Striving for DevOps on a Large Flight Mission

Lessons learned from the ICESat-2/ATLAS flight software test effort

Joe-Paul Swinski

NASA/Goddard Space Flight Center

Flight Software Branch

December 2017

Scope of ICESat-2/ATLAS Effort

Mission Scope: Class C mission. Sole instrument on ~$1B observatory. 8 year development effort. 5 year primary mission. Significant technology development.

Staffing Scope: 19 different people worked on ATLAS flight software at one point or another. At our peak there were 11 people on the team, on average there were 8 people on the team.

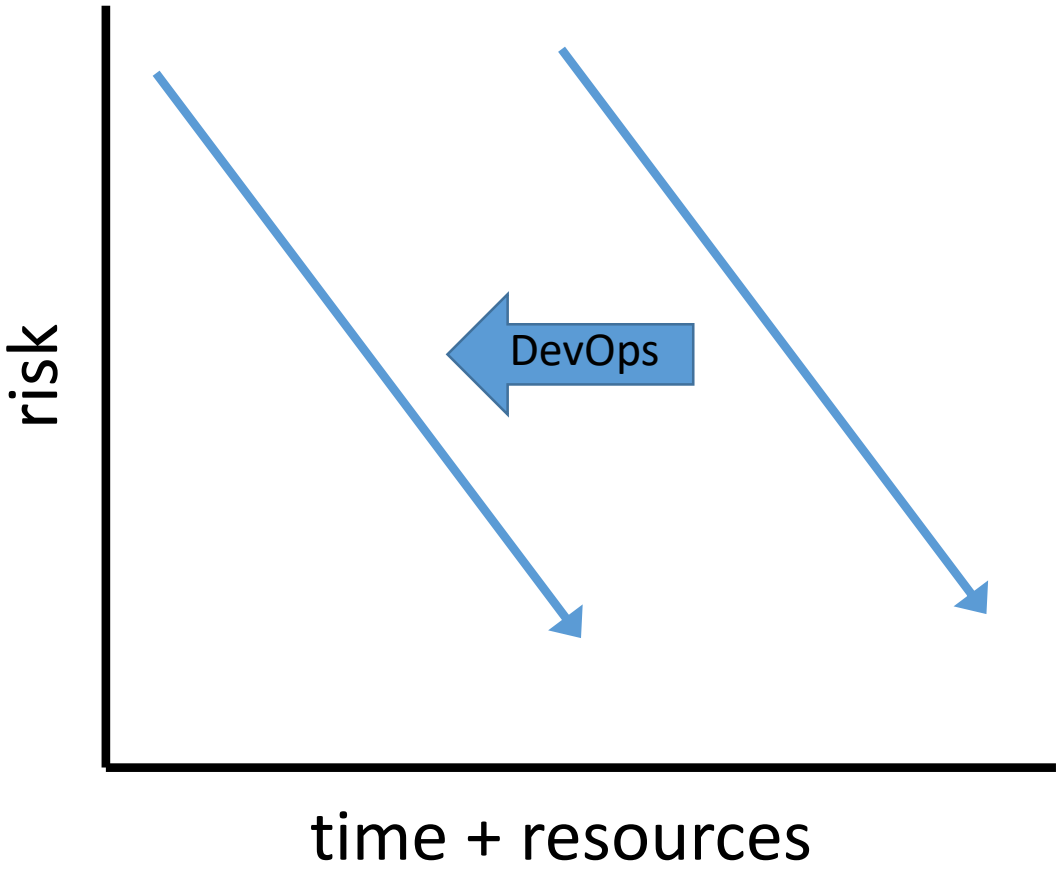
Hardware Scope: Four processor boards, of which two unique (1 Rad750, 3 Leon3FT). SpaceWire network as backplane with 12 nodes and three routers.

Requirement Scope: C&DH, four control algorithms (PID thermal control, mechanism control, geolocation, ground selection), computationally intense and sophisticated ground selection algorithm. Variable data rates from 1Mbits to 80Mbits, 6Mbits nominal in flight. For ground testing, often closer to 40Mbits.

Software Scope: Four CSCIs (two PROM boot loaders at class B, two EEPROM applications at class C). 24 tasks (15 on Rad750, 9 on Leon3FT). ~85KSLOC of flight code. ~500KSLOC of total code including GSE, utilities, and reuse (but not operating systems).

What is DevOps

DevOps is a catch word for *delivering faster*



- Tools and practices that shorten the time from ***dev***elopment to ***op***erations: the amount of time and resources necessary to translate a requirement into a delivered product.
- Programmatically merging (or bringing closer together) the team that ***dev***elops the software with the team that supports the ***op***eration of the software.
- Combining or extending the infrastructure used to **dev**elop the software with the infrastructure that hosts/delivers/**op**erates the software.

Why is DevOps a Worthy Goal

What We Care About

Reliability - Does it behave the way we intended it to behave? (e.g. bugs, brittle design, unreasonable operating constraints)

Stability - Does it behave the way they expect it to behave? (e.g. are we breaking interfaces, breaking previous understandings, or changing behaviors)

What We Can Do

Time & Money – we can take more time, buy better tools, hire more people (and hopefully manage them well)

Risk – we can accept a greater chance of failure or problems

People – we can hire/retain better engineers

Process - we can make better use of what we have already

Why is DevOps a Worthy Goal

What We Care About

Reliability - Does it behave the way we intended it to behave? (e.g. bugs, brittle design, unreasonable operating constraints)

Stability - Does it behave the way they expect it to behave? (e.g. are we breaking interfaces, breaking previous understandings, or changing behaviors)

What We Can Do

Time & Money – we can take more time, buy better tools, hire more people (and hopefully manage them well)

Risk – we can accept a greater chance of failure or problems

People – we can hire/retain better engineers

DevOps

Process - we can make better use of what we have already

Why is DevOps a Worthy Goal

What We Care About

Reliability - Does it behave the way we intended it to behave? (e.g. bugs, brittle design, unreasonable operating constraints)

Stability - Does it behave the way they expect it to behave? (e.g. are we breaking interfaces, breaking previous understandings, or changing behaviors)

What We Can Do

Time & Money – we can take more time, buy better tools, hire more people (and hopefully manage them well)

Risk – we can accept a greater chance of failure or problems

People – we can hire/retain better engineers

DevOps

Process - we can make better use of what we have already

This is nothing new, and DevOps is not the first (nor will be the last) process to promise "free" improvements to what we care about – but due diligence says to take a look and see if there is anything we can steal and use to improve what we do.

DevOps on ATLAS

1. We never thought about it as “DevOps”

Back in 2010 a couple of us on the team came together and had similar ideas regarding our requirement → development → test → delivery cycle. Namely – could we make it fundamentally shorter?

Typical projects we had all worked on had about a 3 to 8 month release cycle. Any new requirement or change would first have to wait until the next release to be given to the test team, which would then take 3 months or more to test, before being delivered.

