From Development Logging to Manufacturing Intelligence

The Architecture That Serves Execution Does Not Serve Investigation

Jonathan Slavic | Circuit Check Inc.

Abstract

Every test system produces data. The question is whether that data is organized to support the decisions manufacturing will eventually need to make.

Flat files are the most common output format in hardware test environments, and they are not inherently flawed. A flat file is a chronological record — a timestamped log of test execution in the order it occurred. For a development engineer debugging a single unit, that sequential record is exactly what is needed. It captures what happened. It preserves the execution flow. It is readable, portable, and easy to generate.

The limitation appears when manufacturing needs to ask a different kind of question. Chronological order answers: what happened, and when? Manufacturing intelligence requires the ability to answer: which channel failed, across which units, after which process change, at which station? Those are relational questions. Flat files do not answer them without reconstruction. Organized, structured data answers them directly.

This paper examines what flat files actually are, why they are built the way they are, and what changes when test data is organized by logical structure rather than execution order. The argument is not that flat files are wrong. It is that the architecture that serves execution does not serve investigation — and building manufacturing intelligence requires understanding that distinction before the first test runs.

1. What a Flat File Actually Is

A flat file is not a poor design choice. It is a direct output of how test execution works.

When a test executive runs, it sequences through steps. Each step executes a measurement, evaluates a limit, records a result, and moves to the next. In multichannel systems, a loop handles the iteration — channel one runs its sequence, channel two runs its sequence, and so on. Under stress or burn-in conditions, that loop repeats. Hundreds of times, if the test demands it.

At each step, data is written to disk. Measurement executes. Result appends to the file. Timestamp added. Loop continues. The result is a flat, sequential record — rows of data in the order they were produced, organized by when they happened rather than what they mean.

This is chronological data. It is accurate. It is complete. Every measurement that was taken is present, in the sequence it occurred. For a development engineer debugging a prototype, it is exactly what is needed — a faithful record of what the test did, in the order it did it, that can be opened in a spreadsheet and scanned from top to bottom.

> *A flat file tells you what happened, in the order it happened. That is its purpose — and in development, that purpose is sufficient.*

What a flat file does not tell you is how the results relate to each other. It does not group channel two's measurements together. It does not associate a test step with the test group it belongs to. It does not link a repeat iteration to the first-pass result it followed. It does not connect a failure to the environmental conditions, firmware revision, or operator shift present when it occurred.

Those relationships exist in the system being tested. They do not exist in the file that recorded it.

2. Why Grouping Builds Intelligence

The shift from chronological data to structured data is not about format. It is about organization — and organization is what determines whether data can answer questions or only report events.

Consider what a test system actually knows at the moment of measurement. It knows which unit is under test. It knows which channel produced the result. It knows which test group the step belongs to, which step within that group, what the expected limits were, and whether the result passed or failed. It knows the firmware revision loaded, the operator logged in, the station running the sequence, and the environmental conditions present. It knows whether this is the first pass or the forty-seventh repeat of a burn-in cycle.

A flat file captures the measurement. It discards the context — or buries it in column headers and timestamp sequences that require reconstruction to interpret.

> **The data is not missing. The organization is.**

Structured data collection preserves that context at the moment of capture. Instead of appending a row to a file, the test executive assigns each measurement to its place in a logical hierarchy: UUT, then channel, then test group, then test step, then result. Metadata — revision, station, operator, environment — is attached to the unit record, not scattered across row headers.

That hierarchy is what manufacturing intelligence runs on. When results are grouped by channel, you can identify which channel drives failures. When test steps are associated with test groups, you can Pareto failure modes without writing a parser. When repeat iterations are indexed rather than timestamped, you can track degradation across a burn-in cycle without reconstructing execution order from a log.

The measurements are identical in both cases. The difference is entirely in how they are organized — and that difference determines what questions the data can answer when production demands them.

