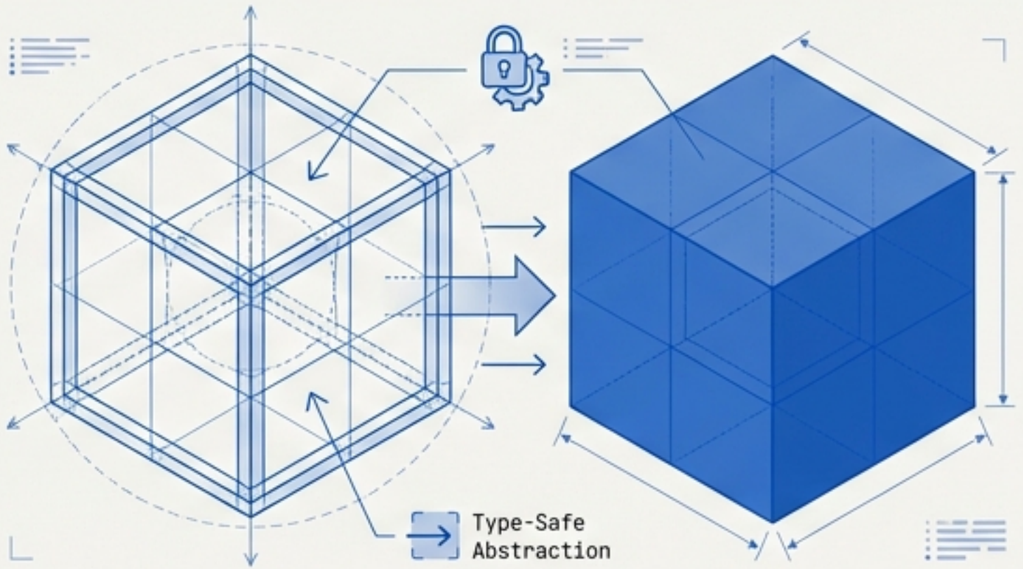
Phase E_3 Debrief: Cache Service Storage Integration

Implementing a Pluggable, Type-Safe Storage Abstraction Layer for MGraph-AI.

DATE: January 2025
STATUS: Completed
DURATION: Single session with iterative refinement



Executive Summary & Objectives

We aimed to create a flexible storage system for performance analysis artifacts. We succeeded by integrating deeply with the MGraph-AI ecosystem rather than building parallel structures.

