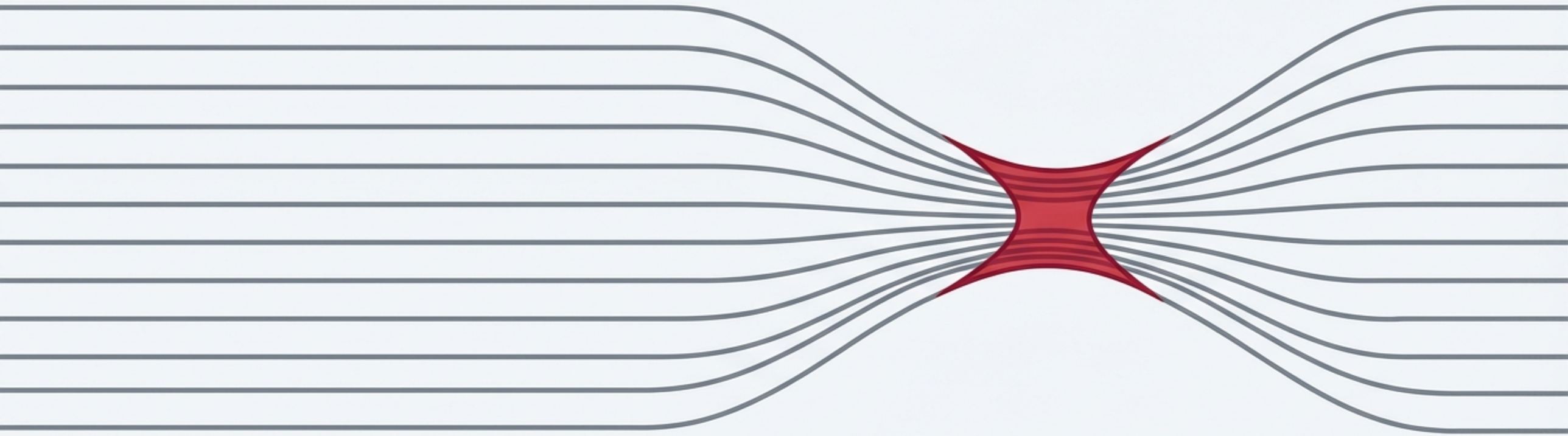
Optimising `Type_Safe`: Solving a 1.9ms Object Construction Bottleneck in the `Html_MGraph` Pipeline.

A data-driven investigation into recursive object creation and a workshop to define the path to a 10× performance improvement.

