PART 3 OF 3 | MANUFACTURING TEST DATA ARCHITECTURE SERIES

The Data Doesn't End at the Unit

Why System-Level Correlation Requires Decisions Made at Unit Test — and How to Make Them

Jonathan Slavic | Customer Solutions and Analytics Architect | Circuit Check Inc.

Series context: Part 1 established that development logging architecture becomes production architecture by default, and that the decoder — the Excel workbook or script built to make flat file output readable — is the artifact that proves the underlying data architecture was never designed for manufacturing. Part 2 showed that standard validation processes are systematically blind to this flaw: rigorous process, wrong scope. This paper examines what that flaw costs when the unit leaves the UUT station, and what has to be true at the data architecture level for the cost to stop compounding.

Abstract

The architectural debt documented in Parts 1 and 2 of this series — the flat file output, the execution-order logging, the decoder built to compensate for data that was never structured for analysis — does not stay at the UUT station. It follows the unit into integration test, into system-level test, and into fielded operation. At each stage, the debt compounds: a new team inherits data it cannot query, builds its own tools to compensate, and creates its own layer of architectural debt that the next stage will inherit in turn.

The scope assumption that produced the debt — the implicit decision that UUT data exists to serve the UUT program — is the assumption that has to change. The data decisions made at unit test are not UUT decisions. They determine whether cross-stage correlation is ever possible, or whether each stage operates as an isolated silo connected to the others only by a serial number and a pass/fail record.

This paper describes what cross-lifecycle data architecture actually requires, why it fails under the conditions most programs operate in, and what the concrete fixes look like — from the data decisions that cost almost nothing at program start to the platform infrastructure that makes those decisions durable across the organizational boundaries that no homegrown tool was built to cross.

> ***The data architecture decisions made at unit test are not UUT decisions. They are lifecycle decisions wearing UUT clothes.***

1. How the Debt Follows the Unit

Part 1 of this series described how development logging architecture becomes production architecture by default: the flat file that was appropriate for a development engineer debugging a handful of prototypes becomes the data foundation for a factory running hundreds of units per day, because no one made an explicit decision to change it. The decoder — the workbook built to make that output readable — travels with the code and becomes the analysis infrastructure for a manufacturing environment it was never designed to serve.

That debt does not stay at the UUT station. It follows the unit, and it compounds at every stage boundary the unit crosses.

1.1 The Integration Tooling Gap

The tooling gap between engineering/integration and production reflects a genuine difference in purpose. Production test tools are built to run: execute fast, log deterministically, minimize latency. They are not designed for human interruption. Engineering and integration tools are built for the opposite — start, stop, breakpoints, variable inspection, iterative probing. That flexibility is what makes them useful for troubleshooting. It is also what makes their output structurally incompatible with production data.

In practice, integration runs on Python scripts, MATLAB routines, pytest frameworks, custom REST clients, shell scripts written to capture a specific bus transaction on a specific afternoon. Text-based, immediate, and aimed at the question in front of the engineer who wrote the script — not at producing data that correlates with anything downstream.

The cultural dimension is worth naming directly. Integration engineers are text programmers. Graphical environments like LabVIEW and production test executives like TestStand are not tools they reach for voluntarily. Expecting them to instrument bench work in production frameworks is a requirement that will be declined, worked around, or ignored.

Whatever code reaches the factory floor becomes manufacturing engineering's responsibility to support, sustain, and maintain. Engineering scripts written for a bench investigation were not built with that lifecycle in mind — no documentation for the technician who inherits them, no error handling for production edge cases, no version discipline for hardware revisions, no consideration for the engineer debugging a failure message at the station without the original author available. When integration code crosses into production, manufacturing engineering inherits a support obligation it had no input into designing.

Underneath all of this is a divide that exists on almost every program and is almost never talked about directly: engineering and manufacturing operate as separate cultures with separate priorities, separate vocabularies, and a handoff between them that transfers deliverables but rarely transfers understanding. Engineering is measured on design performance and schedule. Manufacturing is measured on yield, throughput, and cost. Engineering moves to the next program. Manufacturing lives with the last one. The wall between them is not a process gap. It is a cultural gap sustained by organizational structure, reinforced by program timelines, and most visible at exactly the moment a unit fails in production and nobody on the floor has the context to explain why.