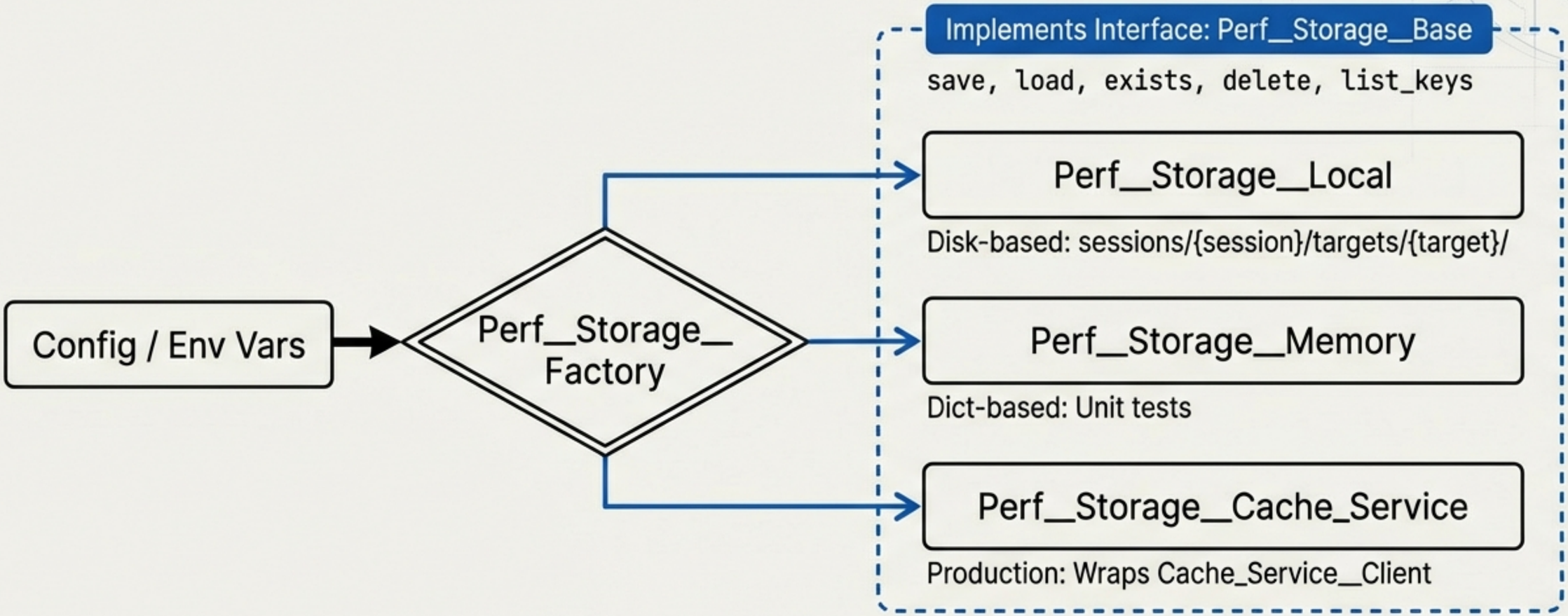
Objectives

1. Create pluggable backend system.
2. Integrate with MGraph-AI Cache Service (Production).
3. Maintain Local/Memory backends (Dev/Test).
4. Enforce Type_Safe compliance.

Key Achievements

- **Architecture:** Three functional backends (Local, Memory, Production).
- **Code Volume:** 40 Files Delivered (25 source, 15 test).
- **Headline Win:** Successful integration with the official MGraph-AI Cache Service client library.
- **Quality:** Zero mock maintenance due to realistic in-memory testing.

System Architecture Overview



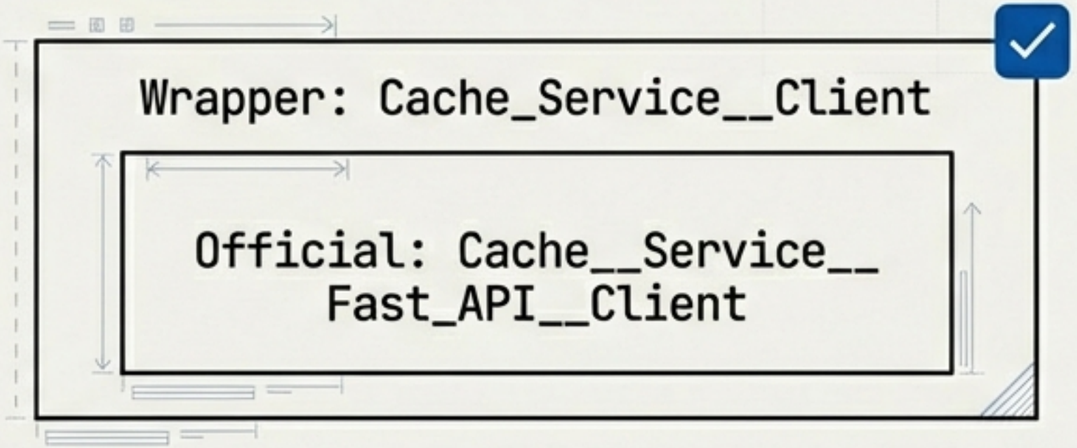
The Factory pattern isolates application logic from storage implementation. The calling code never knows—or cares—where the data lives.

Evolution of the Client Strategy: Wrapper vs. Direct REST

Iteration 1 - Abandoned

- Implementation: Built using 'requests' library.
- **Issues**: High maintenance burden.
- **Issues**: Duplicated authentication logic.
- **Issues**: Brittle mock requirements.

