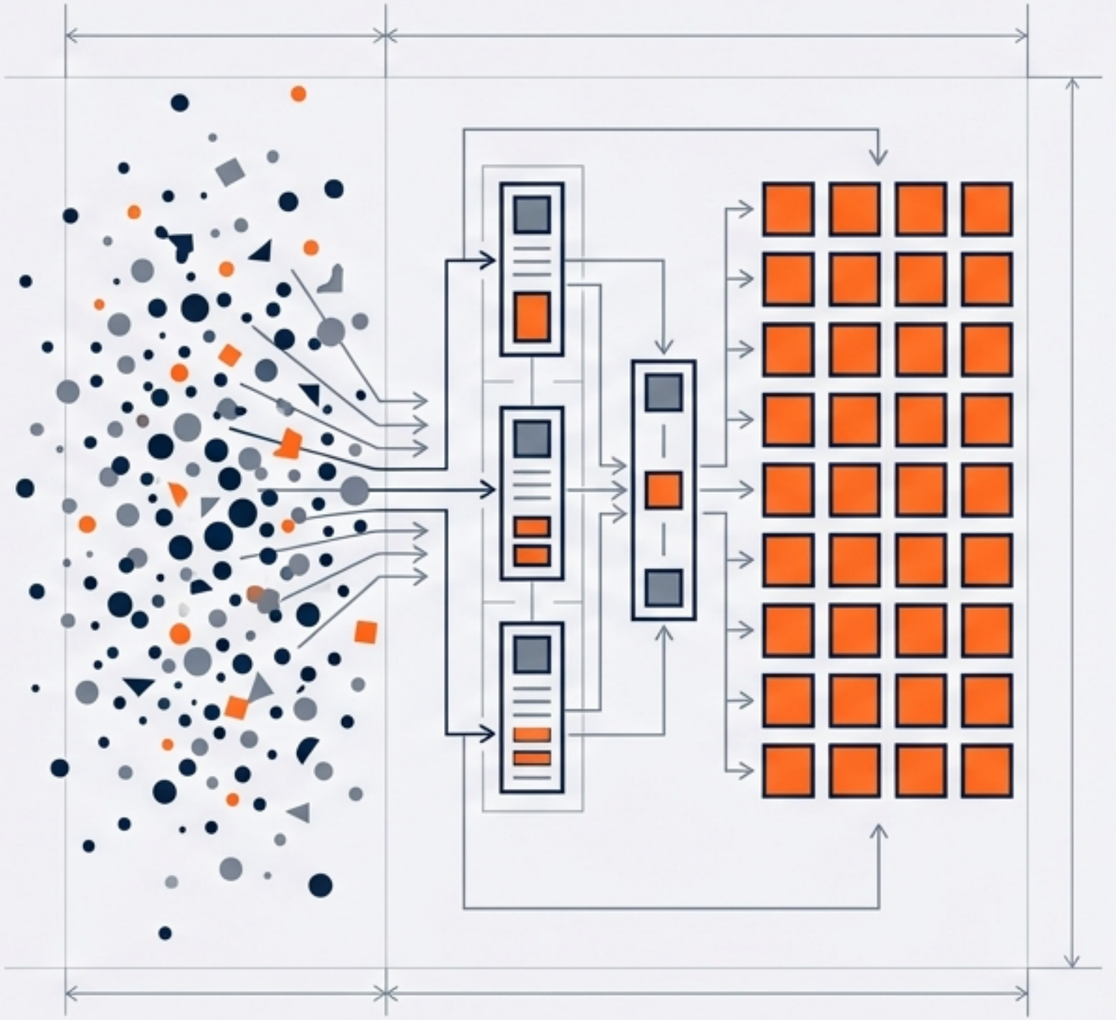
Architecture Modernization

Implementing the LETS Framework & Intelligent Cache Layer



Status

Architecture Design — Ready for Implementation (January 2026)

Brief

LLM Brief v1.4.42

Executive Summary

Building a Transparent, High-Performance Transformation Pipeline

Performance (Caching)

Optimization by reusing previously computed results at every stage. We transition from fresh computation to intelligent retrieval.