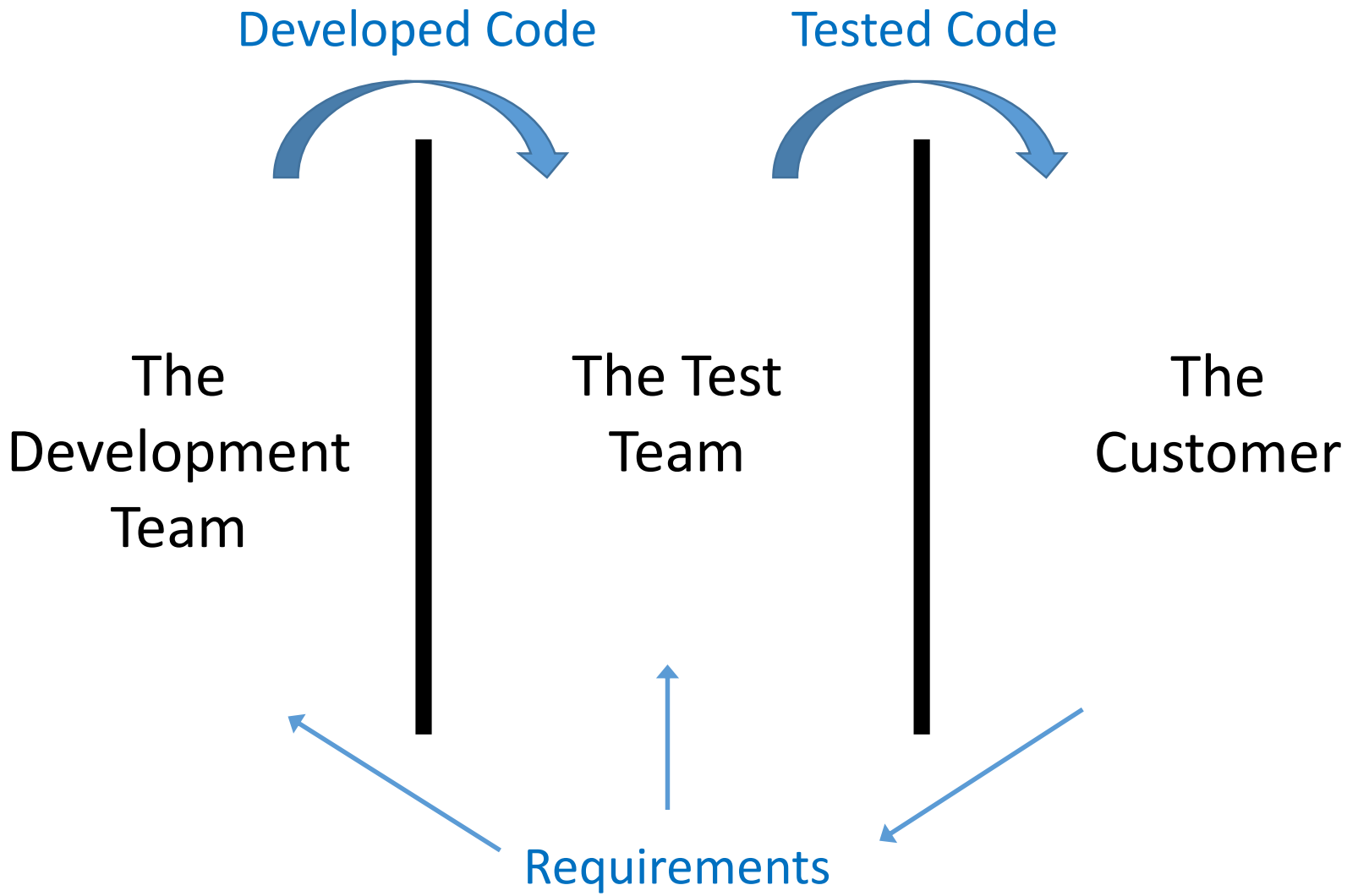
It just seemed wrong… so we asked: how short could we make this cycle? A few weeks? A few days? What about a few hours? Could we develop a process where we received a new requirement in I&T on a Monday and by Thursday of the same week, without any compromises, we delivered a fully tested release of code to the observatory with no greater risk than the current approach?

2. We called it “testing to trunk”

We were using SVN at the time and so “testing to trunk” meant that the test team was only to develop tests for the code that was sitting in trunk (i.e. the tip of development).

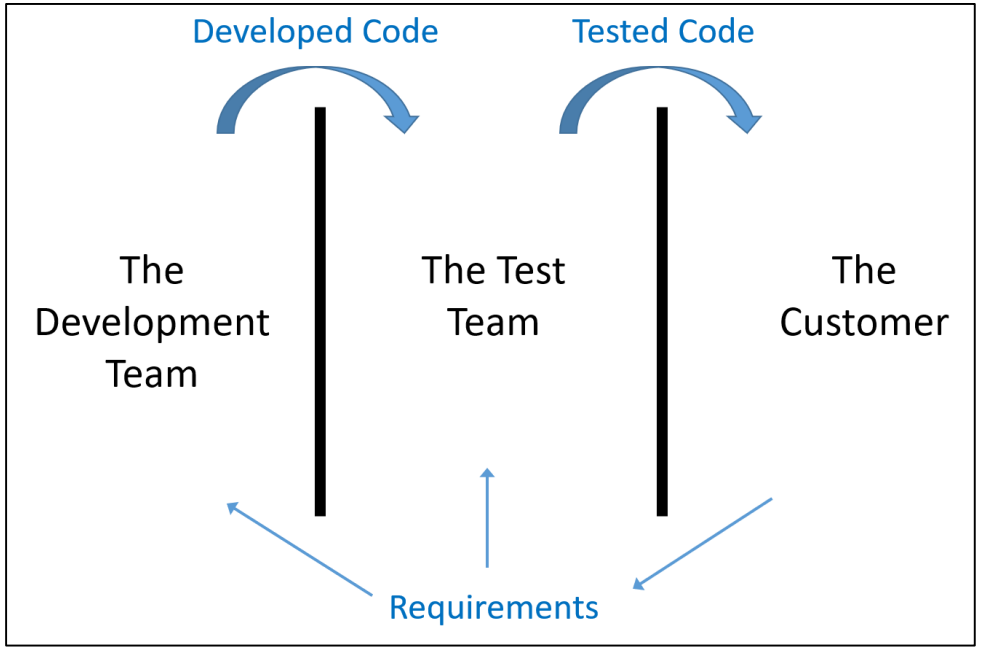
It caused quite a stir in the branch and I had to convince multiple people to let me do it, but finally the branch head at the time gave me the go ahead. In the end it turned out that “testing to trunk” required a lot more than just testing to the trunk.