Iteration 2 - Adopted

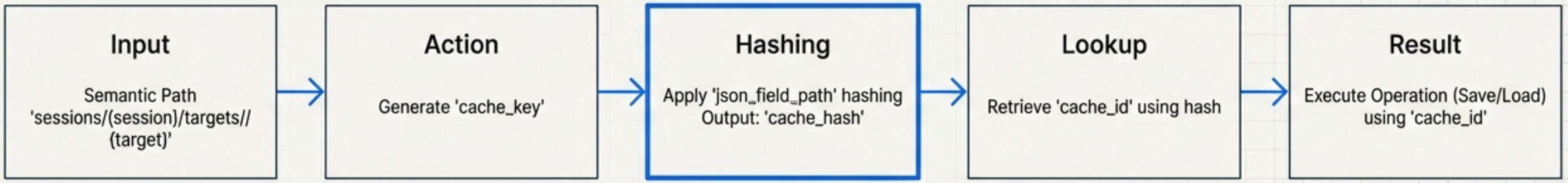


- Implementation: Wraps official client.
- **Benefits**: Consistent API with MGraph-AI services.
- **Benefits**: Built-in auth handling.
- **Benefits**: Type-safe schemas.
- **Benefits**: Support for REMOTE, IN_MEMORY, LOCAL_SERVER modes.

Bottom Line: We chose to leverage existing infrastructure rather than reinventing the wheel.

The Content Addressing Challenge: Hash-Based Lookup

Problem Statement: The Cache Service is content-addressable. It does not natively understand semantic paths like 'sessions/session_1/targets/target_A'.



Quality Assurance: Shifting from Mocks to In-Memory

Metric	Initial Approach (Mocking)	Final Approach (In-Memory Service)
Implementation	Used "unittest.mock"	Real "Cache__Service" in memory mode
Reliability	Brittle tests, low confidence ✗	High confidence, real API contract ✓
Maintenance	High burden	Zero mock maintenance
Performance	N/A	Startup time ~100ms

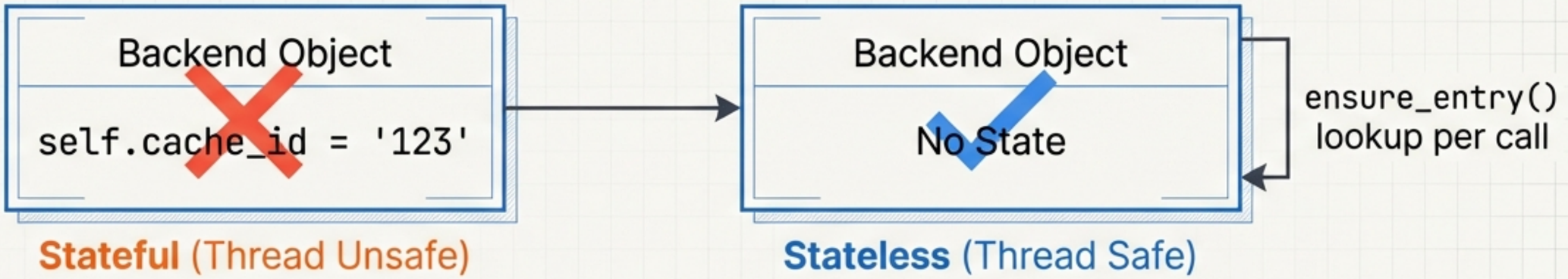
> “Testing against the real implementation provides higher confidence with zero mock maintenance.”

Key Benefits

- ✓ Catches real API contract issues.
- ✓ Eliminates mock drift.
- ✓ Ensures integration behavior validity.

Backend Design – Statelessness & Configuration

Stateless Backend Design



Benefit: Predictability. The caller controls the caching behavior.

Configuration Simplification

Initial: Explicit URLs in config files.

