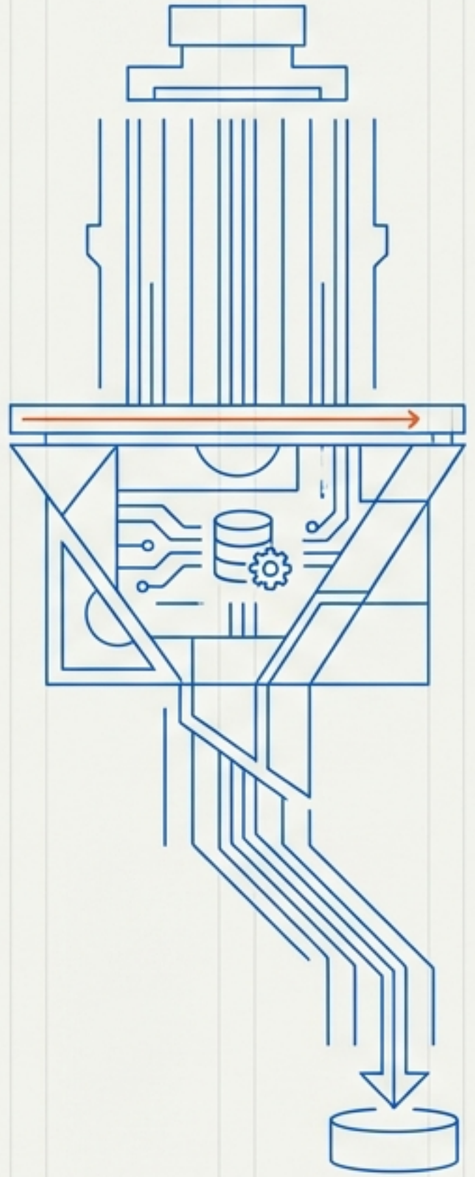
PHASE E_6: PERFORMANCE ANALYSIS & OPTIMISATION DEBRIEF

Pipeline Bottlenecks, Caching ROI, and Scalability Assessment



DOCUMENT TYPE: TECHNICAL RETROSPECTIVE
STATUS: DATA ANALYSED / ACTION PLAN READY
LOCALE: ENGLISH (UK)

System is Viable with High Cache ROI, Despite L2 Bottlenecks

The Win

48x

SPEEDUP FACTOR

Comparison of Cache Miss (82ms) vs. Cache Hit (1.7ms).

Key Takeaway:
Architecture delivers sub-2ms response times for repeated access.

