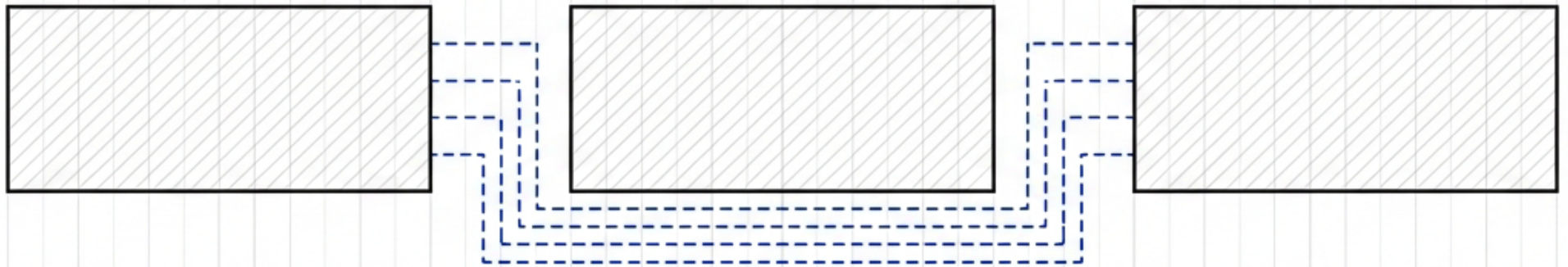


The Phased Development Methodology

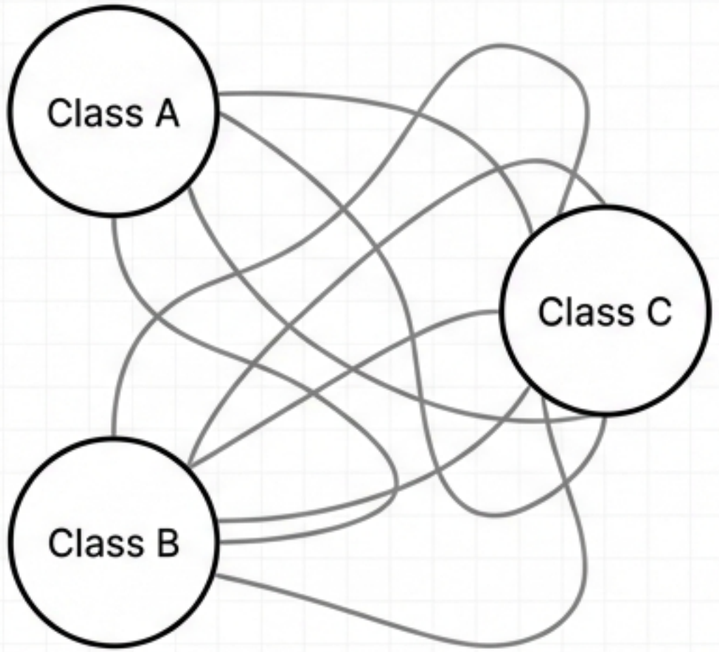
A GUIDE TO BUILDING INDEPENDENT, COMPOSABLE, AND HIGH-PERFORMANCE SYSTEMS

Methodology for breaking complex system architectures into testable, isolated units.

THE SYSTEM AS A PIPELINE

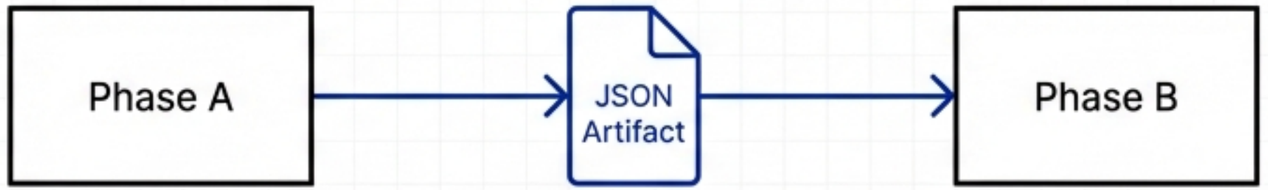
Core Principle: Independent transformations connected via serializable data.