Final: Configuration handled by client library via environment variables.

`AUTH__TARGET_SERVER__CACHE_SERVICE__KEY_VALUE`

Benefit: Leverages standard env vars; removed redundant logic in 'Perf__Storage__Config'.

Retrospective: Validated Patterns vs. Discarded Approaches

What Worked

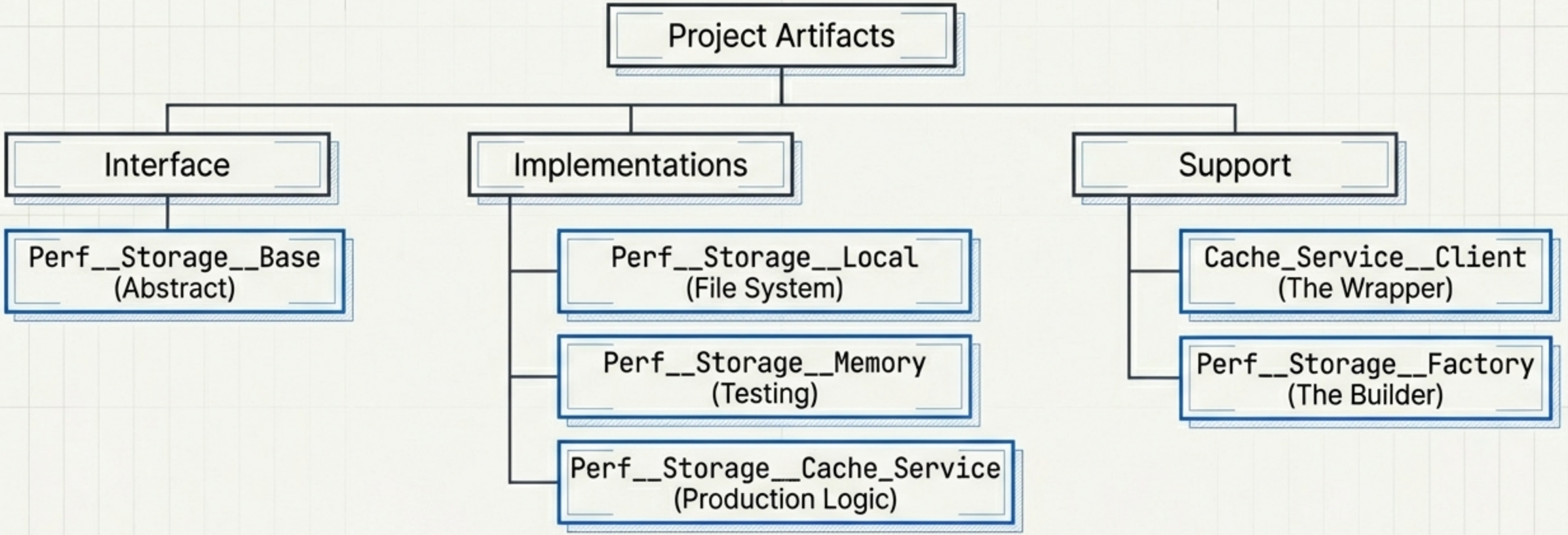
- ✅ **Hierarchical Child Data:** Fits natural artifacts structure (results/reports).
- ✅ **Official Client Wrapper:** Reduces code debt and maintenance.
- ✅ **Factory Pattern:** Clean dependency injection via config.
- ✅ **In-Memory Testing:** High confidence, low latency.

What Didn't Work

- ❌ **Direct REST Calls:** Replaced to avoid duplicating client library features.
- ❌ **Mock-Heavy Tests:** Replaced due to brittleness.
- ❌ **Stored `'cache_id'`:** Removed to ensure thread safety.
- ❌ **Separate `'setup()'` method:** Removed in favour of passing dependencies via constructor.