The Challenge

83%

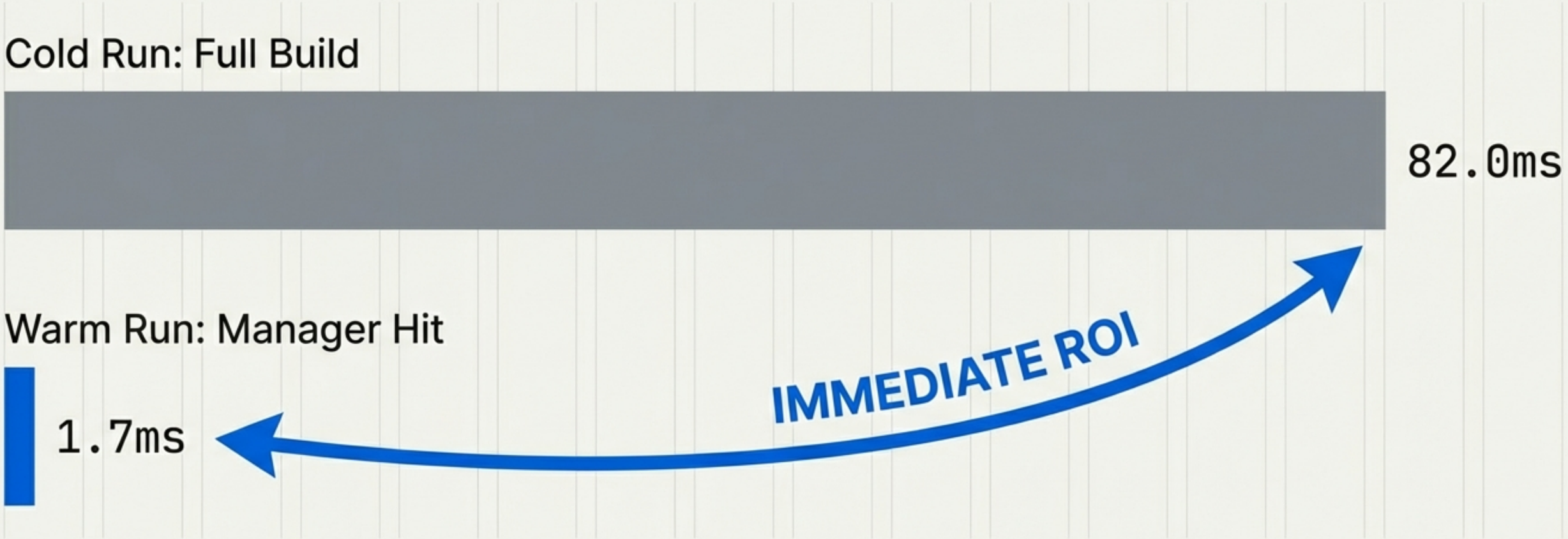
PIPELINE COST

Identified in L2 (MGraph Construction).

Key Takeaway:
Dictionary-to-Graph conversion is the primary constraint on write performance.

SUMMARY: Scalability is linear (~370µs/paragraph). Setup overhead is amortisable.

Caching Strategy Delivers Immediate Latency Reduction

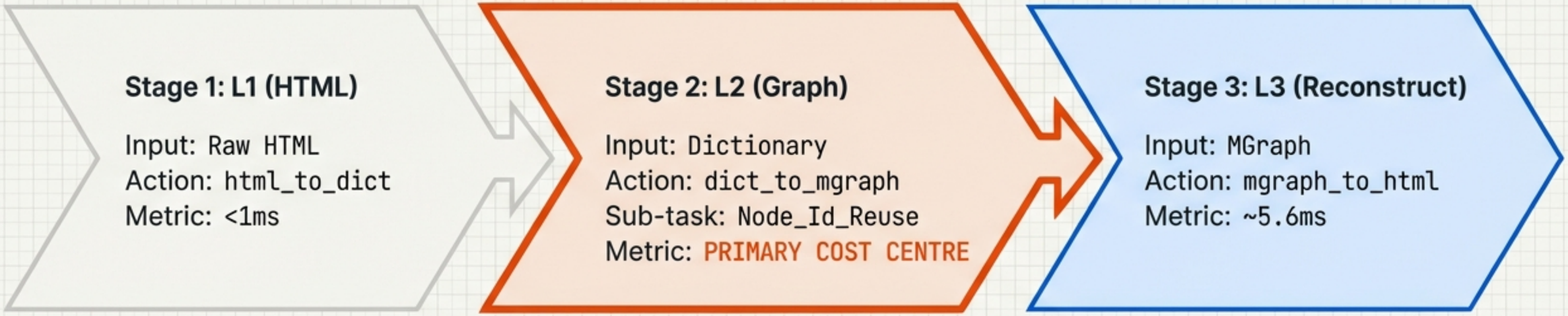


First request absorbs the 82ms build cost.