Provenance (Explainability)

A full audit trail of exactly what happened at each step. We replace ‘black-box’ processing with a fully transparent history.

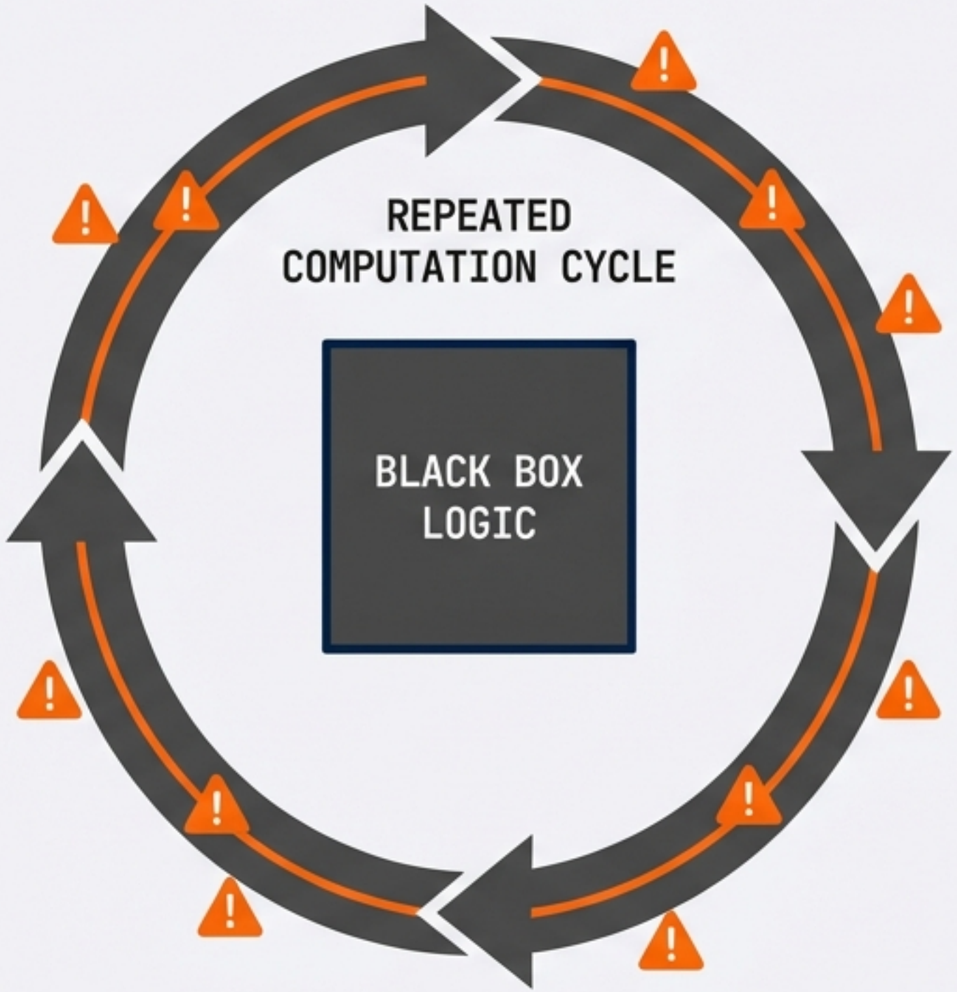
Modularity (LETS)

Composable pipelines where transformations consume the output of previous steps. New products are configured, not coded.

CORE OBJECTIVE: TRANSITION FROM OPAQUE, REPETITIVE SCRIPTING TO A MANAGED, AUDITABLE ENTERPRISE SERVICE.

