Table of Contents

[“Bumper Sticker” statements](#_“Bumper_Sticker”_statements)

[The Rest of the Story](#_The_Rest_of)

[Is there a No-/Extremely-Low-Development Solution for Flight Test Data?](#_Is_there_a)

[How is Flight Test Organized?](#_How_is_Flight)

# “Bumper Sticker” statements

* In flight test data, there’s no such thing as a “low-development” (or less) solution
* The flight test data “stack” and “pipeline” are deep and there are no COTS solutions that cover either end-to-end, requiring AT LEAST integration development
* Test POINTS are PLANNING artifacts, linking all levels of requirements together, most specifically a TEST METHODOLOGY and a TEST CONDITION
* Test CARDS are PROVISIONING artifacts, linking all levels of resources together
* Test RUNS are EXECUTION artifacts, linking the system to reality, most specifically a start and end timestamp pair, with all the associated data
* Test MISSIONS are EXECUTION constructs, collections of RUNS
* EVENT MARKERS are single timestamps, user-generated (could be automated) indicated a moment of significance

# The Rest of the Story

## Is there a No-/Extremely-Low-Development Solution for Flight Test Data?

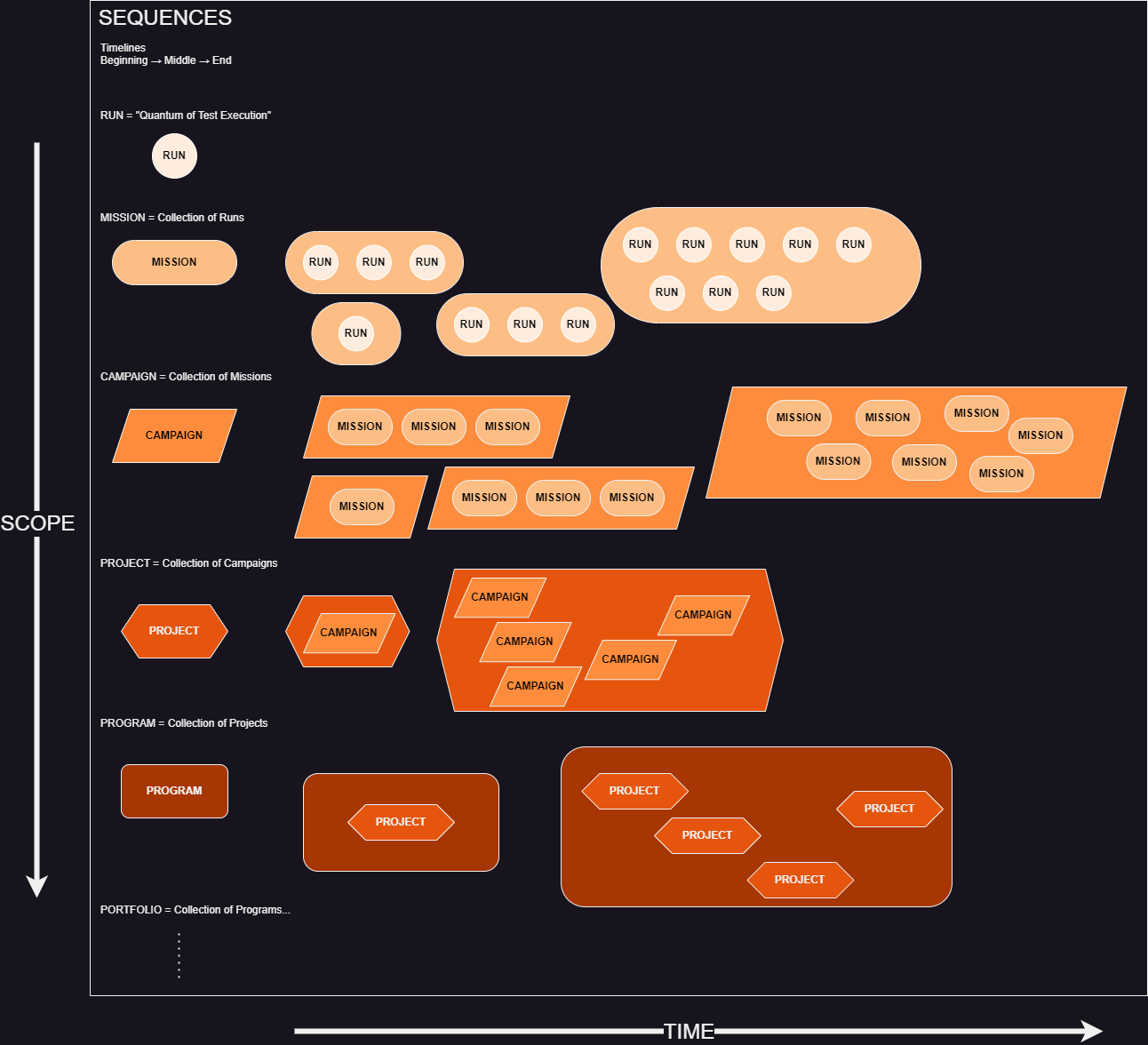
1. There’s not a “No-development” solution in the space of flight test data. There’s not even a “Low-development” solution in the space of flight test data
   1. Historically, either side of the “buy/build” or “in-house/out-house(!)” flight test data decision has involved significant development
   2. Simply put, there hasn’t been sufficient scale of “standard” data handling to have something that has been developed to be “out-of-the-box” in the full stack, where the stack is (notionally, not including the concepts of hypervisors and virtual machines)
      1. Bare metal (aircraft and/or computer hardware)
      2. Operating system (aircraft and/or computer)
      3. App backend: Data Plane
      4. App backend: Control Plane
      5. App frontend
      6. Business logic
      7. User interface
   3. Some places have developed some or most of this in-house
   4. At some point, though, some development is required
      1. Even with something “out-of-the-box” like IADS, there is significant integration development required, both on the backend to get the data in, and in the business logic and user interface to build the display components specific to the particular test, even if IADS itself is “COTS”
      2. Even Grafana dashboards need to be constructed, tested, and iterated for each different subsystem engineer
      3. Winplot, Python, JuptyerHub, etc are all “established” apps/platforms, but they impose development burden on Engineering, as well, AFTER the development burden on IT (minimal for Winplot, larger for Python, large for JupyterHub)
         1. Some of that “burden” is actually “blessing” because it allows freedom of exploration, but as exploration transitions to “exploitation” (in the most positive sense of making use of a resource), freedom needs to transition to discipline (configuration control, versioning, standards)\*
2. Within flight test data there are several “stages” through which the data pass, all of which require care and feeding and yes, development (at the very least integration)
   1. Sensing
      1. existing vehicle transducers
      2. “special/test-specific” instrumentation
   2. “Raw” data transport
      1. vehicle message busses
      2. dedicated “discrete” or “analog” signal lines
      3. “special/test-specific” instrumentation message busses
      4. “special/test-specific” signal lines
   3. Onboard recording
   4. Telemetry to ground station
   5. “Real-time” data display
   6. “Real-time” data transformation (“derived” parameters [read: math operations performed “on-the-fly”] such as unit conversions, combining multiple parameters into a new parameter, etc)
   7. “Post-run” data review/analysis to DECIDE next action [repeat test point, next test point, mission termination safety problem, mission complete technical ] (on a time-scale of seconds-to-minutes, pause/rewind/replay/pause)
   8. “Post-mission” data review/analysis to DECIDE next action [repeat mission, next mission, campaign termination safety, campaign complete
   9. Reporting with publication-quality data products (tables, charts, interactive displays, etc)