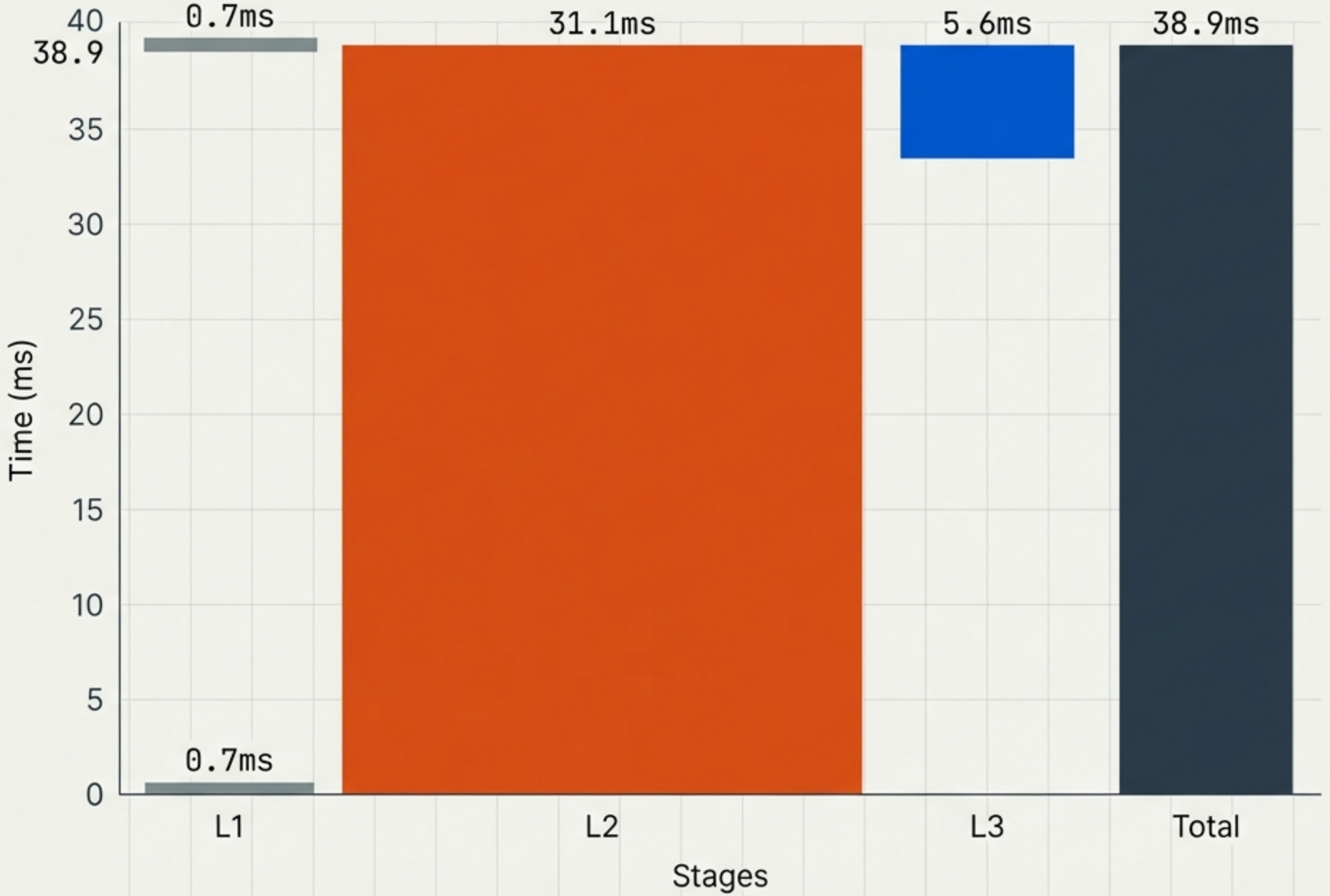
Subsequent requests bypass the pipeline entirely, retrieving the manager instance in <2ms.

Conclusion: The heavy lift of the pipeline is successfully decoupled from read performance.