The Cost of the Current Architecture

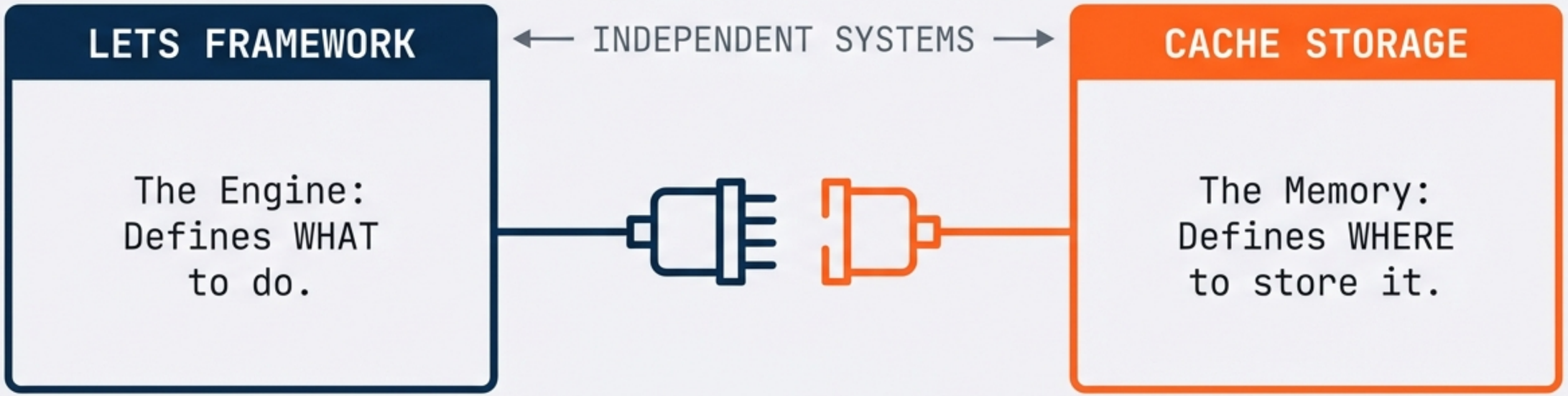
Operational Friction & Technical Debt



PROBLEM	IMPACT
Zero Caching	Identical requests recompute everything. Wasted CPU/Memory.
No Visibility	Impossible to inspect intermediate states (HTML Dict/MGraph).
Tight Coupling	Logic is embedded in handlers. Changes break the entire flow.
Missing Provenance	No record of what transformations were applied or when.