## How is Flight Test Organized?

1. Flight test that includes any more than a single run gets complex quickly
2. The test process can be thought of as a “network” (or “graph” [Graph (abstract data type) - Wikipedia](https://en.wikipedia.org/wiki/Graph_(abstract_data_type))) of interconnected objects
   1. A good approximation of the main “nodes” in the graph are
      1. **Planning** (produces requirements)
      2. **Provisioning** (secures resources)
      3. **Executing** (generates runs)
      4. **Analysis** (produces results)
      5. **Reporting** (produces recommendations for action)
   2. Here’s a graphic (pun intended) I’ve been developing to communicate our test and data management needs to internal and external stakeholders (still WIP) Diagram, schematic

      Description automatically generated
   3. The reason they’re all connected is because very rapidly all of these nodes MUST be linked, otherwise context and meaning (and entire programs) are lost
   4. This linking typically occurs in experts’ brains, or very loosely in documents (Word, PowerPoint, Excel, etc) that refer to each other, but without actual links
3. During PLANNING, program requirements (certification, airworthiness, performance targets, cost targets) are distilled into TEST POINTS
   1. A TEST POINT is a PLANNING device that is a combination of
      1. a test method (procedure/technique)
      2. at a particular test condition (energy state, space-time location(s), system configuration, etc)
   2. A Test Point is the “quantum” of test planning; the smallest unit of planning. Test planning exists to derive a set of test points that will satisfy the test objectives.  
      Diagram

      Description automatically generated
4. During PROVISIONING, the requirements and test points from PLANNING are converted into TEST PROCEDURES (aka test cards) as a way to COMMUNICATE across the test team and the org
   1. A Test “Card” is the “quantum” of test provisioning; the smallest unit of provisioning. The test card describes all the resources that need to be brought to bear, and how they will be employed.  
      A picture containing shape

      Description automatically generated
5. During EXECUTION, REQUIREMENTS and RESOURCES engage with REALITY during RUNS
   1. I don’t have the EXECUTION node detail built, yet, but the idea is that during execution, Test Procedures attempting to satisfy the requirements of Test Points are “Run”
   2. Thus, the “Run” is the “quantum” of test execution; the smallest unit of execution. During the test run, the system and system operator(s) generate the data required to determine results and meet test objectives.
   3. A run is a sequence with a beginning, middle, and end. In its simplest implementation, a “run” consists of two timestamps associated with the beginning and end of the run.
   4. Additional fidelity comes when associated directly with datasets, with additional context/metadata, etc.
   5. Runs can be collected into missions, missions into campaigns, etc
   6. Each level in scope is itself a sequence, with a beginning, middle, and end.  
      
6. It is traditional in flight test to have an “event marker” that annotates a single moment in time. These are very useful for navigating the data and attaching context and meaning.
   1. When annoted in real-time or near-real-time, an event marker is an EXECUTION tool that helps analysis and reporting.
   2. When annotated post-run or post-mission, an event marker is an ANALYSIS tool that helps analysis and reporting.
   3. One way to implement an event marker is as a single timestamp.
   4. Another would be to make it a “special case” of a run, in which the start and end timestamps have the same value.

\* Polarity: Two things that are both right and good and can’t exist without each other <https://www.sloww.co/polarity-thinking-101/>

Classic example: Breathing…Exhale-Inhale  
Diagram

Description automatically generated

Explore (turn unknowns into knowns) & Exploit (turn knowns into advantage)  
Diagram

Description automatically generated