Anatomy of the Processing Pipeline



L2 Construction Consumes 83% of Total Pipeline Time



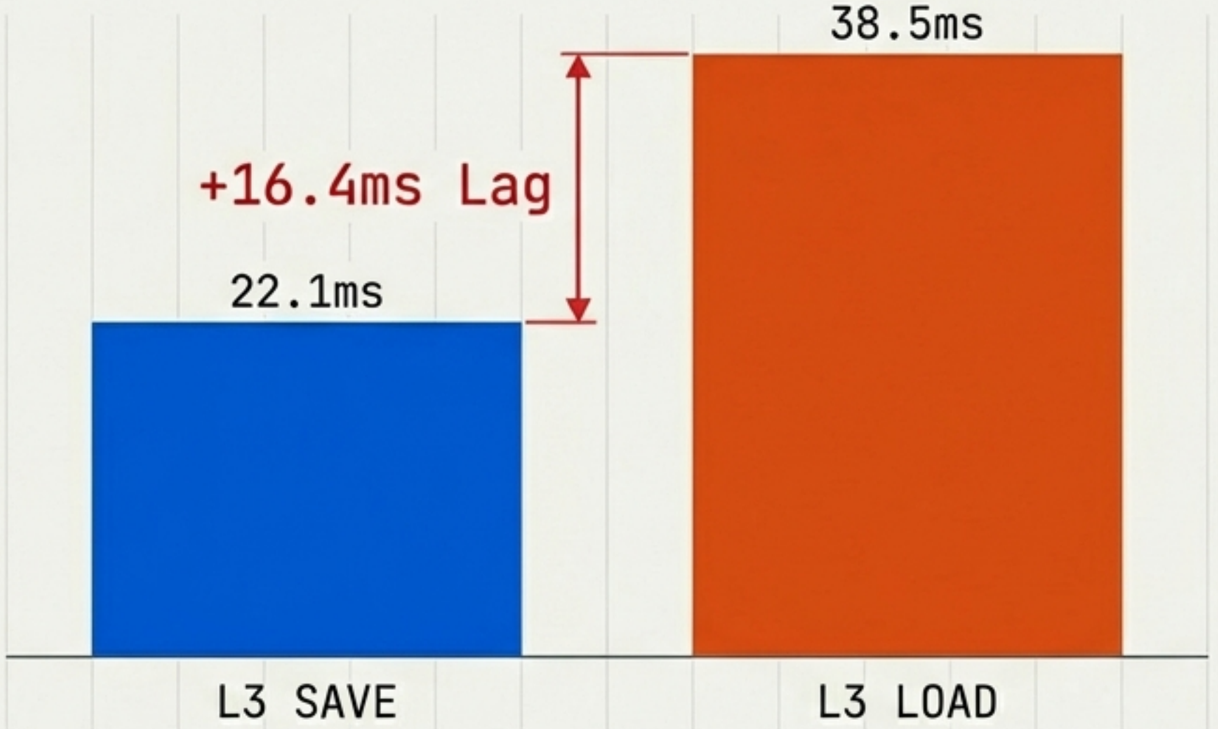
DIAGNOSTIC INSIGHT

- Metric: MGraph creation takes 31.1ms out of a 38.9ms total.

- Insight: The "`dict_to_mgraph`" conversion—specifically "`Node_Id_Reuse`"—is the mathematical bottleneck.

- Comparison: HTML parsing (L1) is effectively free.