THE OLD WAY



Implicit State & Import Chains

THE METHODOLOGY



Explicit State & Serializable Boundaries

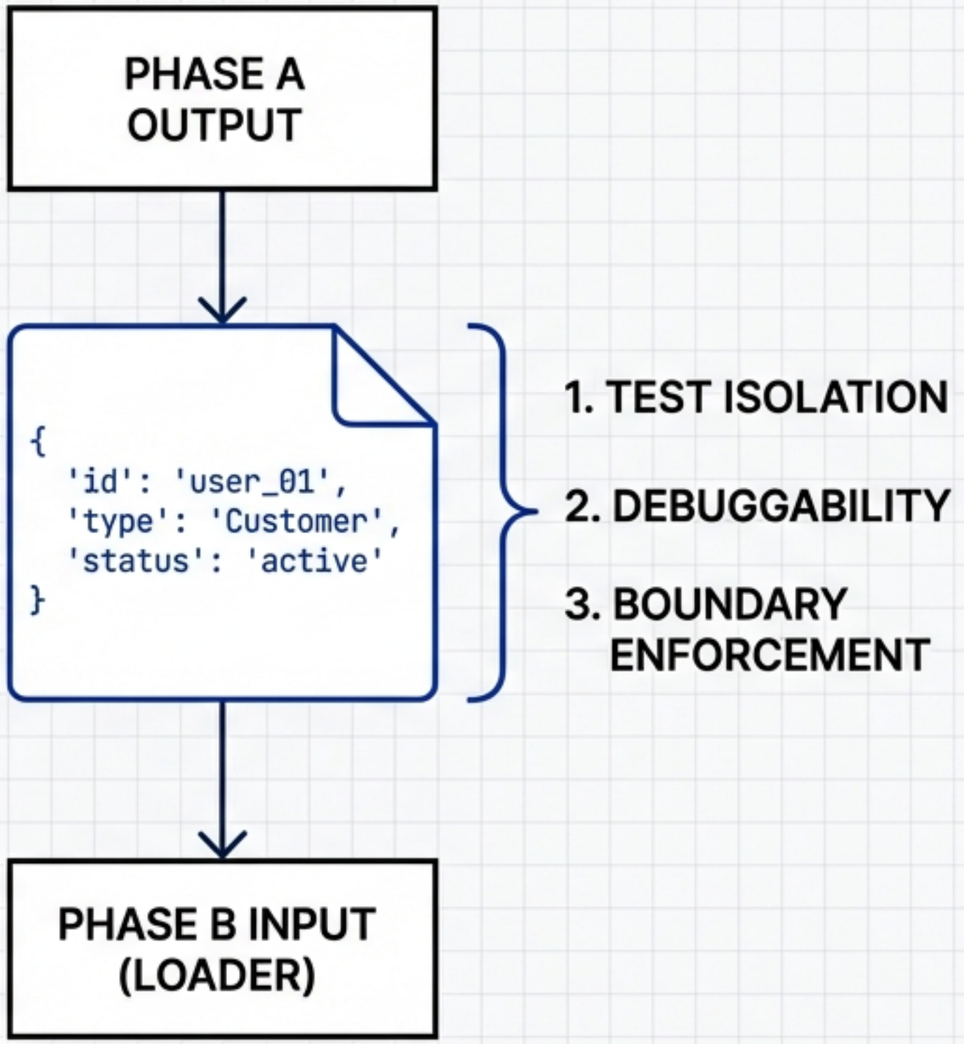
The Golden Rule:
If you can't serialise the state, you can't isolate the phase.