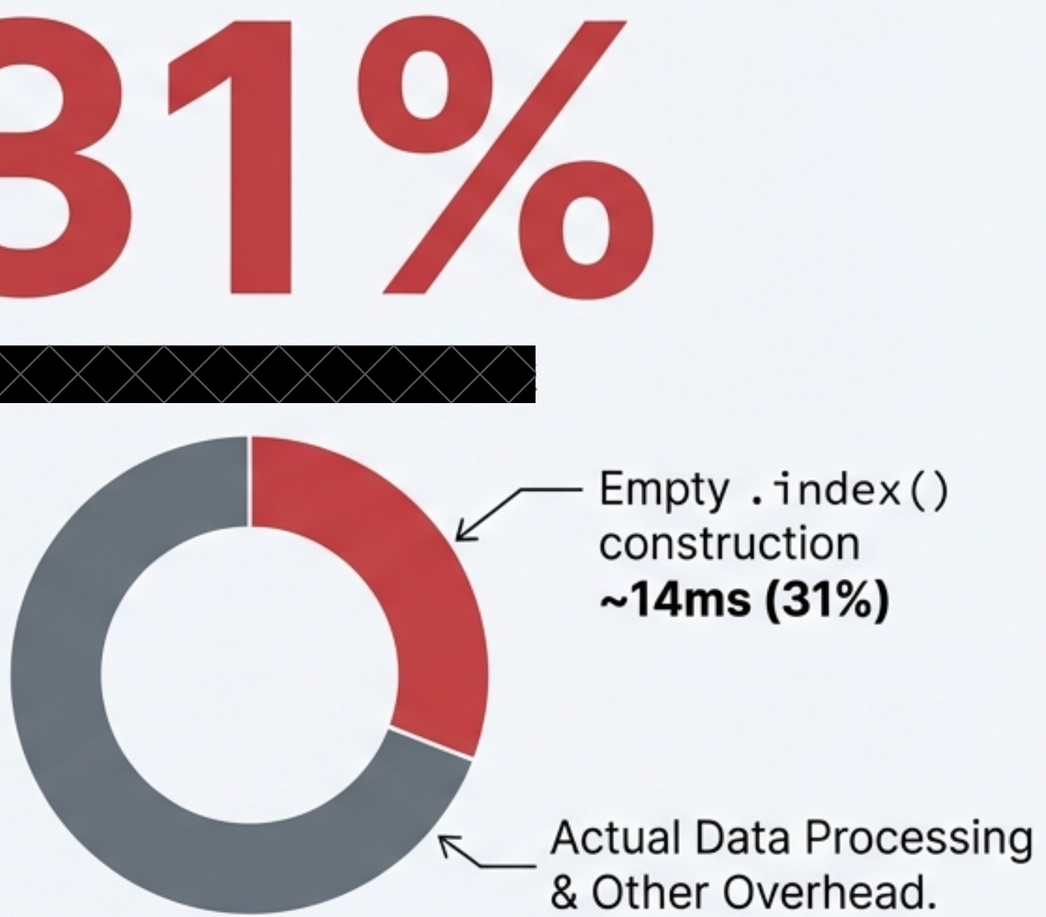


Processing a simple HTML document takes ~45ms, and 31% of that time is spent creating empty index objects.

System: Html_MGraph Conversion Service

Action: Converting a simple `<p>Hello</p>` document into a graph. `<p>Hello</p>`

Observed Total Time: ~45ms



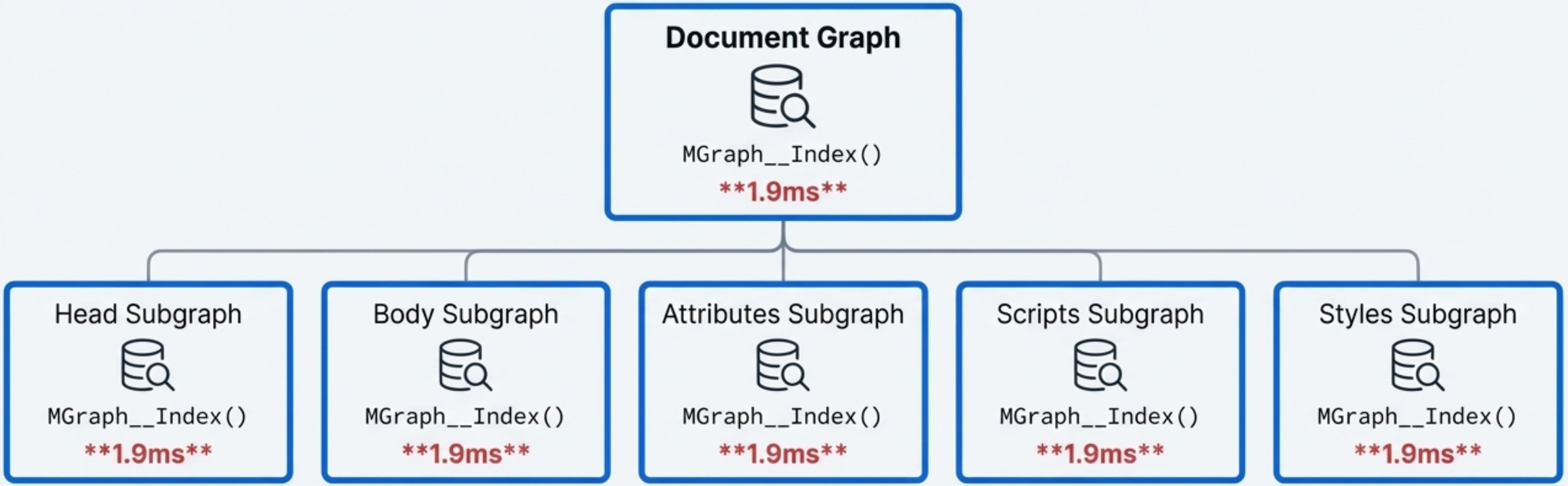
This initialisation overhead occurs before a single HTML element is processed.