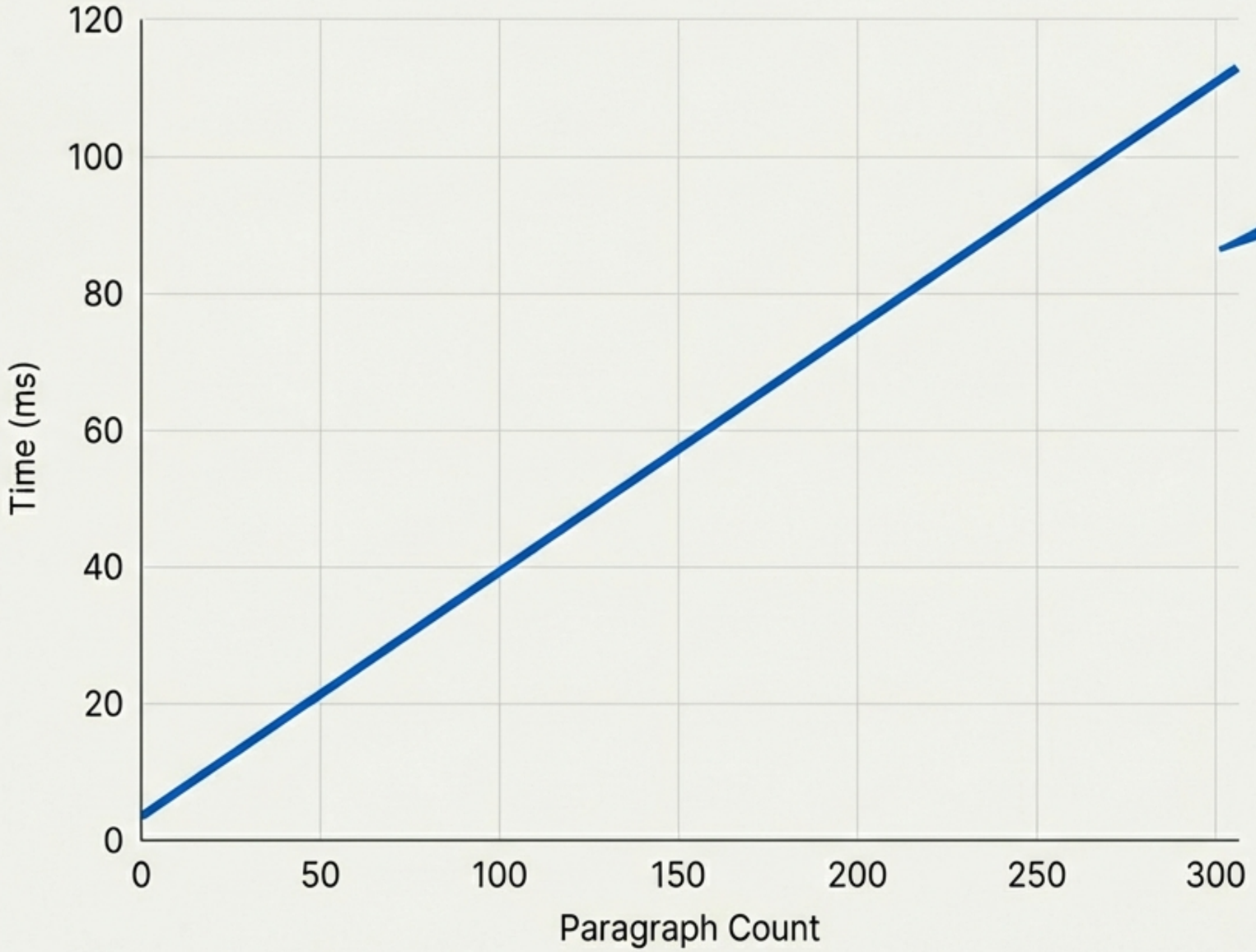
Deserialization is 74% More Expensive than Serialization



ANALYSIS:

- Contrary to expectation, reconstructing the object (Load) is slower than writing it (Save).
- **CULPRIT:** `Html_MGraph__Document.from_json()`
- **IMPACT:** This affects the system's ability to 'hydrate' stored graphs quickly.

Performance Scales Linearly with Document Size



`Time ≈ 4ms + (0.37ms × Paragraphs)`

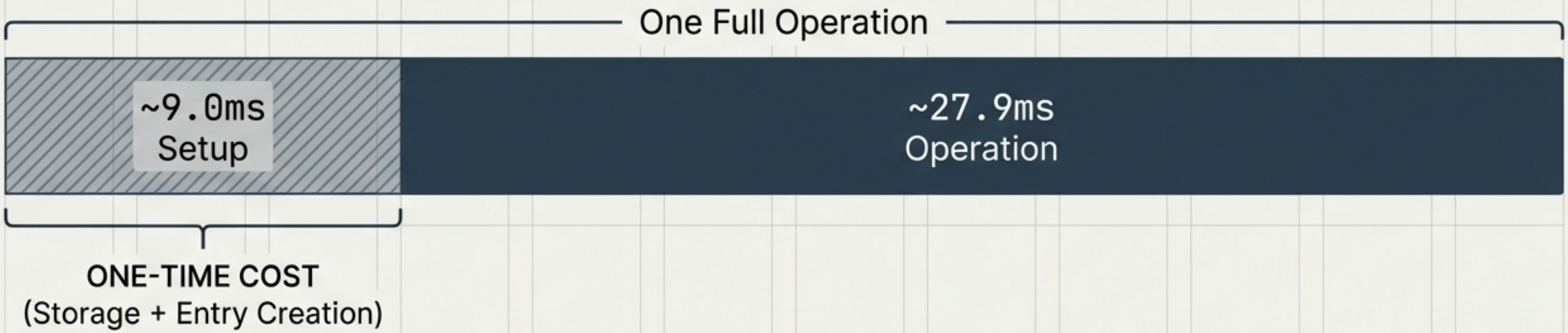
INSIGHT BOX

SLOPE: ~370µs per paragraph.