The Core Insight: Separation of Concerns

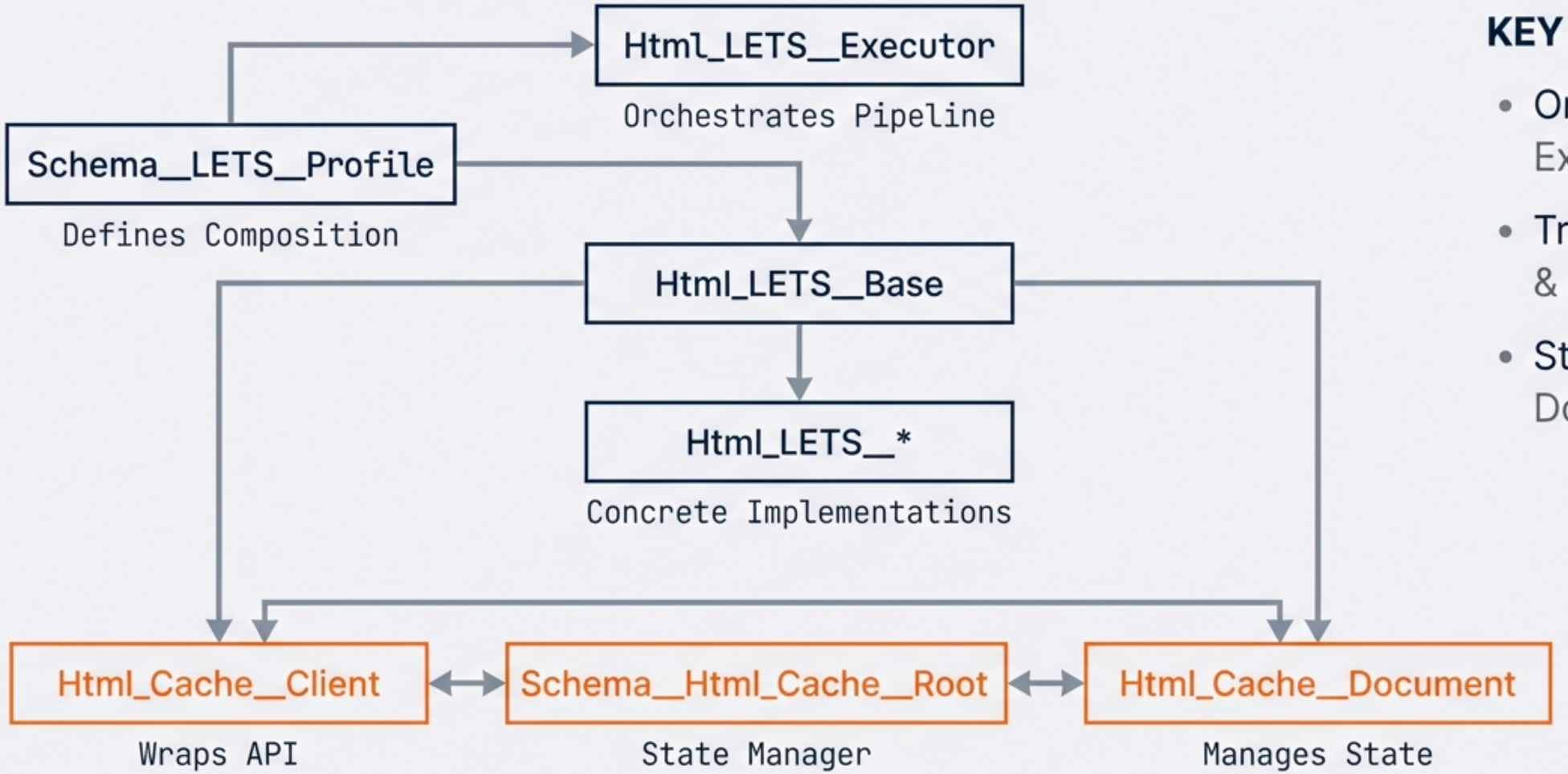
Decoupling Logic from Storage



- LETS is independent from Caching.
- Cache is merely one possible target for Load/Save phases.
- Enables testing logic without infrastructure and swapping storage backends.

System Architecture & Component Responsibilities

The Ecosystem View

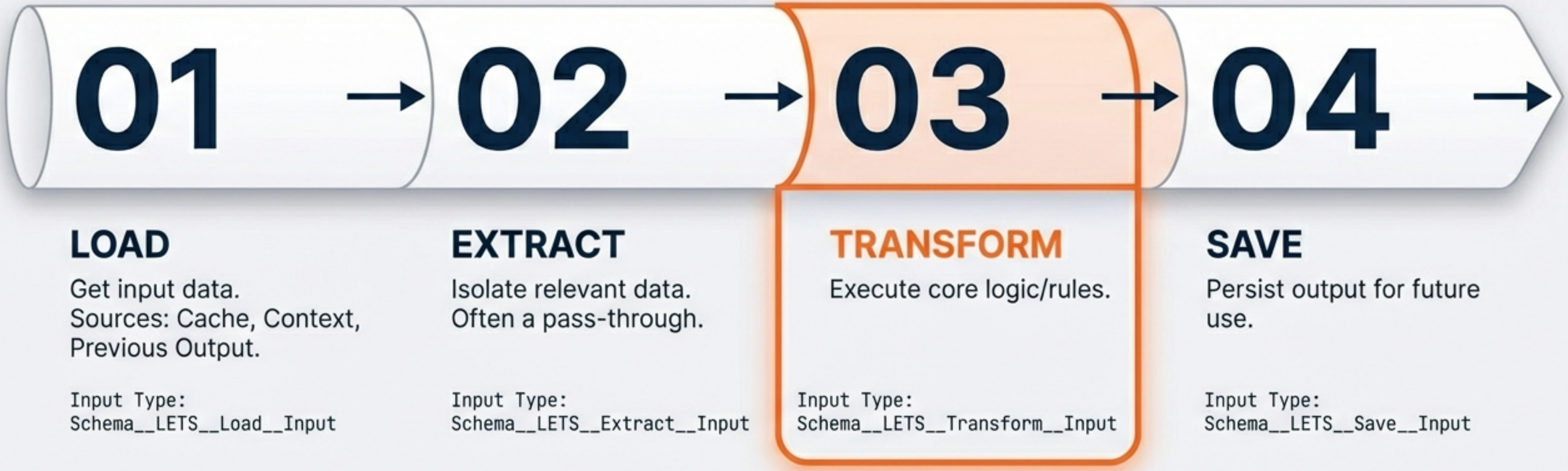


KEY COMPONENTS:

- Orchestrator: Executor & Profile
- Transformation: Base & Concrete Classes
- State Manager: Root Document & Client

The LETS Framework

A Standardized Transformation Pattern



Standardization: Every transformation, regardless of complexity, adheres to this 4-phase lifecycle.