Breaking that wall down requires more than a better handoff document. It requires production and manufacturing engineering to be present during development — early enough to shape the data standard before formats are locked, early enough to influence the test architecture before it is handed off, and early enough to absorb the tribal knowledge of how the system actually behaves before the engineers who built it move on. Not as reviewers at the end. As participants from the beginning. Engineering learns what manufacturing needs to sustain the product. Manufacturing learns why the system was designed the way it was. The knowledge that normally vanishes at handoff gets transferred while both groups are still working the same problem, and the wall that sustained the divide starts to come down before the first unit reaches the floor.

> ***The wall between engineering and manufacturing is not a process problem with a process solution. It is a cultural divide that only closes when both sides are in the room early enough to build something together.***

1.2 The Bench Pass Loop

When a unit fails at the test station, manufacturing engineering does not immediately escalate. They work the problem with the tools and knowledge available: verify the assembly, check for obvious defects, swap known-good units to isolate the failure, rule out fixture issues, confirm the test software is behaving as expected. Real troubleshooting, done systematically, within the boundaries of what production data and production tooling can illuminate.

The limit of that troubleshooting is not a skill limit. It is a data limit. When the failure mode lives in a firmware edge case, a timing dependency that only surfaces under production load conditions, or an interaction between subsystems that integration test would have exercised but manufacturing test was never designed to reach, production data cannot isolate it. Manufacturing engineering has ruled out everything they can rule out. Engineering support gets called in.

The unit goes to the bench. It runs clean. It comes back to production and fails again. This loop is not a result of insufficient effort on either side — it is the direct consequence of two environments that cannot compare notes. The bench environment and the production station are running different tooling, producing data in incompatible formats, with no structured basis for comparison. Engineering confirms the unit is not catastrophically broken. Manufacturing confirms the station is not at fault. Neither can see what the other saw, so neither can identify where the two observations diverge — which is exactly where the root cause lives.

> ***It passed on the bench. It fails in production. Without correlatable data from both environments, that statement is not the beginning of a diagnosis. It is the entire investigation.***

The loop persists because the wall persists. Engineering and manufacturing are looking at the same unit through tools that cannot compare notes, shaped by a divide that began long before the unit failed. Closing the loop requires closing the divide — which means manufacturing engineering at the table during development, building shared context before the handoff, so that when a unit fails at the station the two sides are not starting from opposite ends of an organizational boundary. They are starting from the same understanding of how the system was built, what it was designed to do, and where the edge cases live.

1.3 System Test and Field: Where the Cost Peaks

At system test, failure modes are emergent — arising from interactions between components that were individually conformant at the unit level. Diagnosing them requires tracing back through integration to unit measurements that exist, if they exist at all, in a format designed for execution monitoring at the UUT station. Root cause analysis becomes manual archaeology across stage boundaries that each have their own data format, their own tooling, and their own definition of what a test result is.

In the field, the cost is highest and the latency longest. By the time a field return reaches engineering, the unit may have been in service for months or years. The question — was this failure mode present at unit test, detectable but below the action threshold, or did it develop post-production? — is exactly what a complete unit test record would answer. Without it, analysis is failure mode isolation at the time of return, with no longitudinal context and no path from the field observation back to the manufacturing data that could have predicted it.

> ***The decoder proves the data architecture was wrong at the UUT station. The bench pass loop proves the gap has reached integration. The three-week investigation at system test proves the debt has compounded across every boundary the unit crossed since.***

2. What Cross-Stage Correlation Actually Requires

Cross-stage correlation — the ability to connect a behavior observed at system test or in the field back to a measurement made at unit test — is not technically exotic. It requires three things, simultaneously and consistently across all stages.

A Persistent Unit Identity

The serial number is the primary key of every cross-stage correlation the program will ever attempt. It has to be captured at the UUT station, normalized to a consistent format, and carried forward as the same field in every subsequent test record. A unit that enters UUT test as “SN-00142” and enters system test as “142” and returns from the field as “0142-REV2” is the same unit. A data architecture that cannot resolve that equivalence cannot support cross-stage correlation, regardless of how well-structured each stage’s individual data is.

This decision costs almost nothing at program start. It costs a data reconciliation project at production scale. The difference between those two costs is the entire argument for making the decision early.

Structured Measurement Records at Every Stage

Pass/fail is a verdict. It is not a record. Cross-stage correlation requires the measurement value, the test limit, the test condition, the station identifier, and the timestamp — at every stage, in a form that can be queried by any of those fields. A flat file that captured all of those values at execution time but organized them in execution order rather than by logical test structure is not a queryable record. It is a log that requires a decoder, and a decoder that works for one file does not scale to the population-level analysis that cross-stage correlation requires.