> *Chronological data records what happened. Structured data builds the intelligence to understand why — and to find it across a population, not just in a single file.*

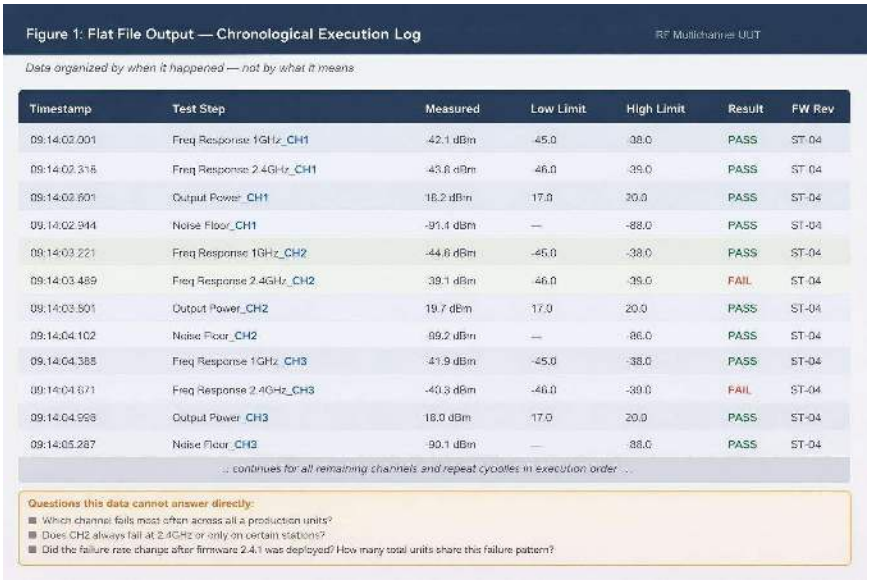

Figure 1: Flat File Output — Chronological Execution Log

RF Multichannel UUT

Data organized by when it happened — not by what it means

Timestamp	Test Step	Measured	Low Limit	High Limit	Result	FW Rev
09:14:02.001	Freq Response 1GHz_CH1	-42.1 dBm	-45.0	-38.0	PASS	ST-04
09:14:02.318	Freq Response 2.4GHz_CH1	-43.8 dBm	-46.0	-39.0	PASS	ST-04
09:14:02.601	Output Power_CH1	18.2 dBm	17.0	20.0	PASS	ST-04
09:14:02.944	Noise Floor_CH1	-91.4 dBm	—	-88.0	PASS	ST-04
09:14:03.221	Freq Response 1GHz_CH2	-44.6 dBm	-45.0	-38.0	PASS	ST-04
09:14:03.489	Freq Response 2.4GHz_CH2	-39.1 dBm	-46.0	-39.0	FAIL	ST-04
09:14:03.501	Output Power_CH2	19.7 dBm	17.0	20.0	PASS	ST-04
09:14:04.102	Noise Floor_CH2	-89.2 dBm	—	-86.0	PASS	ST-04
09:14:04.388	Freq Response 1GHz_CH3	-41.9 dBm	-45.0	-38.0	PASS	ST-04
09:14:04.671	Freq Response 2.4GHz_CH3	-40.3 dBm	-46.0	-39.0	FAIL	ST-04
09:14:04.958	Output Power_CH3	18.0 dBm	17.0	20.0	PASS	ST-04
09:14:05.287	Noise Floor_CH3	-90.1 dBm	—	-88.0	PASS	ST-04

... continues for all remaining channels and repeat cycles in execution order ...

Questions this data cannot answer directly:

- Which channel fails most often across all a production units?
- Does CH2 always fail at 2.4GHz or only on certain stations?
- Did the failure rate change after firmware 2.4.1 was deployed? How many total units share this failure pattern?

Figure 1: Flat file output for a 3-channel RF UUT. The same test step runs for each channel in execution order. The two failures — freqResponse_2400mhz on CH2 and CH3 — are present in the data but invisible as a pattern without manual reconstruction across every row.