Pipeline Composition via ‘Profiles’

Configurable Workflows

Schema_LETS_Profile

1. Step A:
   `Html_LETS_Html_To_Dict`

2. Step B:
   `Html_LETS_Dict_To_MGraph`

3. Step C:
   `Html_LETS_MGraph__To_Analysis`

THE MECHANISM:

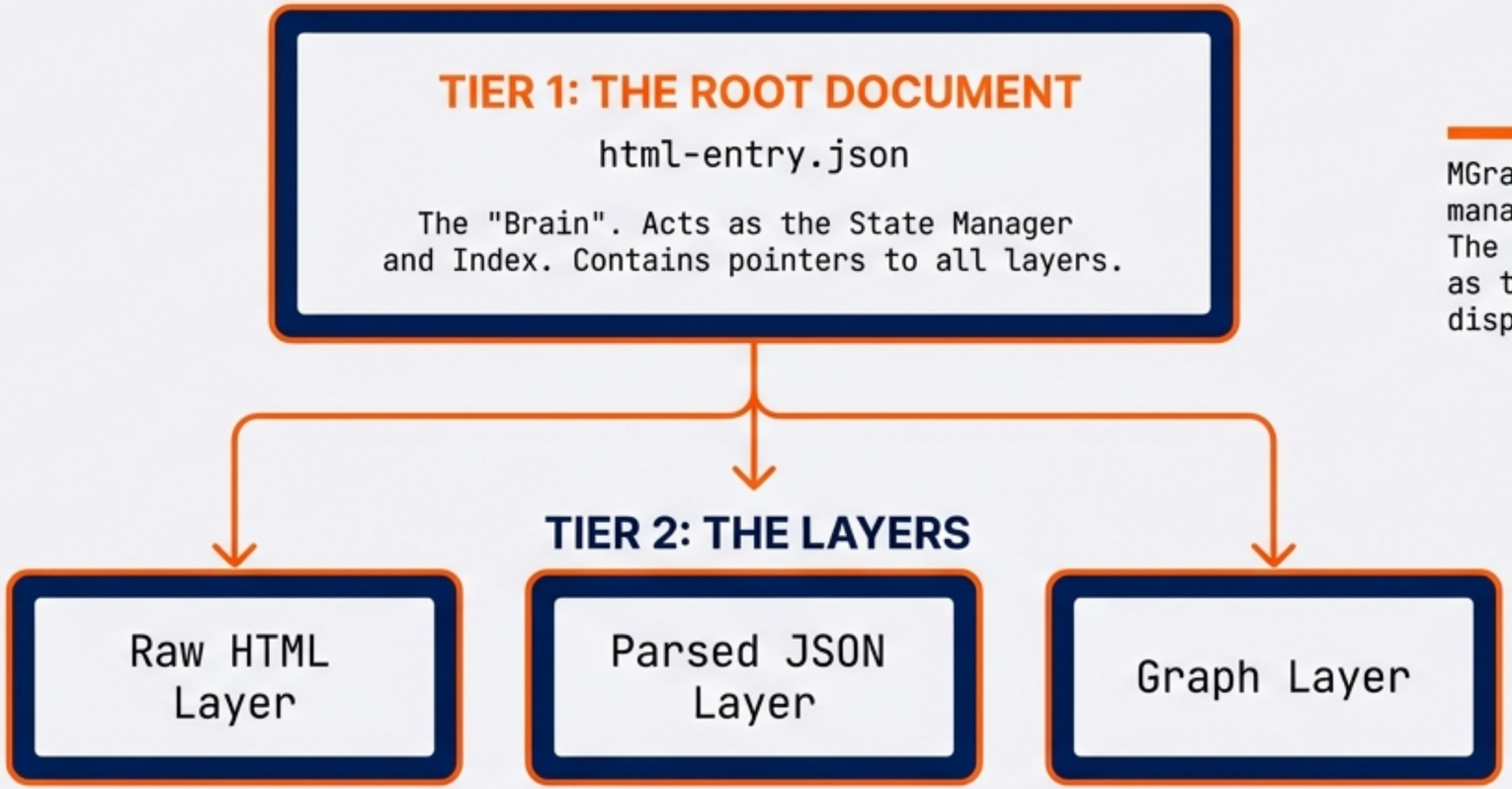
We do not hardcode pipelines. We configure them. Profiles enable **Composable Pipelines** where each each transformation consumes the output of the previous one.

