects That Refused to Disappear

Experiment 3 & 4: Solution Attempts v2/v3

Hypothesis

Set all `Type_Safe`-typed attributes to `None` during `__init__`, then create the real objects on first access using `__getattribute__`.

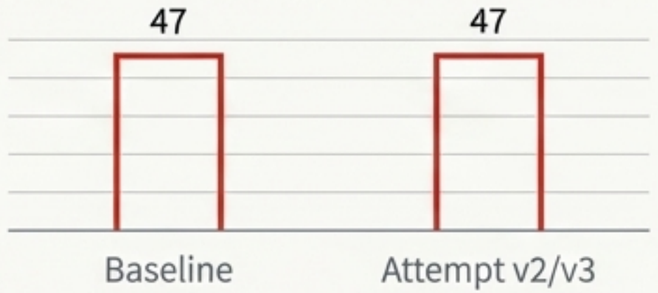
Action

Implemented `type_safe_lazy_v2.py` and `v3` with this approach.

Result

FAILED. The object count remained unchanged, and performance did not improve.

Objects Created: 47 (identical to baseline)
Source Sans 3



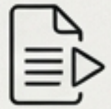
> The debugging logs confirmed that the keyword arguments for nested objects were being correctly set to `None`. The framework is designed to skip creation for `None` values. So why were the objects still being created?

NotebookLM

The Breakthrough: A Single Line of Code Reveals the Culprit

Experiment 5: Deep Debugging

Hypothesis: An unknown process during the parent class's initialisation is triggering our `__getattribute__` override prematurely.



Action: Created `debug_lazy.py` with extensive logging inside the `__init__` and `__getattribute__` methods to trace the exact sequence of events.

ROOT CAUSE FOUND.

```
super().__init__(**kwargs)
```

The parent `Type_Safe` class's `__init__` method performs internal setup and validation. During this process, it accesses its own attributes. Attributes. These internal `self.some_attribute` calls were triggering our `__getattribute__` override, causing the objects to be created immediately, defeating the entire purpose of the lazy-loading mechanism.

NotebookLM

The Solution Architecture: On-Demand Creation with an Init-Guard

Experiment 6: Final Solution (`Type_Safe_Lazy v4`)

Hypothesis: We can solve the premature creation issue by introducing a state flag that disables the on-demand logic during initialisation.

Implementation: A new instance variable, `_init_complete`, was added.

1. **`__init__` Begins**
   `self._init_complete` is set to False. All Type_Safe attributes are set to None.

2. **`super().__init__()` Called**
   The parent `__init__` runs. It internally accesses attributes, triggering `__getattribute__`.

3. **The Guard in Action**
   Our `__getattribute__` now checks the flag: `if not self._init_complete: return None`.
   It simply returns the None value without creating the object.

4. **`__init__` Completes**
   After `super()` returns, the last line of our `__init__` sets `self._init_complete = True`.

5. **Runtime Access**
   Any subsequent access to an attribute now passes the flag check, triggering the on-demand object creation as intended.

NotebookLM

From 1.9 Milliseconds to 90 Microseconds

Speedup

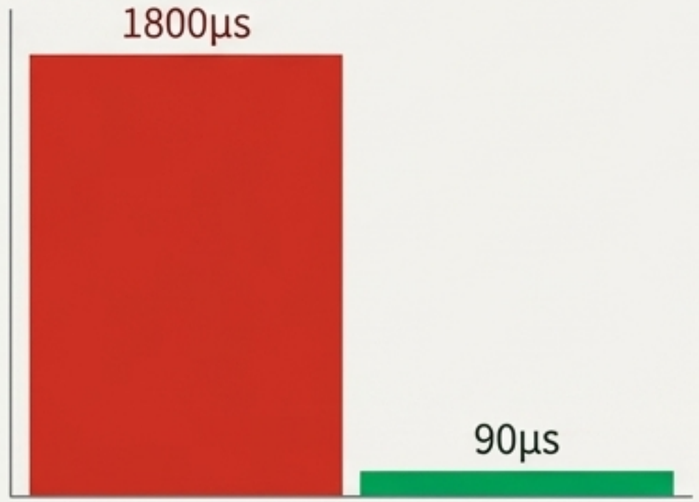
20x

Exceeding the 10x target

Construction Time

90µs

Down from 1.8ms



Objects Created Initially

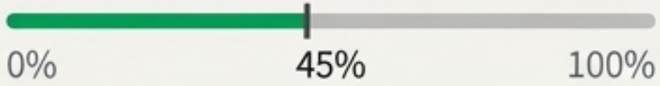
1

A 98% reduction from 47

Target Budget Used

45%

90µs achieved vs. 200µs goal



Drop-in Compatibility

100%

Fully compatible with all existing `Type_Safe` features

NotebookLM

Beyond ‘Lazy’: The Deliberate Choice of ‘On-Demand’

The technical solution was complete, but a collaborative review process led to a crucial refinement in its naming, improving clarity and adherence to codebase conventions.