A single, empty `MGraph__Index()` construction consumes 1.9ms, making it 73 times slower than the work it's designed to do.

Operation	Time
`MGraph__Index()` construction	**1.9ms**
Loading actual data into index	0.026ms
Ratio: Construction vs. Work	**73 : 1**

We spend 73× more time building the container than we do filling it.

The system creates six separate index objects on startup, costing over 11ms before any data is processed.



6 graphs × 1.9ms / graph = **~11.4ms**' mandatory overhead

This fixed cost is paid for every document, regardless of size, making our pipeline inefficient for small, rapid operations.

The root cause is `Type_Safe`'s recursive auto-initialisation, a feature designed for safety that triggers a cascade in nested structures.

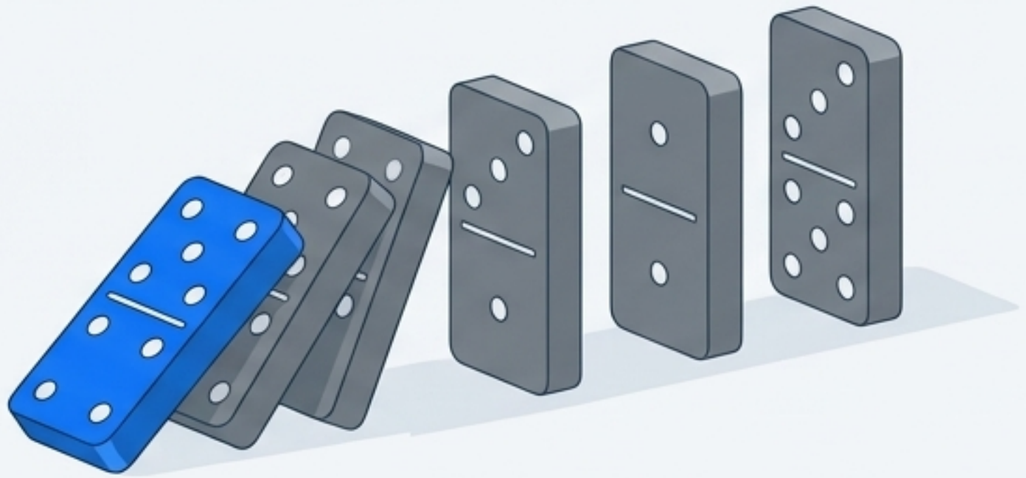
`Type_Safe` Framework Principle

```
class MyClass(Type_Safe):
    attribute: SomeType_Safe_Class
```