BUSINESS VALUE:

New products or outputs can be created simply by rearranging existing LETS blocks—no new code required.

Cache Storage Architecture

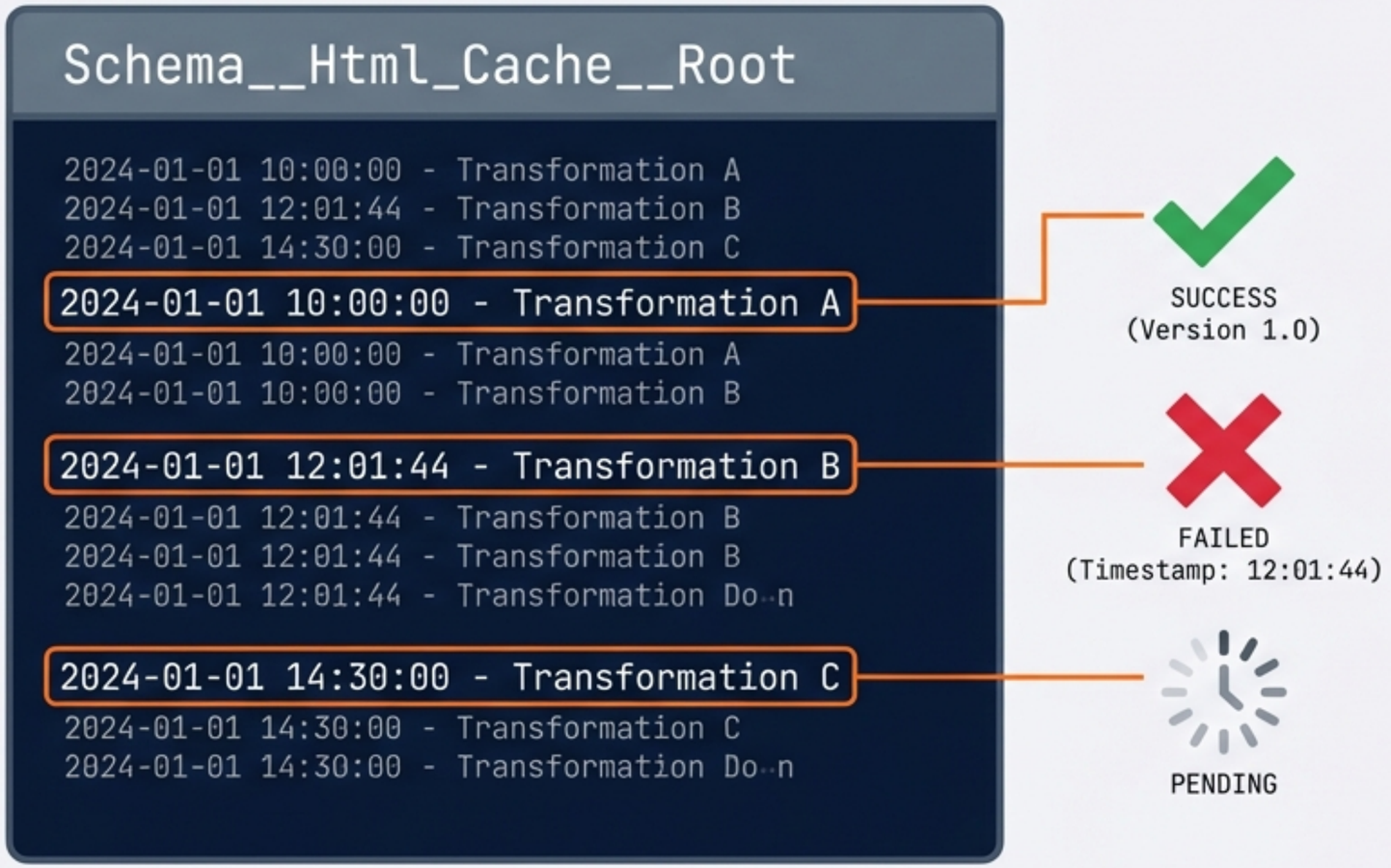
Two-Tier Storage Model



MGraph-AI Cache Service manages the references. The Root Document serves as the map to the dispersed data.

State Management: The Single Source of Truth

Provenance & Auditability



ROLE OF THE ROOT DOCUMENT

1. Tracks exactly which version of code created which piece of data.
2. Records status of every layer (Success/Fail).
3. Enables targeted debugging without re-running the whole pipeline.