Historical Approach: The Walls of Truth



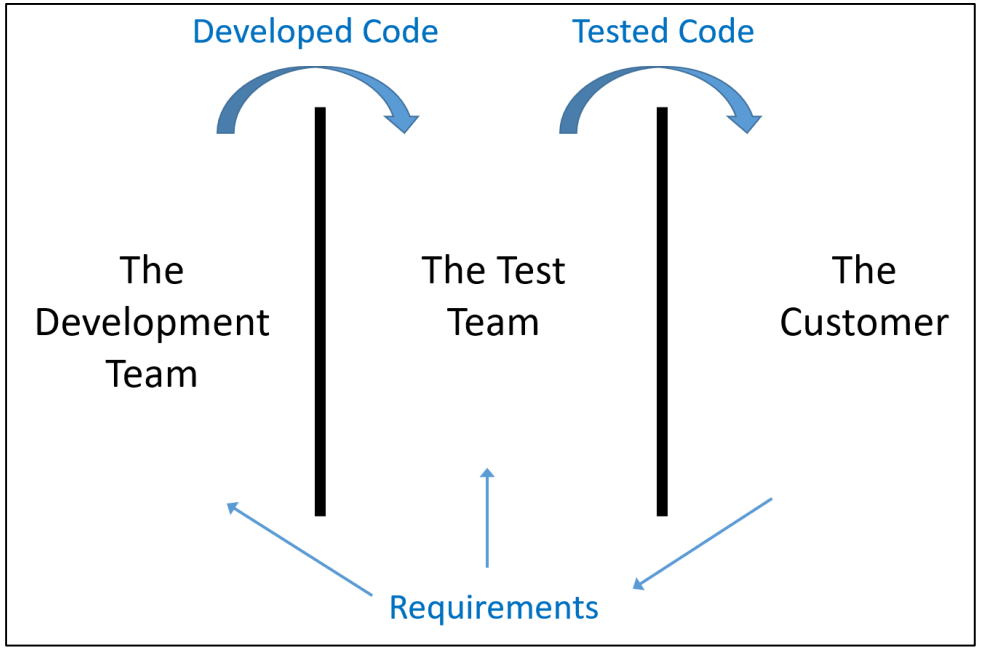
Advantages to The Walls of Truth

- The independence of the test team protects against developer bias
- There are multiple fresh looks at each requirement
- The development and test activities can be managed separately (to some extent) allowing for separate leads, and separate skill sets (e.g. C programmers vs. STOL script writers)
- By the time the customer receives a piece of code, it has been tested, unchanged, for nearly six months (reliability and stability have been bought with time)

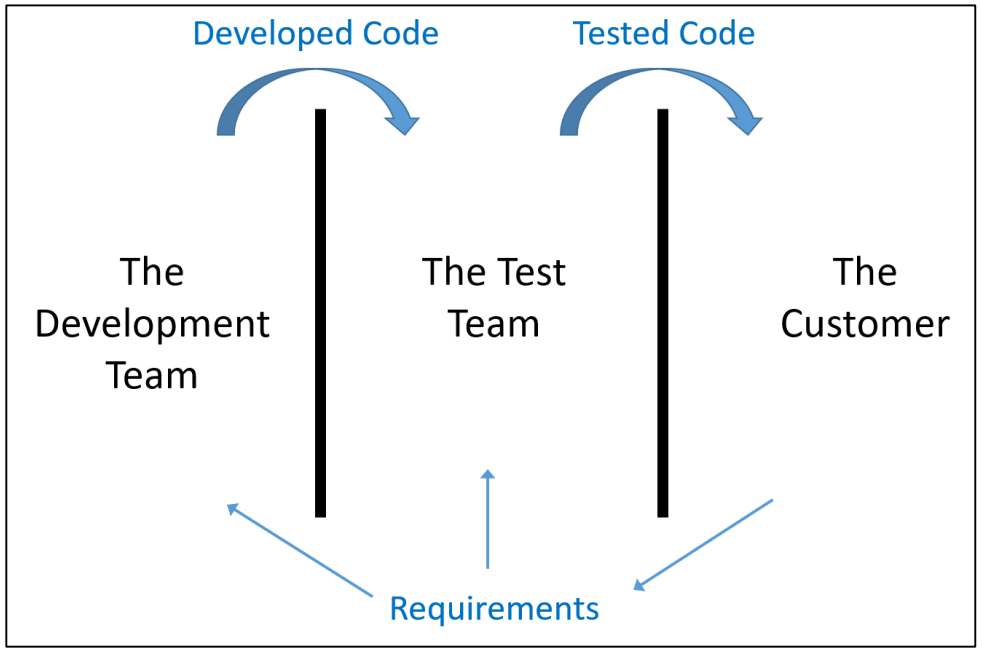


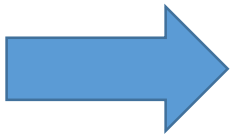
Disadvantages to The Walls of Truth

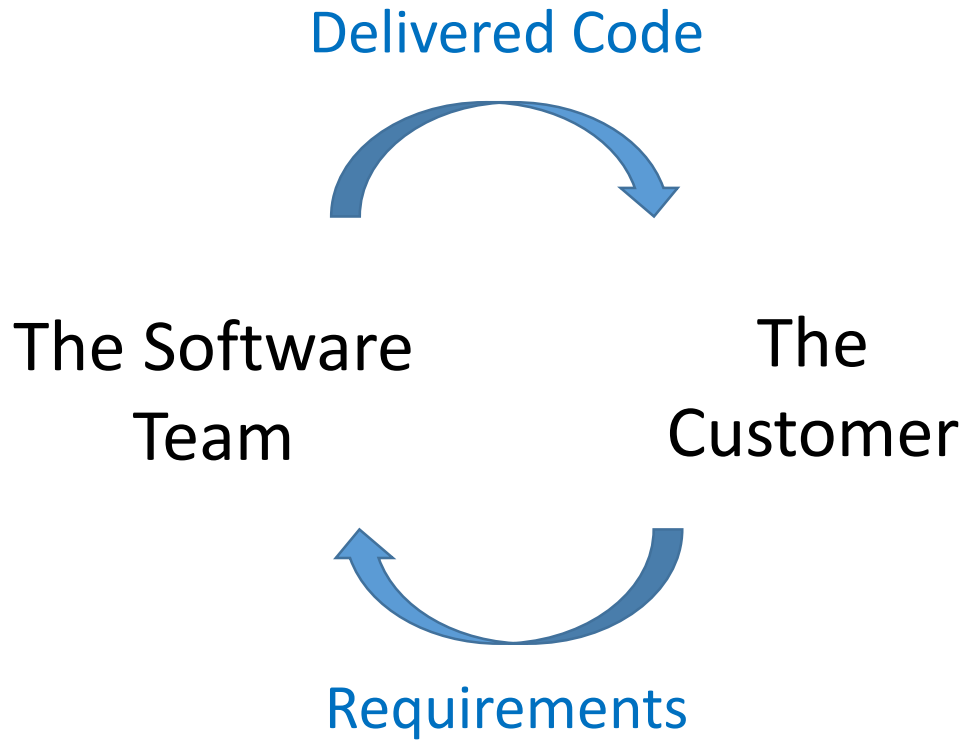
- The rest of the project is operating in the “now” and our code reflects an understanding of the project that is 6 months (or more) old.
- In I&T, when we need to make a change right away, we have to “break” our process to deliver it – either by accepting higher levels of risk, or asking for heroics from our team.
- This approach has inherent inefficiencies in the use of our resources. Software that is already changed is still having resources applied to it.



The Paradigm Shift







The Fundamental Changes

The Goal: Greatly reduce the amount of resources (time, people, risk) it takes to translate a needed change into a delivered product.

Fundamental Change #1: Design it so that it can be automated.

Fundamental Change #2: Design it so that it can be measured.