This is the direct continuation of the argument in Part 1. The data structure decision made at the UUT station — execution-order logging versus structured measurement records — determines whether the data from that stage is ever usable in a cross-stage query. Flat file output with a decoder is not a workaround. It is architectural debt with a compounding interest rate measured in investigation hours.

A Common Schema Across Stage Boundaries

The measurement records from each test stage have to exist in a common data model: a consistent definition of what a test result is, what a measurement is, what a test limit is, and what a station is, applied uniformly across all stages and all organizational boundaries. When each stage defines its own schema independently, a query that begins at system failure and needs to traverse two stage boundaries to reach unit test data becomes a data integration project. When a common schema is defined once at program start and applied at every stage, the same query is a database operation.

The common schema is the structural fix that prevents stage boundaries from becoming silo boundaries. It does not require every stage to use the same test tooling. It requires every stage’s tooling to produce output that maps to the same data model. That is a requirement that can be defined before any stage begins building, at the same program start decision point where the unit identifier is defined.

> ***Pass/fail is a verdict. Correlation requires measurements. The decision to capture one without the other is a data architecture decision, whether or not it is recognized as one.***

3. The Organizational Boundary Problem

The three requirements described in Section 2 are not technically difficult to satisfy. What makes them difficult is organizational: the teams that own each test stage are different teams, with different tooling, different data owners, different program priorities, and no shared infrastructure. The decision to define a cross-stage unit identifier requires someone with authority over all stages. The decision to define a common schema requires someone who understands what each stage's data needs to contain for the stages downstream of it to answer their questions. Neither of those people typically exists in the program structure at the moment the decisions need to be made.

The result is the same pattern that produces architectural debt at the UUT station, repeated at every stage boundary. Each team makes the data decisions that serve its program. The cost of those decisions is paid downstream, by teams that had no input into them, under conditions that are worse than the conditions under which the original decisions were made.

The Incentive Structure That Sustains the Pattern

The UUT test engineer who builds flat file output will not personally experience the system test investigation that results from it, unless they are the engineer who later needs to trace a system failure back to a unit measurement. In most programs, those are not the same person. The cost is externalized. The scope assumption is never challenged because the person who set the scope is not the person who pays for it.

This is not negligence. It is a predictable consequence of organizational boundaries that do not carry accountability for cross-stage data continuity. Every program compresses, and when compression comes, the investment that does not block the next milestone is the first investment cut. Cross-stage data architecture does not block any milestone. Its absence does not manifest as a failure until a stage boundary has already been crossed and the cost of going back is larger than the cost of the original investment would have been.

The Minimum Governance Structure That Changes the Pattern

The instinct is to reach for a documented requirement — a signed artifact that makes the data architecture decision visible and forces someone to acknowledge the downstream cost before the cut is made. The instinct is correct in principle and frequently ineffective in practice. Programs under launch pressure do not stall because a document needs to be modified. Someone with authority signs to accept the risk, the program moves forward, and the signature transfers accountability to a person who will not be present when the cost surfaces. The document exists. The cut happens anyway.

What actually changes the pattern is accountability that stays attached to the decision across time — ownership that does not end at the signature or the launch or the handoff to manufacturing. The person who accepted the risk of cutting cross-stage data infrastructure should be the person who is called when the system test investigation cannot find its root cause eighteen months later. In most programs, that connection does not exist. The decision and its consequence are separated by an organizational boundary and a program transition, and the cost lands on whoever is holding the product when it fails.

The governance fix, realistically, is not a form or a process. It is building the cross-stage data requirements into the program's definition of done — not as a risk to be accepted and signed away, but as a completion criterion that the program has not met until it is satisfied. That is a harder standard to apply under schedule pressure. It is also the only standard that survives it.

> ***A signature that accepts the risk is not governance. It is documentation that the risk was understood by someone who will not be around when it compounds.***

4. The Fixes, In the Order They Have to Happen

The fixes for cross-stage architectural debt are not independent. They have a sequence, and the sequence matters: later fixes depend on earlier ones, and earlier fixes that are skipped cannot be fully compensated for by later fixes. The order reflects the compounding structure of the debt: the cheapest fix is the earliest fix, and the cost of each subsequent fix is higher than the one before it.