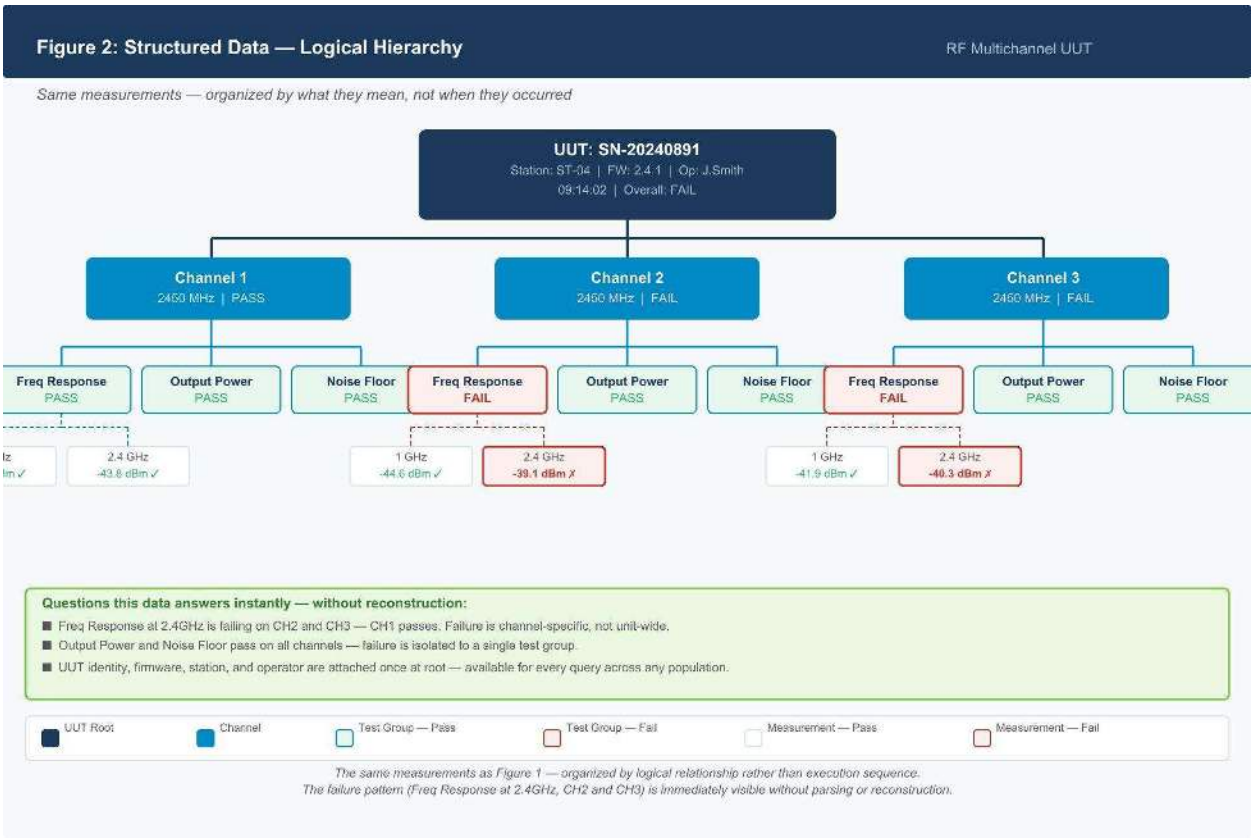

Figure 2: The same measurements organized by logical hierarchy. Channel, test group, and measurement relationships are preserved at the moment of capture. The failure pattern — freqResponse_2400mhz, CH2 and CH3 only — is immediately visible without parsing or reconstruction.

3. The Decoder: A Workaround That Becomes Evidence

At some point during development, the engineer living with flat file output builds a tool to make it readable. Typically an Excel workbook — sometimes a Python script — designed to load a data file from a failing UUT and highlight problems, color-code anomalies, or surface a root cause without manually scanning rows.