Design it so that it can be automated

Mindset #1: Computers run tests not humans.

- Every project I had worked on until ATLAS had testers run their own tests, or at the very least, a knowledgeable tester would run a fellow tester's test. **This ties the very expensive human resource to an otherwise inexpensive process of executing a test.**

- If I want to run tests 24/7 continually for the life of the project, and I want to run them concurrently on three different strings; then it becomes a logistical staffing problem to have people support those tests. It must be automated. **Computers run tests, not humans.**

- **Tests that are not written from the beginning to be autonomous almost never can be executed autonomously.** The way the test is written needs to be designed from the beginning as an autonomous test.
  - It can never halt
  - It must check all error conditions itself and report them centrally in the end
  - It must make any branching decisions on its own
  - The test bed infrastructure must support anything the test wants to do (no probing, no pushing buttons)

Design it so that it can be automated

Mindset #2: Running all the time means handling all failures.

- The test infrastructure needs to be able to handle gross failures like computers crashing, tests timing out, things getting corrupted.

- When we are used to running tests as discrete activities, if a test inadvertently causes EEPROM to get corrupted, then the next time we run a test we see a failure, we investigate, notice the EEPROM corruption and fix the problem. We lose maybe one test cycle. If we are running dozens of tests an hour, and the tests are run unattended, having one test leave EEPROM corrupted will cause dozens and dozens of tests to fail and the loss of a great amount of test time.

- Computers crashing used to be a valid excuse for why there is a delay in a development/test effort. But if you start to scale your test operation to 24/7 and start utilizing multiple strings, then handling failures become a part of normal operating procedures.

Design it so that it can be automated

Mindset #3: Computers are bad at ambiguity.

- When humans run tests, we can be relied on to make fairly descent judgment calls when things are ambiguous. We're good at thinking on the fly, investigating, and resolving issues. Computers are horrible at ambiguity.

- As a result, every operation must unambiguously succeed or fail. Nothing in between. Every telemetry point checked has to have an acceptable range. Plots can't be analyzed later. The test system must trend towards a set of assertions and away from a set of analysis.

- The place where this was most practically felt was in command verification.
  - Latency in command/telemetry verification caused more false alarms that any other single source of reported failures; yet we design our systems with ambiguous command verification techniques like the command accept and reject counters.
  - If telecommands can be generalized as nothing more than RPCs, do we see similar techniques for RPC acknowledgements being used in ground systems?
  - Are the reasons (e.g. small telemetry footprint) for command accept/reject counters still valid in the systems we build today?

Design it so that it can be measured

Mindset #4: Measure how code behaves, not how it works.

- We just talked about running a series of automated tests that verify if the code works. Automated tests map to requirements and use cases that verify and validate if the code works. If the code passes the test, then it works. Measuring how code behaves answers the question of does it work all the time.
  - My code passes the test when it is run after a test that turns off diagnostic data; does it pass the test when it is run after a test that turns on diagnostic data.

- Software of any size cannot have all permutations of inputs checked. To combat this, we measure how it behaves, all the time, across varying conditions.
  - Does my test system allow enough variation of input (time, event sequencing, axillary inputs) to represent a fair set of the real world it will be delivered into.
  - Have I instrumented the test system sufficiently in order to tell if expectations of a behavior are being met.

- **It is a continuous measure of behavior, not a discreet test of correctness**. For instance, if I have an expectation that packets are never corrupted; instead of just writing a test that maps to a requirement that analyzes a set of data to see if the CRCs are correct, also setup a ground processing tool that continuously monitors every packet that is produced by the flight system, and alerts the team for any failed CRC check.

Design it so that it can be measured

Mindset #5: Maximize the time a behavior has been measured.

- The way we are going to be able to deliver software to an observatory that was changed only a few days ago, or even hours ago, and still have tremendous confidence in its reliability and stability, is by maximizing the amount of time every behavior in the code is measured.

- For any release of code, we could in theory enumerate all of the behaviors that the code exhibits. We then in theory could enumerate all of the behaviors our test system measures. We could then ask, how long has each behavior in the code been measured by our test system?

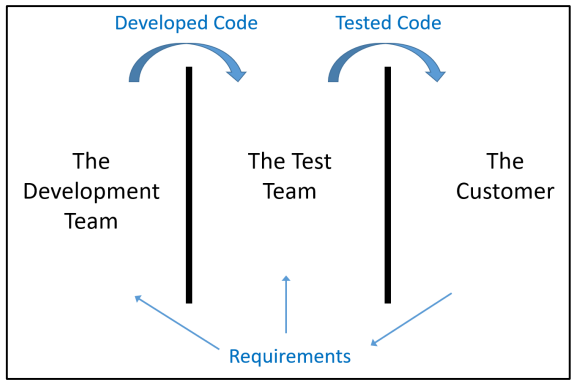
- If this is what we are trying to maximize then it is in our best interest to immediately begin measuring a behavior as soon as possible. Testing an outdated piece of code via the original "walled" method is counter productive and leads to ***less*** reliable and stable code.

Example of automation/measuring benefits

Behavior	Date Added	Days tested if testing started 3/10/15 and code released 4/1/15.	Days tested of testing started right away and code released 3/10/15.
Time stamps increment by one second for each second that passes.	11/2/2014	22 days	128 days
The temperature of fixture A is held to 10C +/- 1C.	1/15/2015	22 days	54 days
The instrument manager No-Op command increments the command counter by one.	1/30/2015	22 days	39 days
A single event upset in memory is corrected and reported in telemetry.	3/4/2015	22 days	6 days

- I can target my resources to mitigate the risk of recent changes, but there is nothing I can do to reclaim lost test time.
- Consider the difference in test time if the first approach doesn't include continuous measurements and only tested requirements discreetly.