DECOUPLING VIA JSON FIXTURES

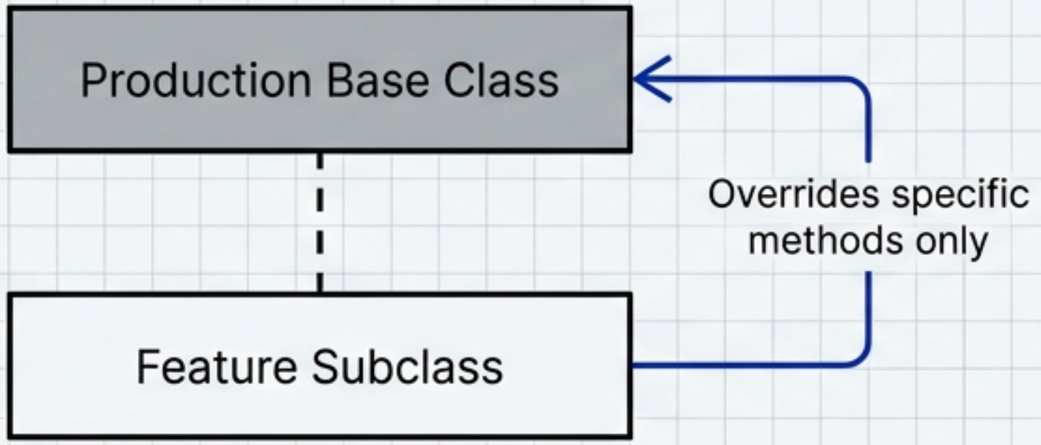
PROBLEM:
Phases traditionally depend on each other's classes, creating import chains. You cannot test Phase B without installing Phase A.

SOLUTION:
- Serialize phase outputs to JSON with full type information.
- The next phase loads the JSON, not the previous phase's classes.

IMPLEMENTATION GUIDELINES:
- **DO** serialize with full type information.
- **DO** use standard reconstruction methods.
- **DO** validate JSON format in tests.
- **DO NOT** use compressed JSON (maintain readability).

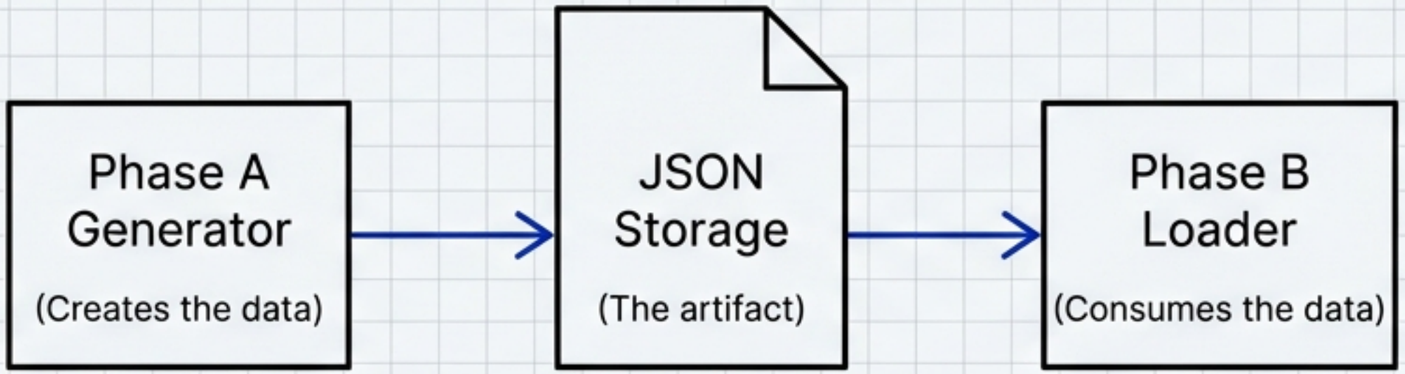


STRUCTURAL PATTERNS FOR MAINTAINABILITY



01. The Subclass Pattern

Modify behavior without touching production code. Safe rollback by deleting the subclass.

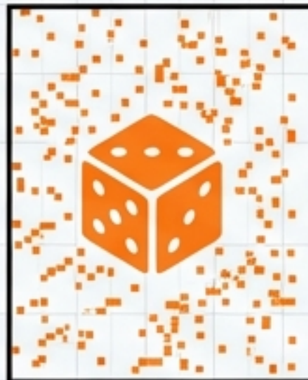


02. The Fixture Ecosystem

Generators create artifacts. Loaders consume them. Tests skip gracefully if artifacts are missing.

DETERMINISM IN TESTING

Reproducibility is King. Eliminate "Flaky" Tests.