The decoder works well in development. One file. One UUT. One engineer who built the tool and knows exactly what every column means. In that context, it feels like a solved problem. The data is readable. Analysis is possible. The workflow makes sense.

> **What the decoder represents:** *An admission, written in Excel, that the raw output is not sufficient for analysis on its own. The engineers already knew the flat file was insufficient — they solved it locally, for themselves, in a tool that was never intended to scale.*

The decoder becomes a liability not when it is used in development, but when it travels with the code. It gets handed to the CM as the analysis tool. It gets referenced during formal validation as evidence that the data is analyzable. It delays — sometimes by months — the recognition that the underlying architecture cannot support production-scale intelligence. Because it works for one file, nobody asks whether it works for ten thousand.

> **The decoder test:** *If your analysis tool can only load one file at a time, it was built for debugging — not for manufacturing intelligence. The fact that it exists is proof the data architecture needs one.*

4. The Core Issue: Execution-Driven vs. Structure-Driven Data

The fundamental problem with flat file logging in multichannel systems is not that it produces large files, or that it requires parsing, or that it is an older format. The problem is architectural.

Flat files generated during loop execution represent chronology. They capture what happened in the order it happened. That is exactly what a development engineer needs for debugging — a sequential record of execution that can be scanned from top to bottom to find where something went wrong.

Manufacturing does not ask chronological questions. Manufacturing asks relational questions:

- Which channel is driving failures across this week's production?
- Did yield drop after the firmware revision pushed on Tuesday?
- Is this test station producing systematic measurement drift?
- Are failures concentrated at a specific operator shift?
- Does this failure mode correlate with a process change upstream?

These questions cannot be answered by scanning a file from top to bottom. They require the ability to group results by channel, correlate across time, compare across stations, and filter by revision — simultaneously, across a population of units, without manual reconstruction.

> *Execution order is useful for debugging. Logical structure is essential for insight. These are not different implementations of the same goal.*

5. What Structured Data Inside the Test Executive Looks Like

The alternative to real-time flat file export is not a more complex file format. It is a different approach to when and how data is organized.

Instead of writing measurements to disk as they are generated, a structured approach collects data inside the test executive during execution. Measurements are associated with their logical context as they are recorded — not reconstructed afterward. At test completion, a fully organized data object is released, representing the unit not as a sequence of events but as a hierarchy of relationships.

A manufacturing-ready data model organizes results by logical structure:

- **UUT identity —** serial number, part number, revision
- **Test context —** station ID, operator, firmware version, environmental conditions, timestamp
- **Channel grouping —** each channel's results nested under its identity, not interleaved by execution order
- **Test hierarchy —** test groups, test steps, measurements, limits, and pass/fail status preserved as relationships
- **Repeat index —** stress cycles and repeated measurements associated with their iteration, not just their timestamp

This structure does not make the test run differently. It does not add measurement overhead. It requires that the data model be defined before execution begins — which is exactly the design decision that development timelines consistently defer.

The difference between these two approaches is invisible in the lab. Both produce output. Both allow a single engineer to analyze a single file. The difference becomes visible at scale, under production pressure, when the questions being asked are no longer about one unit — they are about a population.

6. Architectural Debt and Why It Compounds

Every decision made about data structure during development becomes part of the foundation that production builds on. When that foundation is chronological rather than relational, the programs and processes downstream adapt to work around it. Custom parsers get written. Analysis workflows get built on top of the flat file format. Engineers learn which columns mean what and pass that knowledge informally to the next person. The workarounds accumulate.

This is architectural debt — the hidden cost that accrues when a structural decision made for short-term convenience must be compensated for, repeatedly, over the life of the program. Unlike financial debt, architectural debt does not appear on a balance sheet. It appears in engineering hours spent reconstructing context that should have been captured at the source. It appears in delayed root cause analysis when yield drops and the data cannot answer the question directly. It appears in the gap between what the data contains and what manufacturing needs to know.