The Updated Paradigm Shift



Flight Software
Automated Test Infrastructure
Continuous Measurements

Requirements

The Customer

The Software Team

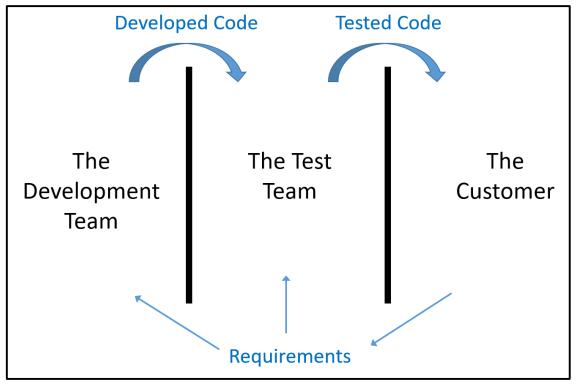
Test Bed

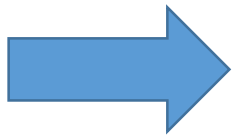
Delivered Code

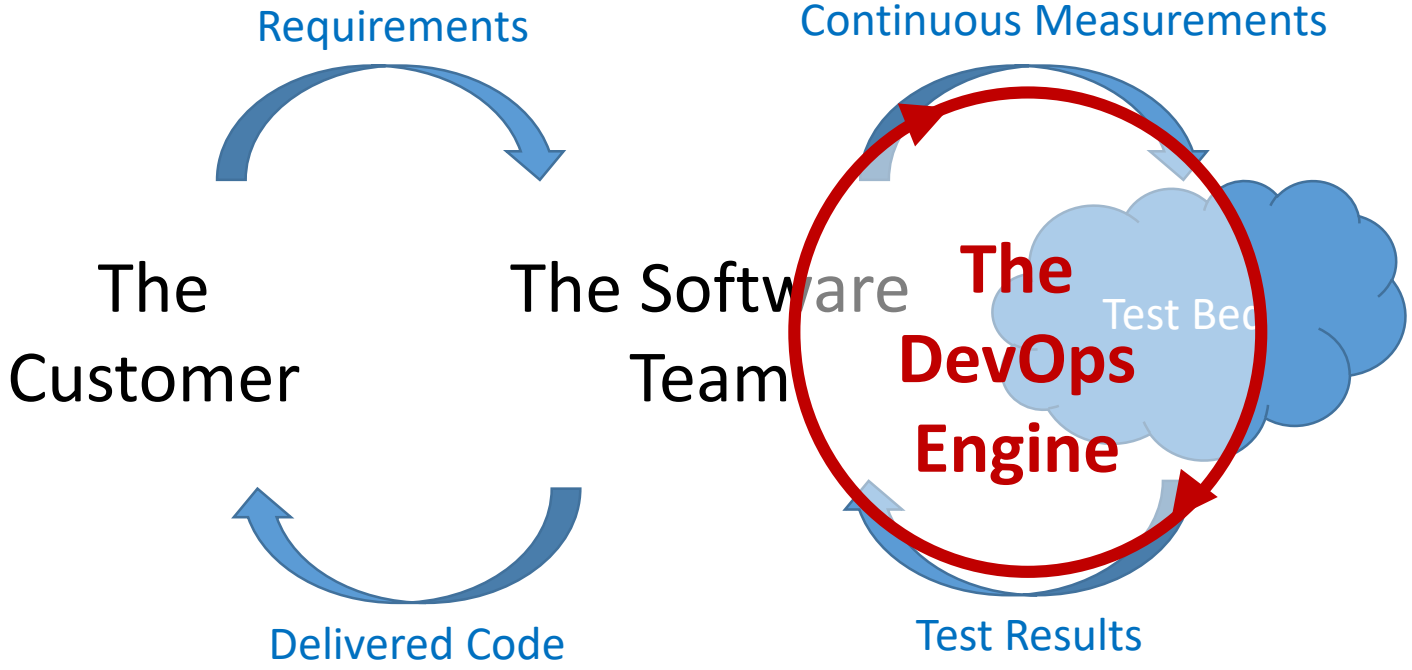
Test Results
History of Performance
Tested Flight Software

Change from building and delivering software to building a development/test bed that delivers software

The Updated Paradigm Shift







The take away lesson I see from DevOps practices is to invest in the above development "engine". Instead of focusing on producing a great software product, we focus on producing a great software producing product.

Practical Lessons Learned

Four Steps in the Right Direction

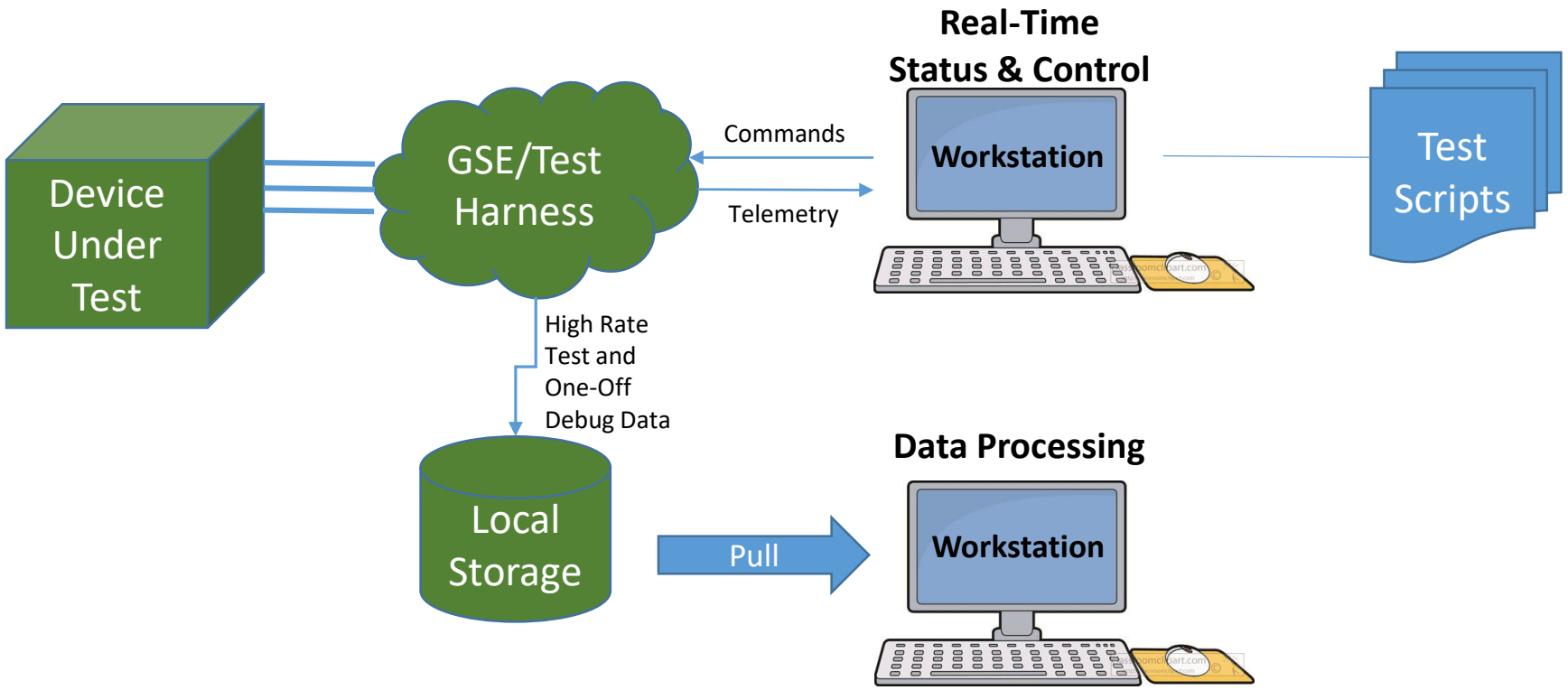
1. Data Centric Test Bed
2. Desktop Simulation Environment
3. Test to Trunk
4. Owning the GSE

Four Steps in the Wrong Direction

5. STOL testing language
6. Manual data versioning
7. The plight of timeouts in command verification
8. The behemoth test procedure