Random UUIDs
(The Flaky Path)

ID: a7b9-4c21-...

- Cannot assert specific values.
- Debugging requires tracing dynamic variables.
- Breaks cross-phase tracking.

Deterministic IDs
(The Solid Path)

ID: user_01

- Hardcoded expected values.
- Seamless cross-phase tracking.
- Eliminates "works on my machine" randomness.

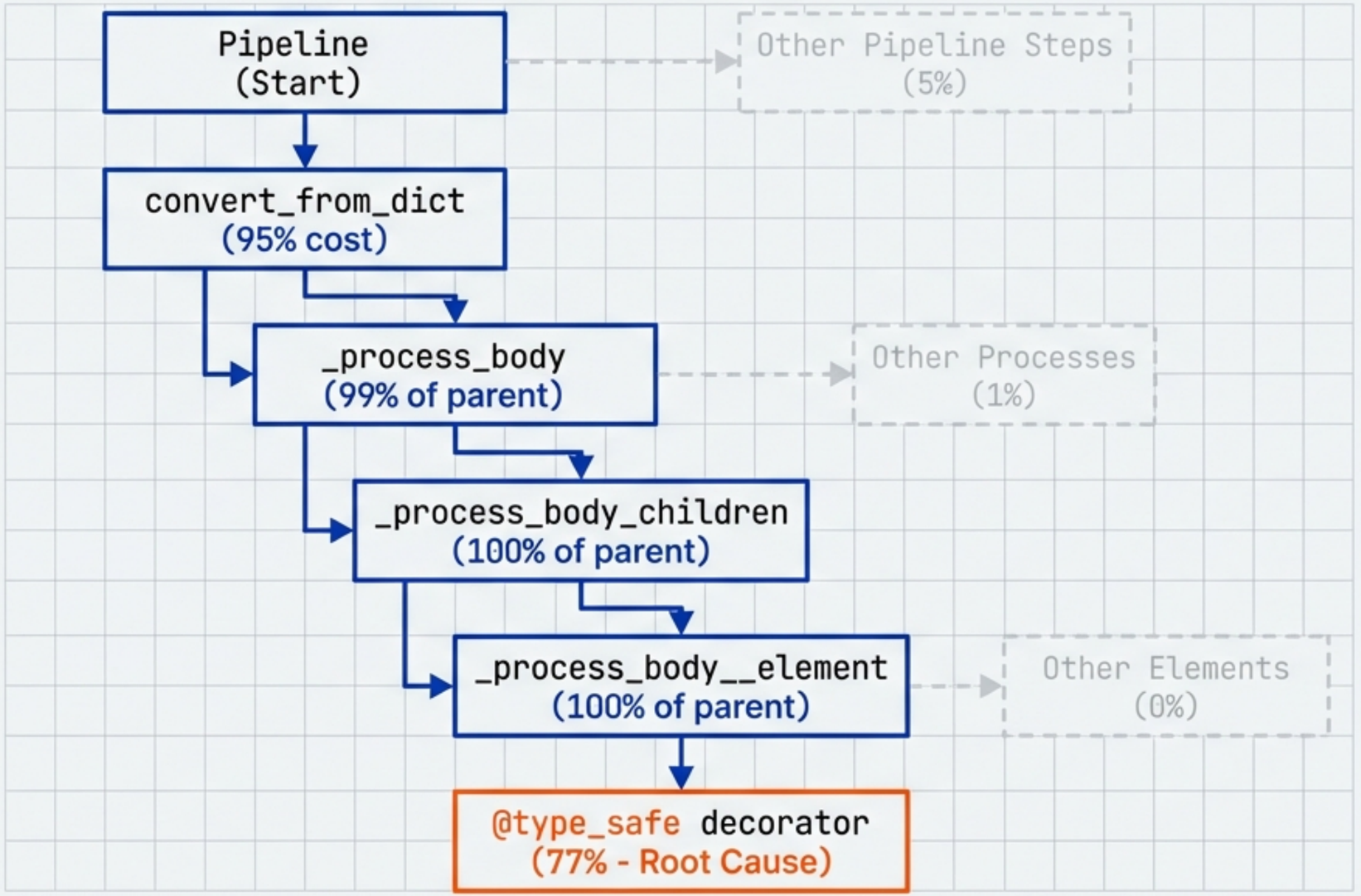
Technical Solution:

Use 'graph_deterministic_ids()' context managers in all test fixtures.