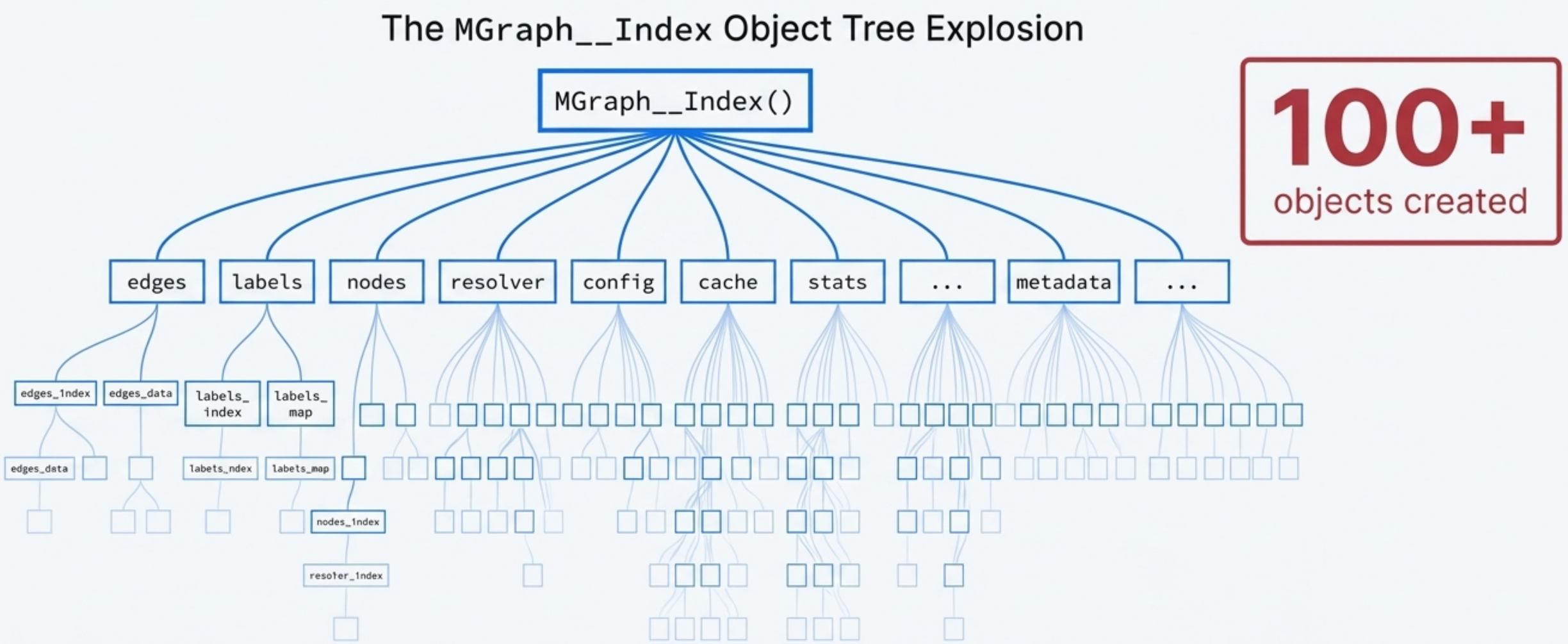
`Type_Safe` ensures that `attribute` is never `None`. It automatically calls `SomeType_Safe_Class()` during initialisation if no value is provided. This is the correct and desired behaviour for type safety.

The Cascade Effect

When `SomeType_Safe_Class` also contains `Type_Safe` attributes, its initialisation triggers their initialisation, and so on, recursively.



This cascade creates over 100 nested objects for a single `MGraph__Index` instance.