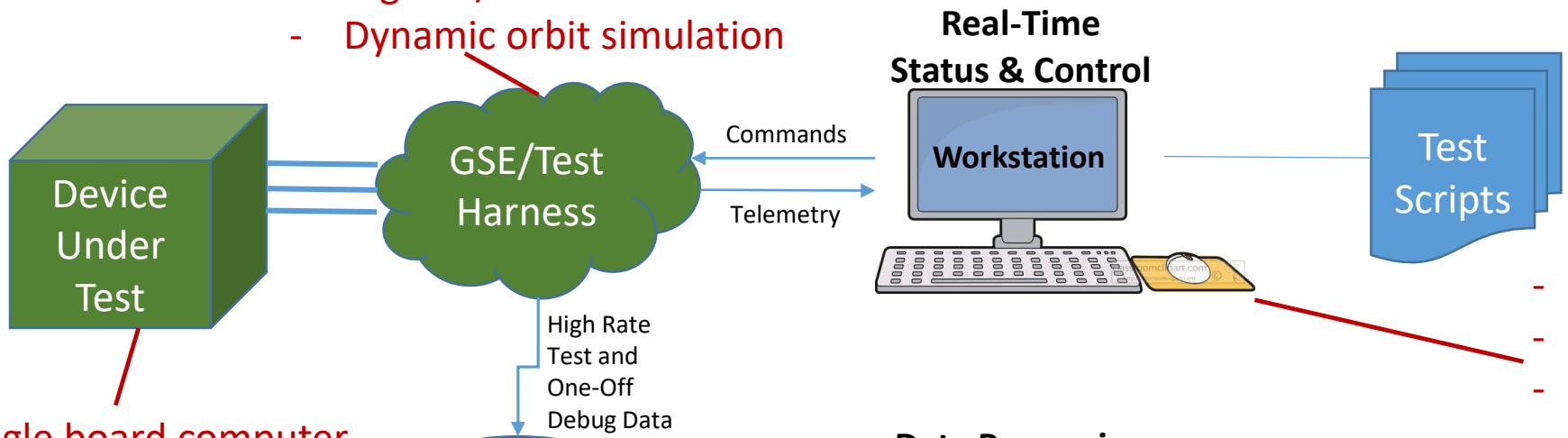
Lesson #1: Data Centric Test Bed

We started out connecting our workstations directly to our GSE.



Lesson #1: Data Centric Test Bed

- Custom logic boards
- 1553 bus controller
- Digital I/O cards
- Dynamic orbit simulation



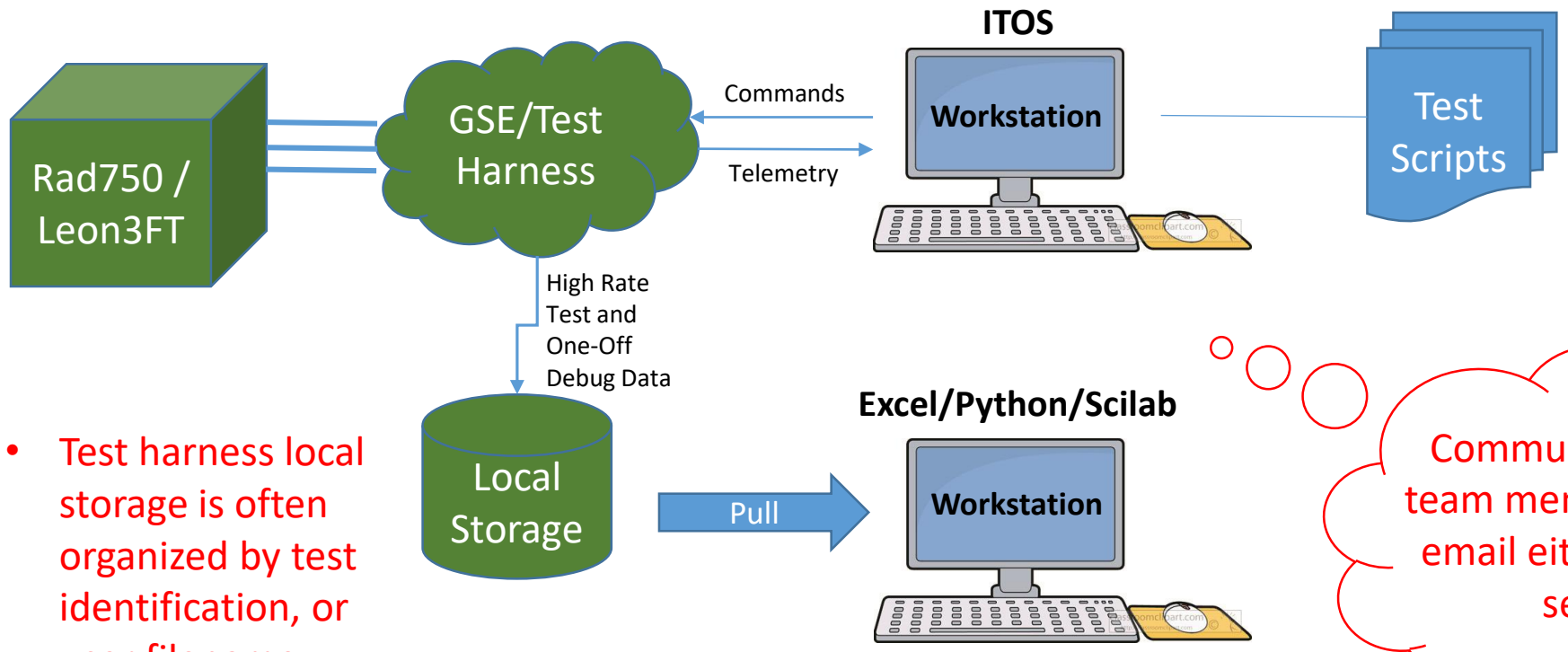
- Single board computer hosting flight software
- Signal processing boards
- Spacecraft bus
- Fast steering mirror and mechanism

- ASIST
- **ITOS**
- GSEOS
- COSMOS
- AstroRT

- **Excel**
- MATLAB
- **SciLab**
- **Python**

Lesson #1: Data Centric Test Bed

We started out with files and emails…



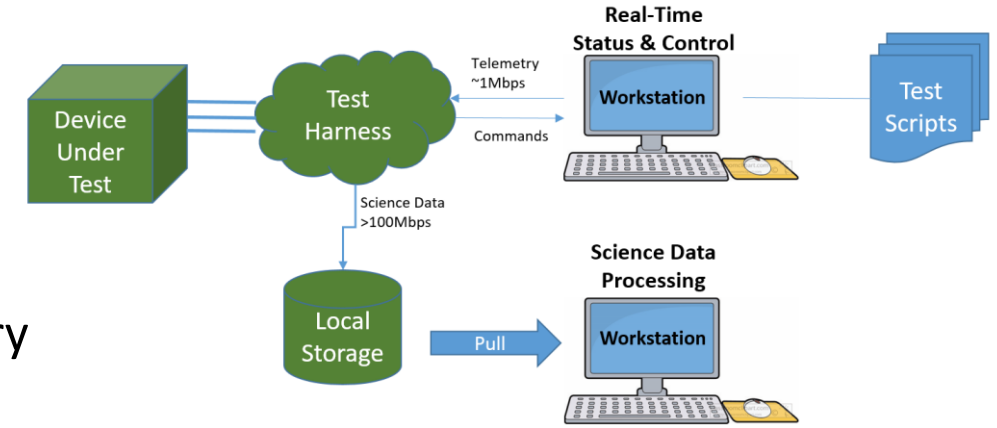
- Test scripts are not aware of processing results
- Test harness local storage is often organized by test identification, or user filename choice
- On a separate computer performance results are organized by test identification