> **Architectural debt:** *The hidden cost that accrues when a structural decision made for short-term convenience must be compensated for, repeatedly, over the life of the program. It does not appear on a balance sheet. It appears in engineering hours.*

The Debt Is Invisible Until It Isn't

Unlike a test that fails or a system that crashes, architectural debt does not trigger an alarm. Production runs. Data accumulates. Files land in folders. The decoder opens them. Everything appears to be working because, at low volume, it is.

The debt is invisible precisely because the flat file architecture succeeds at what it was designed to do. It captures data. It stores it. It can be opened and read. No error is thrown. No process fails. The system looks healthy from every angle that development validation measures.

The debt becomes visible the first time someone needs to answer a population-level question and cannot. A yield drop triggers a request: which channel is failing, and across how many units? The answer should come from the data. Instead it requires pulling files, running the decoder one at a time, manually aggregating results across a folder that now contains thousands of logs. What should be a query becomes a project. The debt that has been accumulating since the first test run arrives, on schedule, at the worst possible moment.

The Interest Payments

Architectural debt does not sit idle. It is paid in recurring installments, in forms that rarely get attributed to the underlying cause.

Every time a yield problem requires root cause analysis, someone writes a one-off parser or extends the decoder to handle a new question. That script gets saved somewhere, used once or twice, and becomes part of an informal toolkit that only the author understands. Every time a process change requires before-and-after comparison, an engineer manually extracts data from two sets of flat files and reconciles them in a spreadsheet. Every time a new test station is added or a new channel is introduced, the column definitions shift and every existing analysis tool must be updated to match. These are not engineering failures. They are interest payments on structural debt that was taken on the day the first flat file was written.

> *Every one-off parser, every manual aggregation, every analysis spreadsheet rebuilt after a column change — these are not engineering tasks. They are interest payments on a structural decision made in development.*

The Knowledge Transfer Problem

Flat file architecture depends heavily on tribal knowledge. The column definitions are not self-documenting. The channel indexing convention was decided by the engineer who wrote the loop and lives in their memory, not in the data. The timestamp format, the repeat counter logic, the meaning of a particular flag value — these are known to the people who built the system and passed informally to whoever works with it next.

When that engineer moves on — to another program, another company, or another role — the decoder becomes archaeology. The person inheriting the system must reverse-engineer what the columns mean, reconstruct the logic the original author encoded in the analysis tool, and validate their interpretation against units whose behavior they may not fully understand. This is not a personnel problem. It is a data architecture problem. Structured data is self-describing. A flat file is only as interpretable as the person who built it.

> *Structured data is self-describing. A flat file is only as interpretable as the person who built it — and that person will not always be available.*

The Scale Inflection Point

There is a specific volume threshold at which flat file architecture breaks down as a manufacturing intelligence tool. It is not a sudden failure. It is a gradual transition from manageable to unworkable — and it is entirely predictable, yet almost never anticipated.

At ten units, the decoder works fine. At one hundred units, it is slow but functional. At one thousand units, manual file-by-file analysis is no longer a realistic option for population-level questions. At ten thousand units, the folder of flat files is a liability — full of data that answers nothing without engineering effort to extract it. The inflection point is crossed somewhere in that range, and it is crossed at exactly the moment production pressure is highest and engineering bandwidth is lowest.

The compounding effect of architectural debt ensures that the cost of addressing it grows with every unit produced under the old structure. The longer the flat file architecture runs, the more historical data exists in a format incompatible with the structured model that would replace it. Trend analysis breaks at the transition point. Yield comparisons require two separate workflows and manual reconciliation. Correcting the architecture does not recover what was lost — it only stops the accumulation from that point forward.