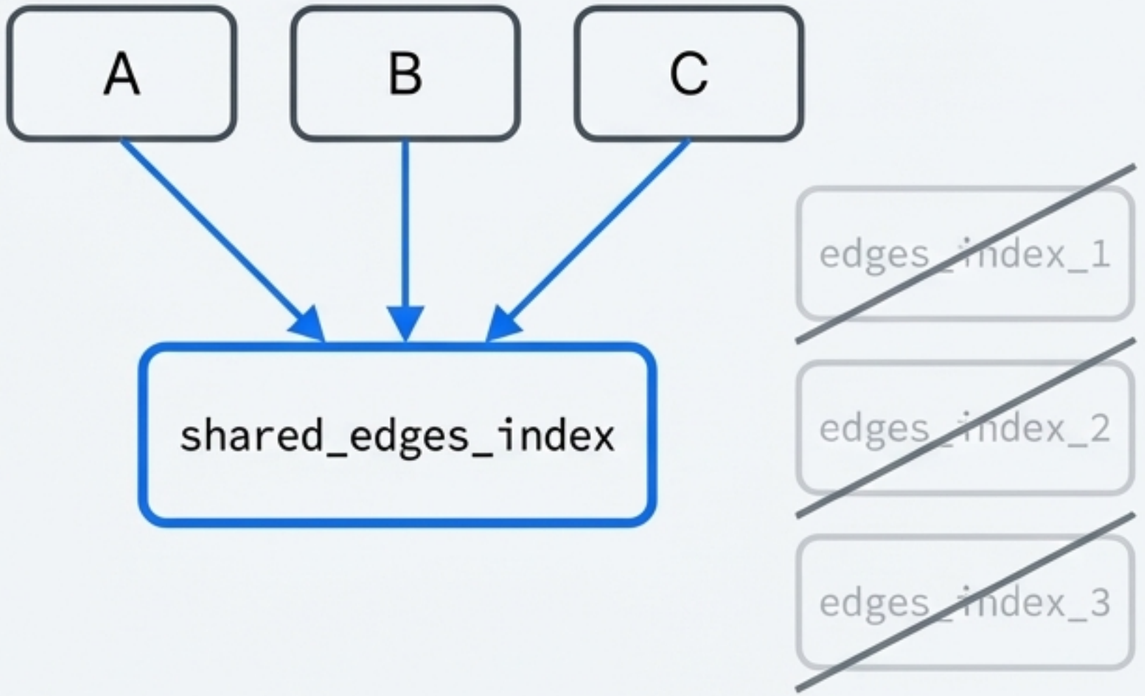


Even if each construction is only 15µs, `100 objects × 15µs/object = 1.5ms`.

The design intends to share sub-objects, but `Type_Safe` creates duplicate instances first, which are then immediately discarded.

What `Type_Safe` Does First

What the `__init__` Code Intends



The `_sync_index_data()` method later re-wires these attributes to point to shared instances, but the performance damage from creating the duplicates is already done.

Serialisation of a single `MGraph__Index` object proves massive sub-component duplication occurs on every call.