Communication between team members was through email either pointing to or sending files

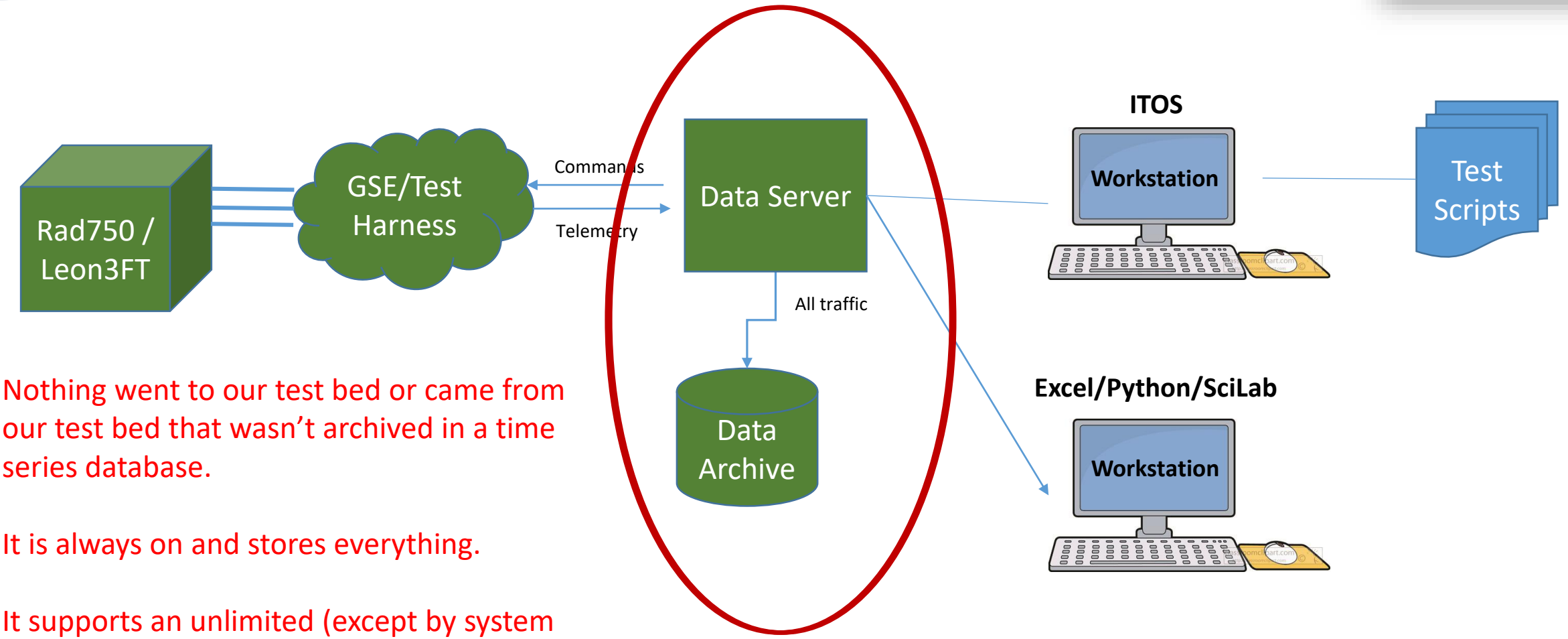
Lesson #1: Data Centric Test Bed

An example work flow from how we started:

- The 1553 remote terminal test is written in STOL and executes on the ITOS workstation.
- The 1553 bus controller residing in the test harness is instrumented to capture all of the time stamps of 1553 telemetry responses it receives and write them to a local file.
- The person running the tests sees a problem in a response coming too soon and copies the file with the time stamps on the test harness to a thumb drive and then emails the file to the developer of the remote terminal task.
- The developer has a few questions and emails the tester and asks where the STOL test script log file is on the ITOS workstation.
- The test is run a few more times, each producing more time stamp files and more log files to keep track of.
- A couple of them get posted to our bug tracking server; others reside only in emails.

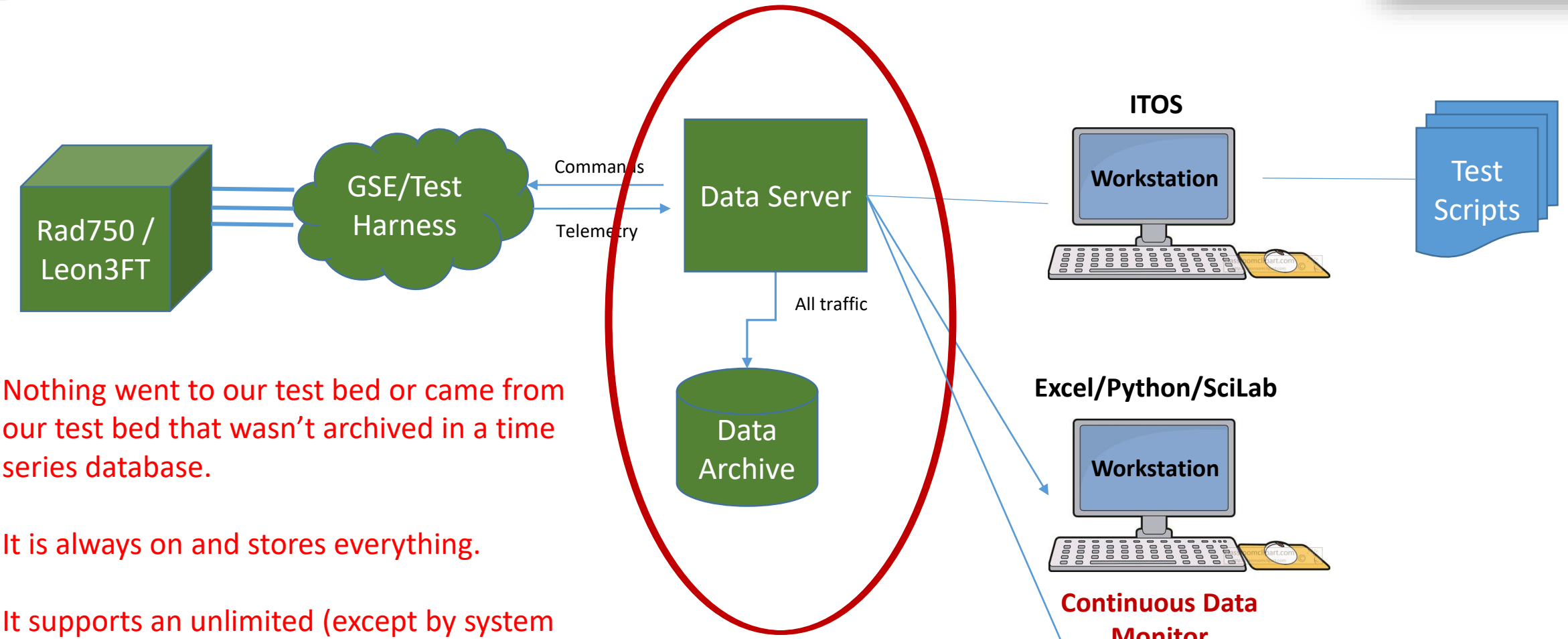


Lesson #1: Data Centric Test Bed



- Nothing went to our test bed or came from our test bed that wasn't archived in a time series database.
- It is always on and stores everything.
- It supports an unlimited (except by system resources) number of connections to either receive a real-time stream, receive the results of a query, or connect as a data source.