Fix 1: Define the Data Architecture Before the First Test Is Built

Three decisions, made once at program start, determine whether cross-stage correlation is ever possible. They cost almost nothing at this stage and cannot be recovered at equivalent cost later.

- **Normalize the unit identifier.** Define the serial number format that all stages will use. Make it the primary key of every test record the program produces. Do this before the first test station is designed.
- **Require structured measurement output.** Define what a test result record contains: measurement value, test limit, test condition, station identifier, timestamp, unit identifier. Not a summary. Not a pass/fail outcome. A structured record that can be queried by any of its fields. This is the decision that means a decoder is never needed, because it never creates the condition a decoder was built to compensate for.
- **Define the cross-stage schema.** Specify the common data model that all stages will conform to. The fields vary by stage. The core model does not. This is the decision that means a cross-stage query is a query, not a project.

These three decisions, written into the program's test architecture document before any stage begins building, are the structural fix. Everything downstream of this decision is either building on a foundation that supports cross-stage correlation or compensating for a foundation that does not.

Fix 2: Make the Cross-Stage Requirement Visible at Every Compression Point

Every program compresses. When compression comes, the cross-stage data architecture requirement will be offered as a cut, because its absence does not block the next milestone. The governance fix described in Section 3 — a documented, signed data continuity requirement — is the mechanism that makes the cost of that cut visible at the moment it is proposed, rather than two years later when the system test investigation cannot find its root cause.

The requirement does not prevent the cut from being made. It ensures the cut is made explicitly, with the downstream cost on the table, rather than implicitly, as a byproduct of schedule pressure that no one named as a data architecture decision.

Fix 3: Retrofit Existing Programs Without Stopping the Line

For programs already in production and already carrying architectural debt, the fix is not to stop the line and rebuild the data architecture from scratch. It is to intercept the data that exists, normalize it into a structured form, and route it into a cross-stage data store that supports the queries the program needs to answer — without touching the test executive or requiring the production program to absorb a redesign.

The decoder, in this context, is not removed. It is replaced with a normalization layer that produces structured records from the flat file output the test program was already generating, and routes those records into a data store alongside the structured output from newer stages. The production line does not stop. The architectural debt stops compounding. The program begins building the cross-stage data history that future investigations will need, even though the past data remains in its original form.

This is not a complete solution. It is the minimum viable fix for a brownfield program that cannot afford a clean-slate rebuild. Its value is that it stops the meter from running further while the program builds toward a more complete architecture on its own schedule.

> ***The brownfield fix does not repay the debt. It stops the meter. The program that stops the meter today is in a better position for every investigation it will need to run for the rest of its production life.***

5. Where the Platform Earns Its Value

The fixes described in Section 4 address decisions and governance. They define what the data architecture needs to be and who is accountable for it. What they do not address is the technical infrastructure that makes those decisions durable across the organizational boundaries, tool changes, and program transitions that every production program eventually experiences.

The homegrown tool — the decoder's successor, the stage-local analytics workbook, the custom Python pipeline built to pull data from three separate systems and reconcile it into a quarterly yield report — is the default infrastructure that most programs rely on for cross-stage data management. It is built to the scope assumption: it serves the stage that built it, with the schema that stage needed, at the scale that stage required when the tool was written. It reaches its architectural ceiling at the boundary where cross-stage correlation becomes necessary, which is exactly the boundary where the cost of ceiling is highest.

What Homegrown Tools Cannot Sustain

The practical ceiling of a homegrown cross-stage data tool is visible in three failure modes that every program running one will eventually encounter:

- **Scale failure.** The tool that reconciles data from three stages for a quarterly report was built for quarterly reports. When the program needs to run the same query on demand, at production velocity, across a population of tens of thousands of units, the tool's architecture — designed for a human analyst running it periodically — cannot sustain the load. The program either accepts slow answers or assigns an engineer to maintain a tool whose complexity has grown beyond its original scope.
- **Boundary failure.** When a new test stage is added, or an existing stage changes its tooling, or a field service system is brought into the data architecture, the homegrown tool requires a new integration. Each integration is a custom development effort. Each custom development effort is an opportunity to introduce inconsistencies in the cross-stage schema. Over time, the tool that was built to enforce data continuity becomes the primary source of data inconsistency.
- **Knowledge failure.** The homegrown tool was built by an engineer who understood the data architecture deeply enough to build it. When that engineer moves to a different program, the tool becomes unmaintainable in any direction that requires understanding why it was built the way it was. The organization that built the tool to avoid vendor dependency has created a deeper dependency: on a single engineer whose knowledge is not documented and whose departure is an unplanned program risk.