Swiss Engineering Schematic

PERFORMANCE: THE "RABBIT HOLE" METHODOLOGY

Systematic Bottleneck Isolation



Technique:

Start at the highest level.

Identify dominant cost.

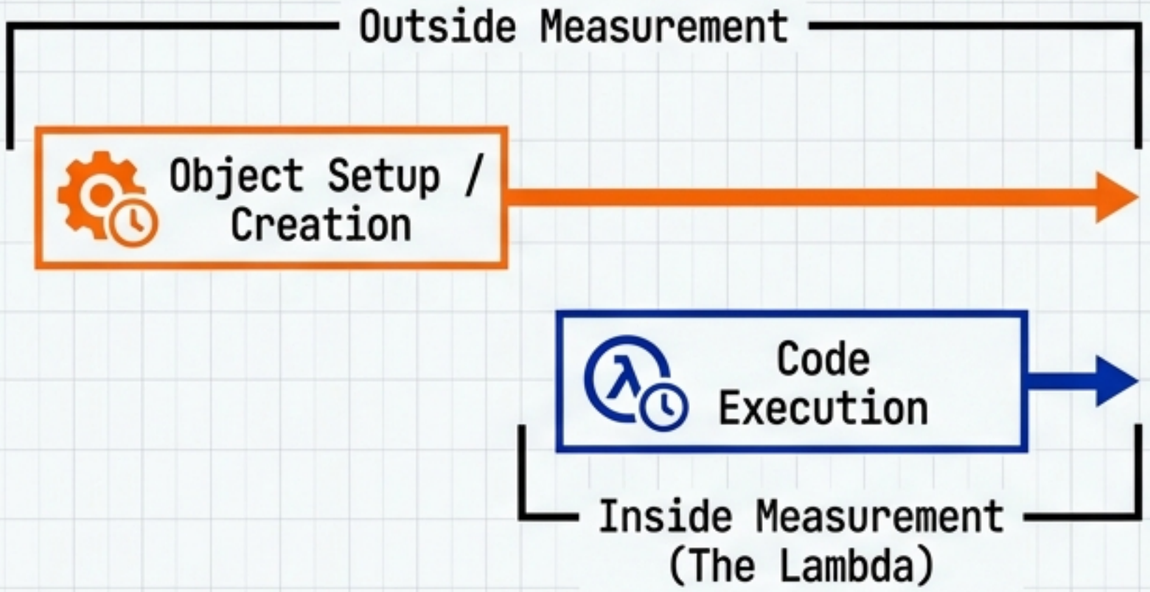
Drill down one level.

Repeat.

Use context managers like 'type_safe_fast_create' to isolate overhead.

BENCHMARKING INFRASTRUCTURE & MATH

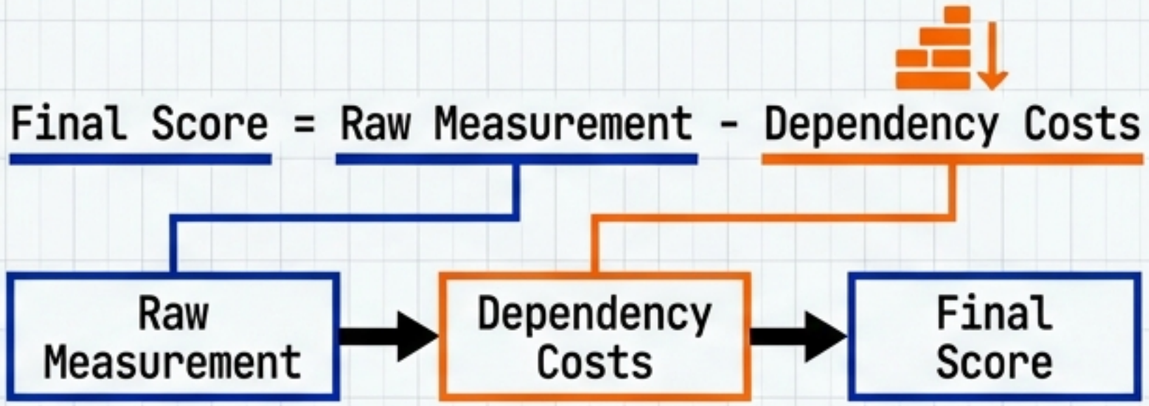
Object Reuse Strategy



Problem: Creating objects inside benchmark lambdas measures setup overhead.