Name	Assessment
Lazy	Standard computer science term, but describes the *mechanism*, not the user benefit.
Deferred	Professional and accurate, but still mechanism-focused.
Fast	Emphasises the benefit but is not descriptive of the behaviour.
OnDemand	Clearly describes the behaviour: "Objects are created when you ask for them.”

The Final Decision:

`Type_Safe__On_Demand`

- **Behaviour-Driven**: The name is self-documenting.
- **Convention-Aligned**: Follows the existing `Type_Safe__<Feature>` pattern in the codebase.
- **Clarity**: A developer encountering this class immediately understands its purpose.

Internal attributes were also renamed to match (e.g., `_lazy_init_complete` → `_on_demand__init_complete`).

NotebookLM

The Experimental Journey: A Map of the Investigation

File Created	Purpose	Outcome
`benchmark_type_safe.py`	Baseline measurement	✅ Confirmed 1.8ms, 47 objects
`lazy_type_safe_solution.py`	Attempt v1: Override `__cls_kwargs__`	❌ Slower than baseline
`test_none_defaults.py`	Validate if `= None` works	✅ Confirmed 71x speedup mechanism exists
`type_safe_lazy_v2.py / v3.py`	Attempt v2: Use `__getattribute__`	❌ Failed: Still 47 objects created
`debug_lazy.py`	Deep trace of `__init__` flow	🎯 Breakthrough: Found premature creation bug
`type_safe_lazy_v4.py`	Solution: `_init_complete` flag	✅ Success: 20x speedup achieved
`Type_Safe__On_Demand.py`	Final refined & renamed version	🚀 Final Deliverable

Total Iterations: 8 major attempts across 10 files, systematically progressing from failure to success in ~45 minutes.

NotebookLM

The Enablers: Tools for Rigour and Scale

The session's success depended on tools that provided both the full complexity of the production environment and the precision needed for performance engineering.

1. Real-World Code Access (`GitHub Digest Service`)

Source: 177 filtered files from the `OSBot-Utils` repository
Scale: 340,778 bytes of source code
Context Size: ~68,147 tokens

Benefit: The solution was developed against real-world complexity, not a simplified model.

2. Production-Grade Benchmarking (`Performance_Measure__Session)

Method: Fibonacci-based sampling (1,595 invocations)
Analysis: Automatic outlier removal and weighted scoring (60% median, 40% mean)

Benefit: Guaranteed that the results were reproducible and would meet performance criteria in the main codebase.

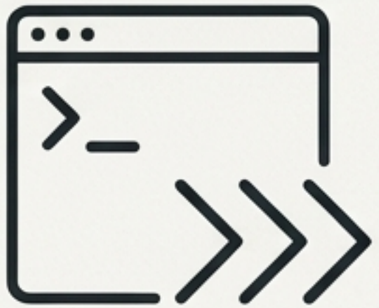
NotebookLM

A Repeatable Formula for High-Impact Development

This 20x optimization was not an accident. It was the result of a systematic process that combines a powerful interactive environment with rigorous engineering discipline.

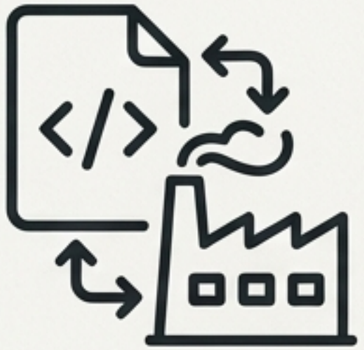
1. A Clear, Quantified Brief

Provides an unambiguous target and a reproducible starting point, saving hours of hours of discovery work.



2. An Executable Environment

Enables rapid, iterative cycles of hypothesis, implementation, and measurement.



3. Access to Real Source Code

Ensures solutions are robust and handle the true complexity of the production system.

4. Systematic, Hypothesis-Driven Debugging

Transforms failures and puzzles into the critical insights that lead to a breakthrough.

5. Collaborative Refinement

Elevates a functional solution into a clear, maintainable, and well-integrated piece of software craftsmanship.

NotebookLM