×5
`edges_index`


×4
`labels_index`

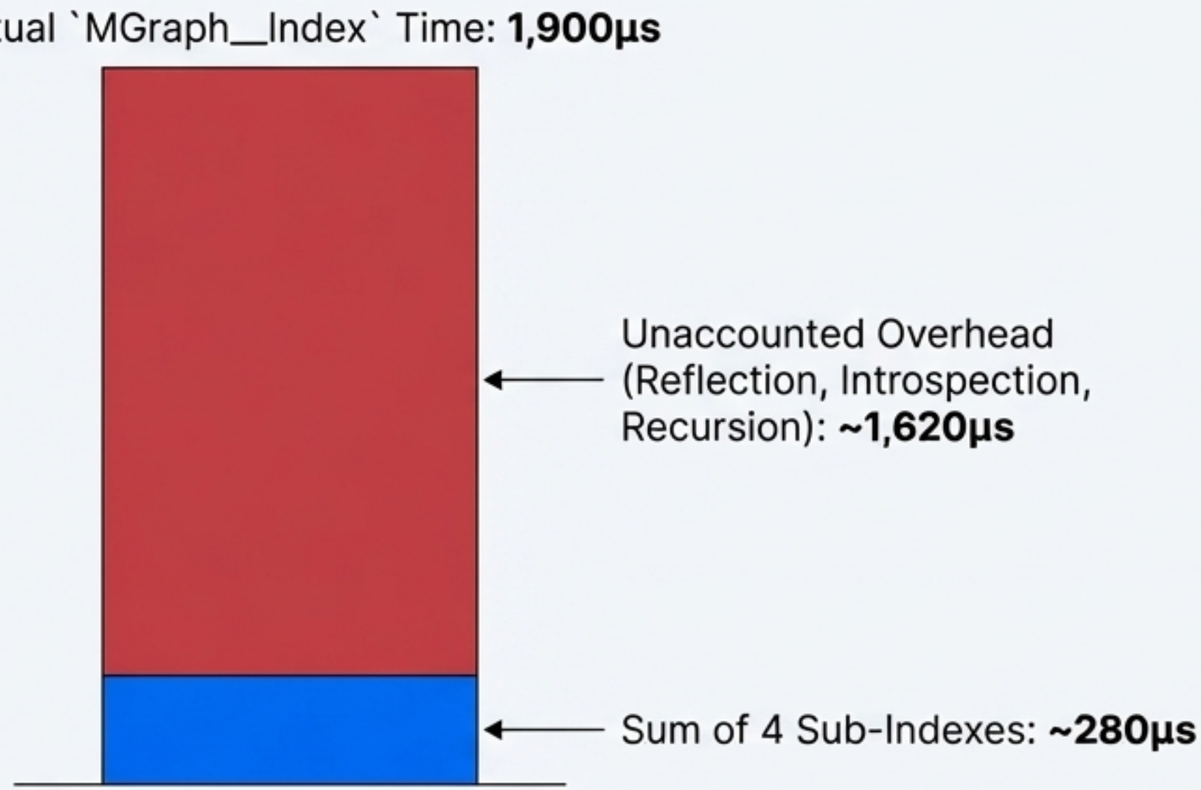

×4
`types_index`


×3
`values_index`


×2
`resolver`

These duplicated objects are created and memory-allocated, only to be orphaned moments later when references are synchronised. This is pure computational waste.

The 1.9ms cost is not in the sub-indexes themselves, but in 1.6ms of unaccounted `Type_Safe` overhead from the creation cascade.



Optimising the individual sub-indexes will have negligible impact. The solution must address the core `Type_Safe` initialisation mechanism.

Our goal is a 10× performance improvement, reducing the 1.9ms index construction to under 200µs.

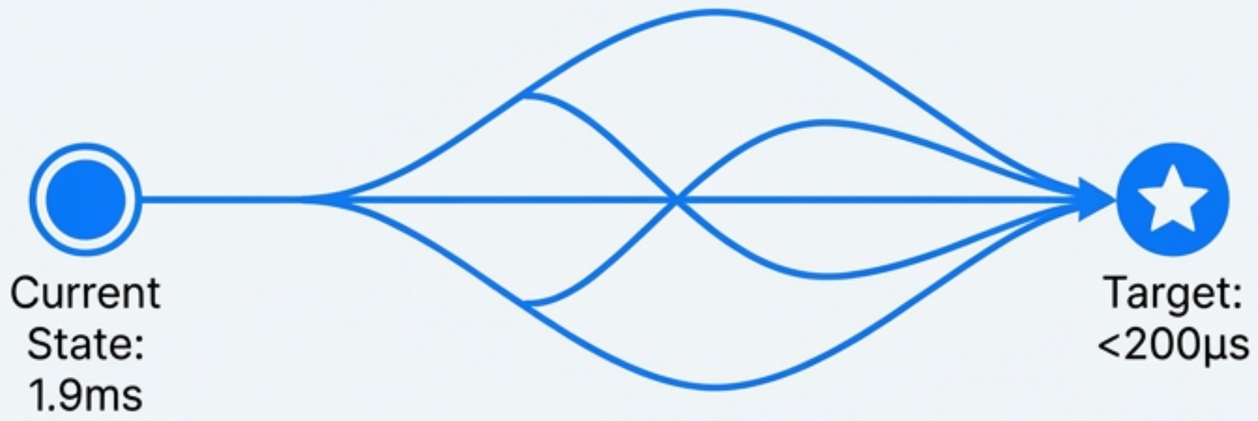
<200µs

Success Criteria

Metric	Current	Target
`MGraph__Index()` construction	1.9ms	**< 200µs**
6 index constructions	~11.4ms	**< 1.2ms**
Simple HTML total time	~45ms	**< 20ms**

Achieving **this target** will make index construction a negligible part of the overall processing pipeline.

To achieve our 10× target, we can explore several optimisation strategies, each with distinct trade-offs.