Solution: Create objects ONCE before registration.

Post-Benchmark Score Adjustment



Challenge: Operations with dependencies result in cumulative costs.

Method: Measure cumulatively, then adjust scores programmatically using 'final_score' and 'raw_score' properties.

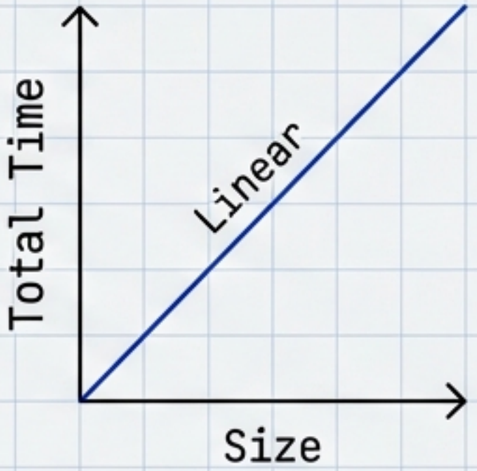
Check: Sum of adjusted parts == Full measurement.

Benchmark code should mirror production code.
Run the real flow, adjust mathematically afterwards.

VALIDATION STRATEGIES

1. Scaling Validation (Linearity)

Size	Total Time	Per-element	Scaling Status
100	3.20ms	32µs	Baseline
200	6.40ms	32µs	2x Scaling ✔
500	16.00ms	32µs	5x Scaling ✔