Delivered Artifacts & Component Breakdown

40 Total Python Files
(25 Source, 15 Test)

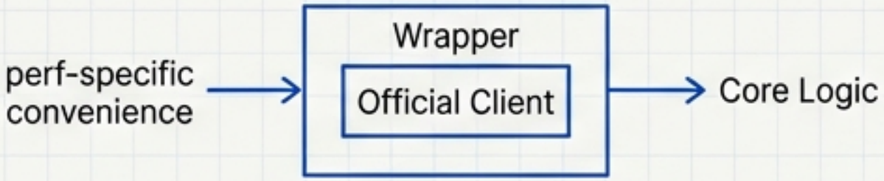


Schema Updates: 'Schema__Perf__Entry.py' (Added cache_key), 'Schema__Perf__Storage__Config.py' (Cleaned).

Technical Decision Matrix

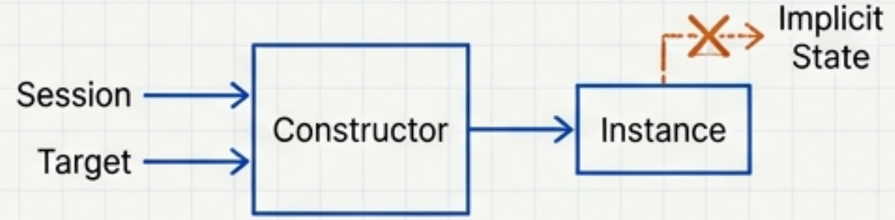
Decision: Wrapper vs. Direct Client

Rationale: Add perf-specific convenience while delegating core logic to the official implementation.



Decision: Constructor Arguments

Rationale: Pass Session/Target defaults explicitly to avoid hidden state changes.

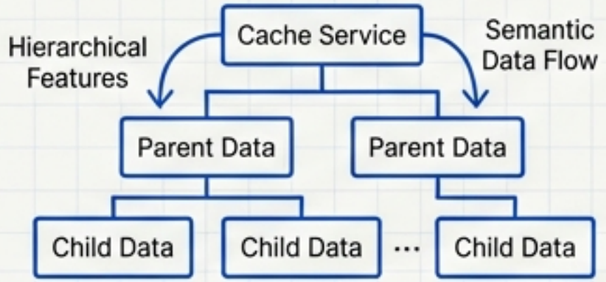


VALIDATED APPROACH

VALIDATED APPROACH

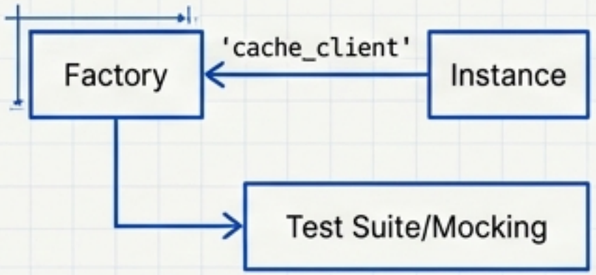
Decision: Child Data Pattern

Rationale: Leverages cache service's native hierarchical features to keep data semantic.



Decision: Dependency Injection

Rationale: Factory accepts 'cache_client' to support easier testing and mocking scenarios.



Roadmap: Phase E_4 Candidates

1. **List Keys Implementation**
   Use the cache service 'refs' API to list child data files.
2. **Batch Operations**
   Mechanism to store/retrieve multiple files efficiently.
3. **Cache Invalidation**
   Logic to delete an entry and all its children.
4. **Compression**
   Handling for large string content.
5. **Metrics**
   Tracking storage operation latencies.

Conclusion: Production-Ready Storage Abstraction

Phase E_3 has successfully bridged the gap between local analysis needs and scalable cloud storage.

Core Assets Delivered

Hash-Based Lookup

The key pattern unlocking semantic storage on the Cache Service.

Type_Safe Wrapper

Ensuring compliance and reducing runtime errors.

In-Memory Confidence

A testing strategy that mimics production without the latency.

Final Status: Ready for deployment. Extensible for Phase E_4.