FIXED OVERHEAD: ~4ms (intercept).

CONCLUSION: Predictable performance budgeting is possible for larger documents. There is no exponential complexity risk.

Setup Overhead is Significant but Amortisable



- Setup accounts for roughly a quarter of a single operation's time.
- **MITIGATION:** Reusing storage and layer instances eliminates this cost for subsequent operations.
- **NOTE:** Layer creation itself is negligible (6µs).

High-Impact Optimisation Targets

PRIORITY 1 - HIGH IMPACT

Profile Deserialization

TARGET CODE

`Html_MGraph__Document.from_json()`

REASON

Accounts for 77% of load time.

PRIORITY 2 - HIGH IMPACT

Profile Node Reuse

TARGET CODE

`Html__To__Html_MGraph__Document__Node_Id_Reuse`

REASON

Drives the 83% bottleneck in L2 build time.

PRIORITY 3 - MEDIUM IMPACT

Lazy Loading

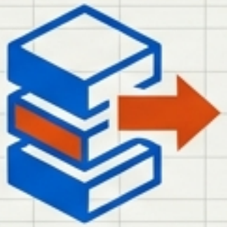
ACTION

Only deserialize MGraph when technically required.

REASON

Avoids the expensive L3 load cost on read-only operations.

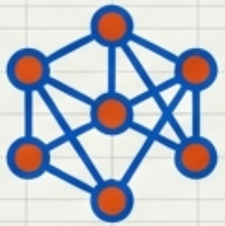
Operational Efficiencies: Batching & Pooling



Batch Cache Operations

Current State: ~8 requests per cycle.

Recommendation: Reduce to ~2 via batching.



Connection Pooling

Current State: ~1.0ms overhead per request.

Recommendation: Maintain persistent connections.

Instance Reuse

Current State: 9ms setup cost per run.

Recommendation: Keep storage layers alive to permanently bypass setup.

NON-ISSUES: Layer creation (6µs) and HTML parsing are already fully optimised and require no action.

Performance Budget Targets for 100-Paragraph Documents

Metric	Current Performance	Status/Target
Cache Hit	~1.7ms	🟢 Excellent (Maintain)
Full Pipeline Build	~39ms	🟡 Target: <50ms (Acceptable)
L3 Deserialization	~38ms	🔴 Target: <20ms (Needs Optimisation)
First Request (Cold)	~82ms	🟡 Acceptable for initial fetch

Summary of Key Findings

1. **MGraph Construction** is the primary bottleneck (83% of pipeline).
2. **Deserialization** is unexpectedly 63-74% more expensive than serialization.
3. **Caching** is highly effective, delivering a 48x speedup.
4. **Scaling** is linear and stable (no exponential risk).
5. **Setup Costs** (25%) can be negated through **instance reuse**.

Appendix: Raw Benchmark Data

E_6_1 Results
A_01__L1__html_to_dict: 700µs (1.9%)
A_02__L2__dict_to_mgraph: 31.1ms (83.2%)
A_03__L3__mgraph_to_html: 5.6ms (15.0%)
B_01__full_pipeline: 38.9ms (100%)

E_6_3 Results
C_01__manager_miss: 82.0ms
C_02__manager_hit: 1.7ms
B_03__load_L3: 38.5ms
A_04__save_L3: 22.1ms

E_6_5 Cost Breakdown
A_01__save_pipeline: 39.4ms
A_02__load_pipeline: 27.9ms
B_05__full_setup: 9.0ms