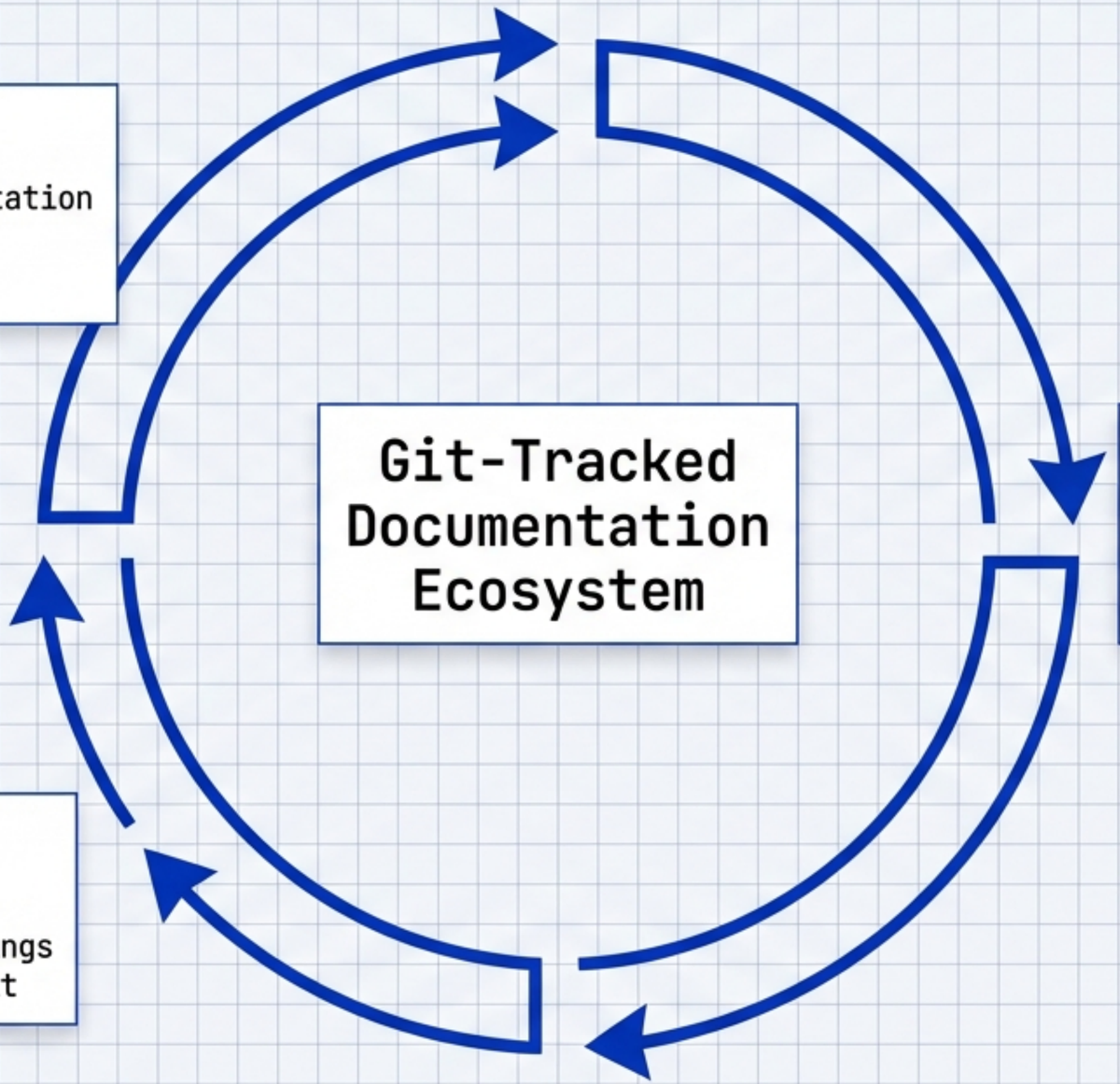
2. Elimination Testing

Purpose: Prove causality. **Method:** Remove the suspected bottleneck layer to observe performance delta.

3. Layer Bypassing

Purpose: Isolate which abstraction layer causes overhead.

The Development Lifecycle



The Brief

- Must exist before implementation
- Diagrams of components
- Data flow definitions

The Build

- Unit Tests (per file)
- Integration Tests (per phase)
- Benchmark Tests creation

The Debrief

- Capture learnings
- Document performance findings
- Regressions checked via Git

Git-Tracked Documentation Ecosystem

COMMON PITFALLS & ANTI-PATTERNS

AVOID (The Wrong Way)	ADOPT (The Right Way)
~~Importing previous phase classes~~	Loading JSON fixtures only.
~~Random UUIDs / Non-deterministic data~~	Hardcoded, deterministic IDs.
~~Modifying core classes for new features~~	Using the Subclass Pattern.
~~Compressed, unreadable JSON~~	Full formatting for readability.
~~Measuring setup inside the lambda~~	Object reuse (Setup outside).
~~Substring assertions~~	Exact value matching.

DEFINITION OF DONE: CHECKLISTS

PHASE COMPLETION

- [x] Brief and Debrief documents created.
- [x] Unit and Integration tests pass.
- [x] Fixtures generated and validated for the next phase.
- [x] No imports from previous phase classes.

PERFORMANCE INVESTIGATION

- [x] High-level benchmark identifies bottleneck area.
- [x] Systematic drilling (Rabbit Hole) reaches root cause.
- [x] Scaling validation confirms linear behavior.
- [x] Results stored and Git-trackable for regression history.

SUMMARY OF BENEFITS

ISOLATION

Develop and test phases independently without full system installs.

REPRODUCIBILITY

Deterministic fixtures ensure consistent results and eliminate flakes.

MAINTAINABILITY

Subclass pattern keeps core code clean; safe rollback capability.

DEBUGGABILITY

Human-readable JSON fixtures make state inspection trivial.

PARALLELISATION

Multiple phases can be worked on simultaneously by different teams.

ROOT CAUSE ANALYSIS

Systematic drilling identifies true bottlenecks, not guesses.

Complexity is managed through strict isolation and rigorous measurement.