> *Designing structure into data collection from the beginning costs a fraction of what it costs to reconstruct that structure after the fact — and reconstruction never fully recovers what chronological logging discarded at the source.*

7. Multichannel Systems Magnify the Risk

The challenges described in this paper exist in any test system that produces flat file output. In multichannel systems, they are amplified.

Multichannel UUTs introduce execution complexity that flat file logging handles poorly at scale. Channels are interdependent — a failure in one may mask or trigger conditions in another. Test sequences may vary by channel based on conditional branching. Repeat counts may differ across channels within a single test run. Stress and burn-in cycles generate measurement sets that grow rapidly in volume.

In a timestamped flat log, all of this appears as a sequential stream. The log faithfully records what happened. It does not preserve what belongs together, what is nested, what correlates, or what deviates from expected behavior across the channel population.

> *A timestamped log tells you what happened in sequence. Structured data tells you what belongs together, what correlates, and what deviates. In high-complexity systems, that distinction determines whether root cause analysis takes hours or weeks.*

When a failure occurs in a multichannel system and the data is execution-driven, reconstructing the failure context requires cross-referencing timestamps, identifying channel indices in sequential rows, separating repeat iterations from first-pass results, and correlating conditions that may be separated by thousands of rows in the log file. For a single engineer debugging a single unit, this is manageable. For a manufacturing engineer investigating a yield problem across a production population, it is not.

The loop structure that makes multichannel test software efficient at runtime is precisely what makes its flat file output difficult to analyze at scale. The architecture that serves execution does not serve investigation.

8. A Strategic Shift in Mindset

The change required is not primarily technical. The tools, frameworks, and platforms to implement structured data collection exist. The engineering effort required at the test executive level is real but not prohibitive. What is harder to change is the question that gets asked.

Development teams ask: where do we write the results file?

The question that should be asked is: how will this data need to be analyzed when something fails six months into production?

These are not the same question. The first optimizes for getting data out of the test system. The second optimizes for making that data useful when it matters most — under production pressure, with yield on the line, when the answer needs to come from the data and not from the engineer who built the decoder.

> **The strategic question:** *Not where to write the results file. How will this data need to be analyzed when something fails six months into production?*

This shift does not require abandoning development speed. It requires separating two concerns that development timelines tend to collapse: execution logging, which captures the runtime sequence for debugging, and analytical storage, which organizes results for manufacturing intelligence. These are different outputs serving different purposes. Designing for both from the beginning does not slow development. It prevents the architectural debt that accumulates when manufacturing is left to work with what development left behind.

For contract manufacturers, this means receiving test code that produces structured output — not code that requires a custom decoder to interpret. For OEM manufacturing teams, it means that the formal rewrite is not just a translation of development functions into a controlled environment, but an explicit redesign of the data model to serve production demands. For program managers and engineering leadership, it means recognizing that the test system deliverable includes the data architecture — and that architecture should be evaluated against manufacturing requirements, not just development criteria.

9. Conclusion

Flat file logging is not a flawed technology. It is an appropriate solution for the conditions under which it is typically built — small UUT populations, rapid iteration, engineering-level analysis, development timelines that do not accommodate architectural planning.

The problem arises when that architecture follows the code into production. When a folder full of timestamped logs becomes the data foundation for a factory running hundreds of units per day, the mismatch between what the architecture was designed for and what manufacturing demands of it becomes expensive — in engineering time, in delayed root cause identification, in production risk that accumulates invisibly until it surfaces as a yield problem.

Organizing data within the test executive — preserving hierarchy, context, and relationships from the moment of measurement — transforms test output from a historical record of execution into operational intelligence.

> *The transition from prototype to production does not begin at the factory. It begins in the structure of the first line of test code — and it must be designed before the decoder gets built.*

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 this paper examines. 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 1 of a three-part series on test data architecture and manufacturing intelligence.

For inquiries about test data architecture or manufacturing intelligence strategy, contact Circuit Check Inc.