What a Purpose-Built Platform Provides

A platform designed for cross-lifecycle manufacturing data provides the five capabilities that homegrown tools cannot sustain at production scale: ingestion from existing test environments without requiring those environments to be rebuilt; enforcement of the cross-stage unit identifier and common schema across all ingested data; a data model that treats all stages as a single queryable data set rather than separate systems that can be joined by a sufficiently motivated analyst; cross-stage analytics as a built-in function rather than a custom development project; and an integration layer built to handle the brownfield complexity — duplicate records, format inconsistencies, gaps in historical data — that every production program accumulates.

This is where the platform earns its value beyond what any homegrown tool can reach: not in the individual stage query, which a well-built stage-local tool can answer, but in the cross-stage query that begins at a system failure and needs to traverse two or three stage boundaries to reach the unit test measurement that could explain it. At that point, the difference between a platform built for cross-stage correlation and four separate

data environments connected by manual effort is the difference between a query and a three-week investigation.

The platform does not replace the upstream fixes. A platform ingesting flat file output through a normalization layer is a better position than four separate flat file directories, but it is not the same position as a program that defined structured measurement output at program start. The platform makes the upstream decisions durable. Without the upstream decisions, the platform is compensating for an architecture that was never designed to support what it is being asked to do. Without the platform, the upstream decisions degrade at every organizational boundary they cross, because organizational boundaries do not carry data architecture requirements automatically.

> ***The platform earns its value at exactly the point where homegrown tools reach their ceiling: the cross-stage query that no single stage's tools were built to answer, on a population of units that no single stage's data was built to contain.***

Closing

The argument in this series has followed a single structural problem through three escalating scopes. Part 1 showed where the problem is created: development logging architecture becomes production architecture by default, and the decoder — the tool built to compensate for data that was never designed for analysis — is the artifact that proves it. Part 2 showed why it persists: validation is thorough and well-intentioned, but its scope stops at execution quality. It never asks whether the data the test system produces is architecturally fit for what manufacturing will need from it.

This paper has shown what the problem costs when it is not caught: architectural debt that compounds at every stage boundary the unit crosses, multiplying the investigation cost of every failure that requires cross-stage context to diagnose, and foreclosing the population-level analysis that could convert those investigations from reactive root cause exercises into proactive yield intelligence.

The fixes are available at every stage of a program's life. They are most valuable, and least expensive, at program start — before the first test station is built, when the unit identifier and the measurement record structure and the cross-stage schema cost an afternoon to define and a career's worth of investigation time to defer. They are still meaningful in production, where the normalization layer and the platform integration stop the meter from running further even when the debt from earlier decisions cannot be fully repaid.

What they require, in every case, is recognizing that the data architecture decision is being made — whether or not it is named as one. The decoder was a data architecture decision. The flat file output that required the decoder was a data architecture decision. The stage boundary that became a silo boundary because no one defined a cross-stage schema was a data architecture decision. Every one of those decisions was made by someone who was thinking about something else at the time.

The fix is thinking about the data architecture when the decision is made, not when the cost of the decision surfaces. That moment is almost always earlier than it feels, and almost always less expensive than what comes after it.

About the Author

Jonathan Slavic is a Customer Solutions and Analytics Architect at Circuit Check Inc. with over 25 years of experience in hardware test engineering, manufacturing systems, and production data intelligence. He has served as a Principal Test Engineer at SRCTec LLC, a Department of Defense contractor, and at MSA Safety Inc., an OEM — giving him direct experience across the CM, OEM, and government-contracted environments examined

in this series. He serves as an Adjunct Professor in the Electronic Technology program at SUNY Onondaga Community College.

In his role at Circuit Check Inc., Jon leads pre- and post-sales integration of production data systems, including WATS and TestPartner, helping manufacturers eliminate architectural debt and transform legacy test environments into scalable manufacturing intelligence platforms. Circuit Check Inc. is the leading fixture and test system manufacturer in the United States.

This is Part 3 of a three-part series on test data architecture and manufacturing intelligence.
For inquiries about test data architecture, lifecycle data strategy, or WATS integration, contact Circuit Check Inc.