“We replace ‘I think this happened’ with ‘The state manager records this happened’.”

Engineering for Reliability: The Type System

Zero Raw Primitives

Legacy / Unsafe

```python
def process(data: dict):
    name = data['name']  # KeyError risk
    id = str(data['id']) # Casting risk
```

New Architecture / Safe

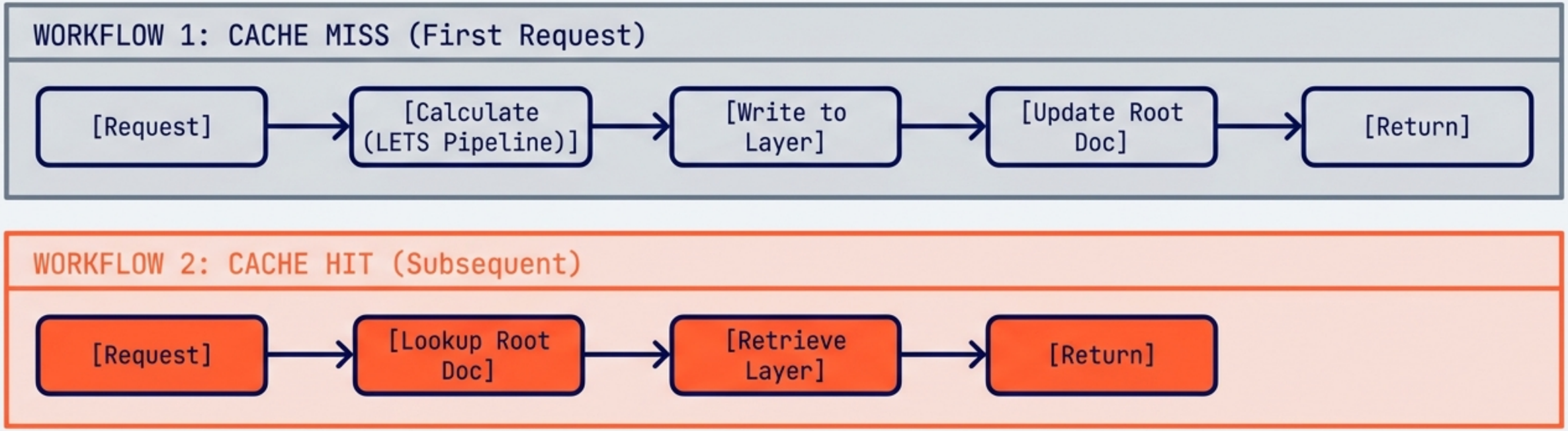
```python
class Safe_Str__LETS__Name(Safe_Str): ...
class Safe_Str__Cache_Key(Safe_Str): ...

def process(name: Safe_Str__LETS__Name):
```

Key Principles:

1. All data wrapped in Safe_* classes.
2. Enum values ARE class types (Type[X]).
3. Validation logic is baked into the types. If it initializes, it is valid.