1. Change how we access attributes (Lazy Initialisation)
2. Change how we construct objects (Factory Pattern)
3. Change the framework itself (`Type_Safe` “Fast Mode”)
4. Change how we manage instances (Singleton / Flyweight)
5. Bypass the framework entirely (Manual `__init__`)
6. Change when we validate (Deferred Validation)

The following slides will outline the most promising paths.

Let’s examine three promising approaches: lazy initialisation, a framework “fast mode”, and an explicit factory pattern.

🕓 Approach 1: Lazy Attribute Initialisation

Concept: Only create sub-objects when they are first accessed.

✅ **Pros:** Pay-for-what-you-use performance.

❌ **Cons:** Changes access patterns (`.value` needed); loses immediate validation on assignment.

⚡ Approach 3: `Type_Safe` “Fast Mode”

Concept: A new class-level option to skip recursive initialisation.

✅ **Pros:** Opt-in; minimal code changes for consumers.

❌ **Cons:** Requires core framework modification; shifts initialisation responsibility to the developer.

🏭 Approach 2: Factory Pattern

Concept: Pre-create shared objects and pass them into the constructor.

✅ **Pros:** Guarantees no duplicate objects are ever objects are ever created.

❌ **Cons:** Can lead to complex factory/builder code; bypasses standard construction logic.

To select the best path forward, our discussion must focus on answering these critical technical questions.

1. **Performance Internals:** What is the precise breakdown of the 1.9ms? How much is `__init__` overhead versus object allocation versus reflection?

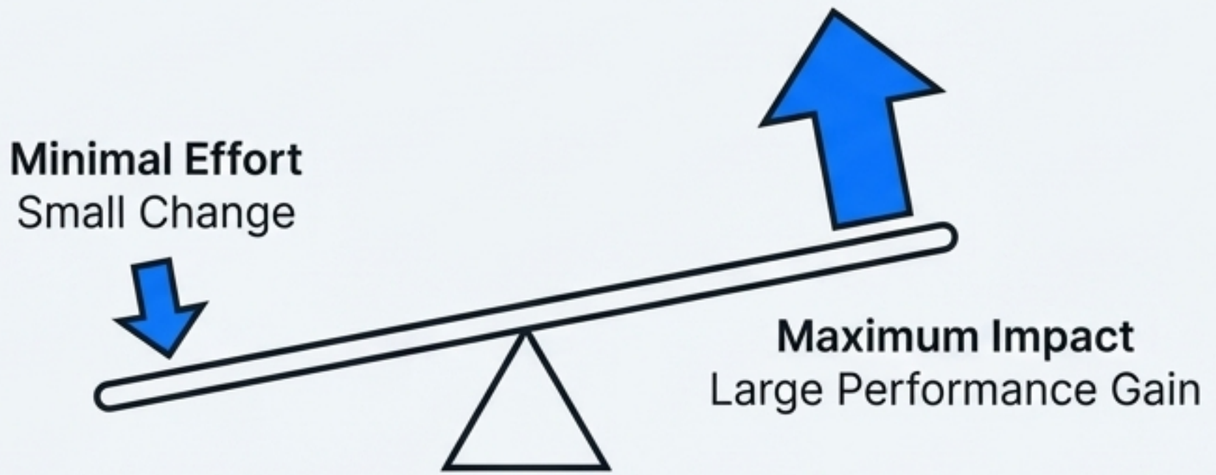
2. **Caching:** Can `Type_Safe` cache parsed type annotations at the class level to avoid expensive re-parsing on every `__init__`?

3. **Framework Support:** Is a “fast construction” mode feasible within `Type_Safe`? How would it handle validation (e.g., on first access, or via an explicit call)?

4. **De-duplication:** Can the framework be made smart enough to detect that the same `Type_Safe` class is required multiple times in a tree and automatically share a single instance?

The Central Challenge

What is the minimal change we can make to our existing code and frameworks that will deliver the maximum performance impact?



#Pragmatism #Impact #Maintainability #Performance