Data Flow: Optimizing for Performance Cache Miss vs. Cache Hit

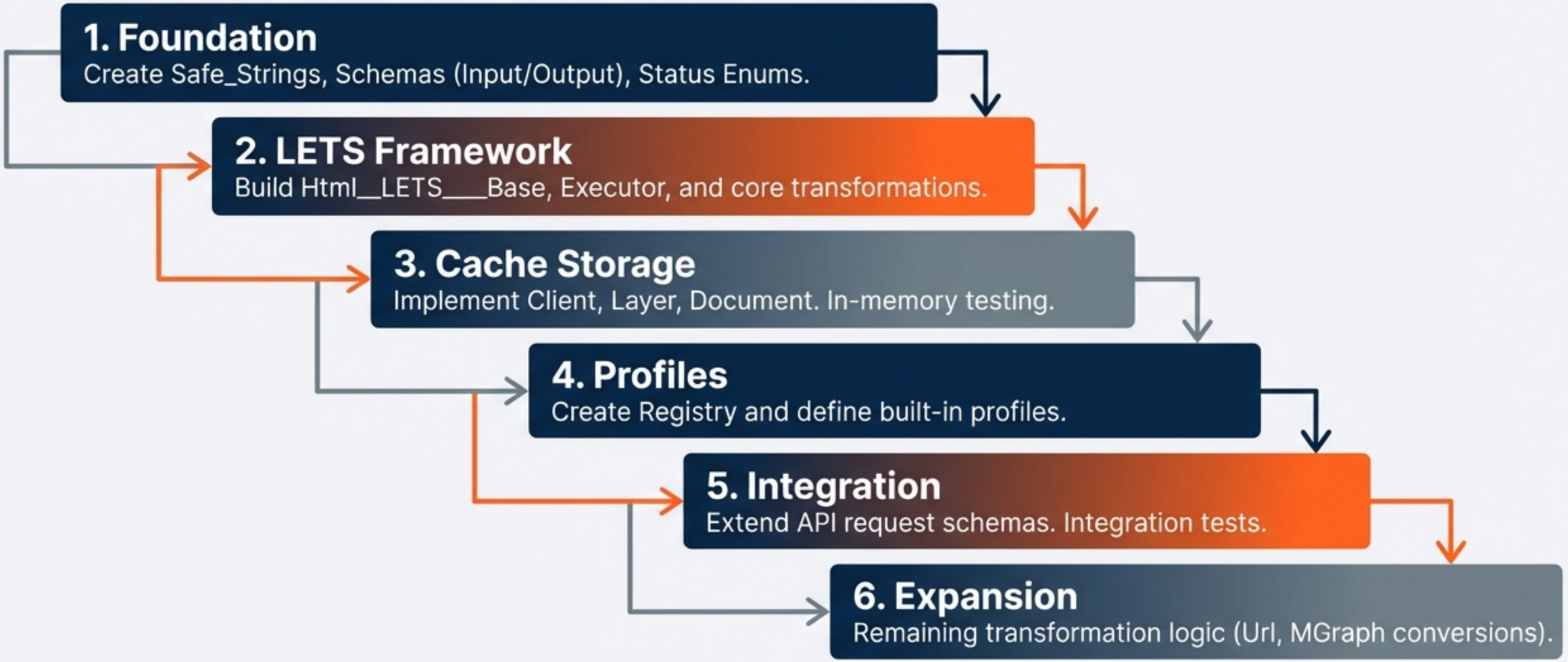


ROI: The 'Hit' path bypasses all calculation logic, resulting in instant response times and zero compute cost.

Roadmap to Production

READY FOR DEV

Implementation Checklist



Summary: What Success Looks Like

Business & Technical Impact

SPEED

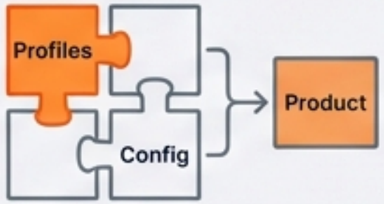
Cached results eliminate redundant compute.

TRUST

Full provenance via the Root Document. We know the history of every byte.

AGILITY

Composable Profiles allow new products via config, not code.



STABILITY

Type-safe architecture prevents entire classes of runtime errors.

LETS DEFINES ‘WHAT’. CACHE DEFINES ‘WHERE’. INDEPENDENCE DRIVES RELIABILITY.