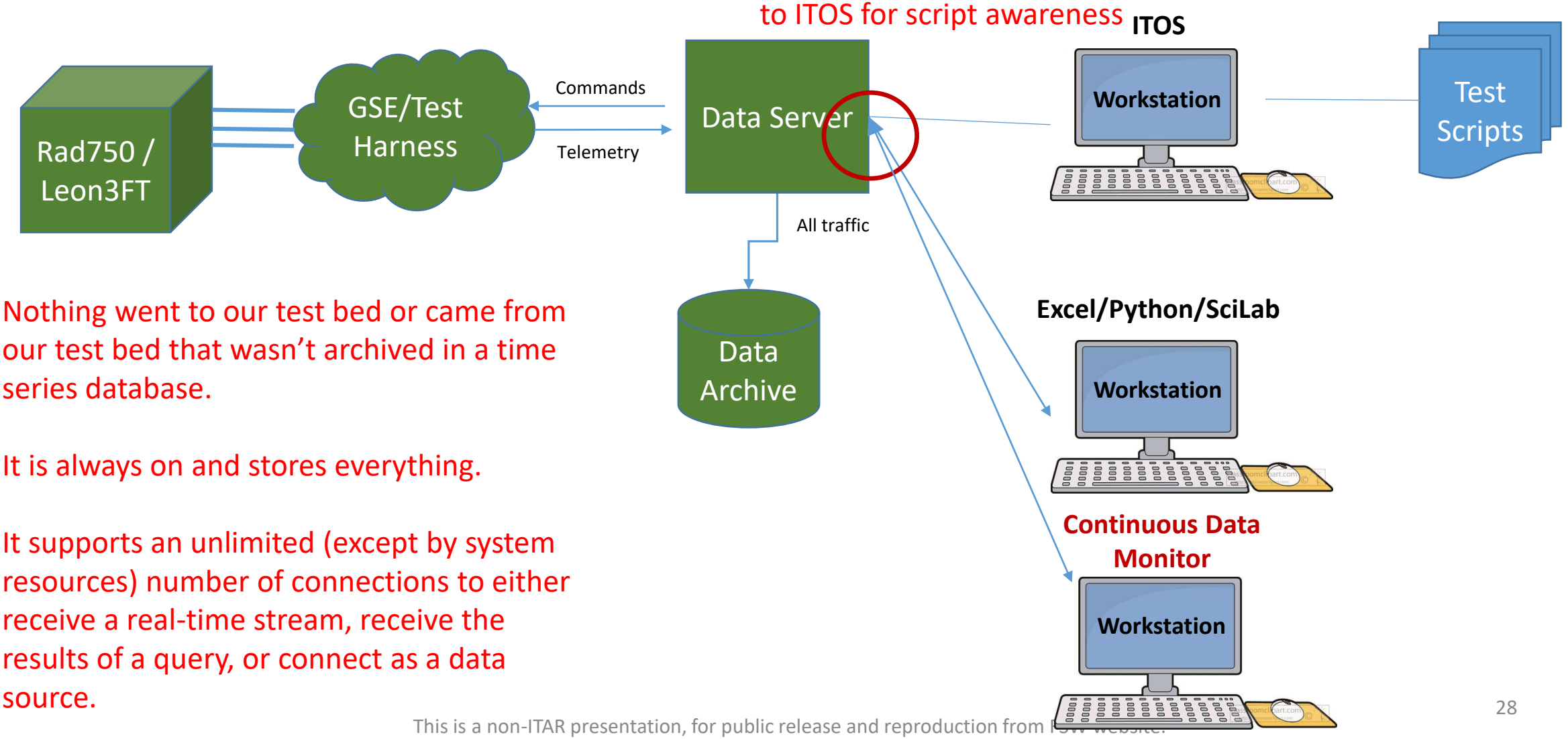
Lesson #1: Data Centric Test Bed



- Nothing went to our test bed or came from our test bed that wasn't archived in a time series database.
- It is always on and stores everything.
- It supports an unlimited (except by system resources) number of connections to either receive a real-time stream, receive the results of a query, or connect as a data source.

Lesson #1: Data Centric Test Bed

- Data from continuous data monitoring/processing can be fed back to data server and distributed back to ITOS for script awareness

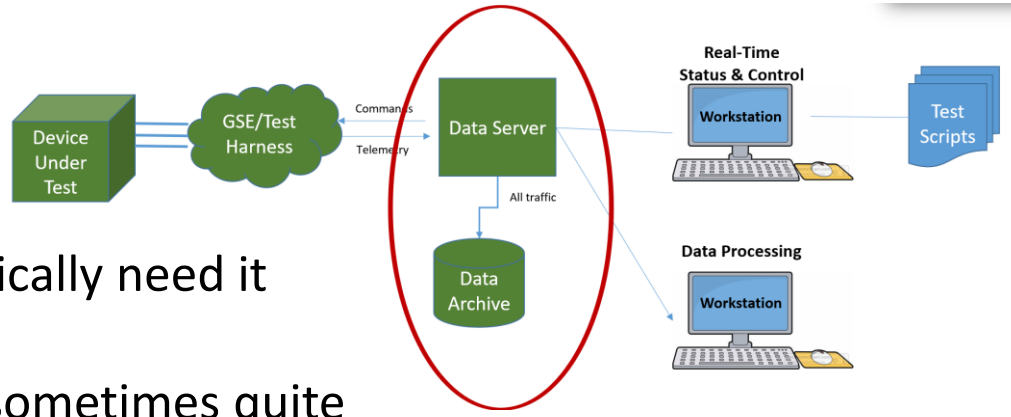


- Nothing went to our test bed or came from our test bed that wasn't archived in a time series database.
- It is always on and stores everything.
- It supports an unlimited (except by system resources) number of connections to either receive a real-time stream, receive the results of a query, or connect as a data source.

Lesson #1: Data Centric Test Bed

What this allowed us to do:

- Instead of haphazard local files holding instrumented data… create packet and telemeter it; it will then forever be stored and available for analysis later – even if the test doesn't specifically need it
- **Referencing a time range uniquely identifies data**
  - We went from trying to manage and version binary files (sometimes quite large) to using a simple timestamp string like YYYY:DOY:HH:MM:SS
  - Timestamp strings were very easy to attach to bug tickets and to communicate to other team members
  - The timestamp was a key to finding ***all*** the data at the time specified, not just the data that was locally captured
- **Tools can now be developed around the data server**
  - Continuous monitor tools can be written and attached to the real-time stream
  - Test scripts can autogenerate "tickets" that have their start and stop time which can then be used to kick off automated pulls and analysis of collected data.



ATLAS Example: Files to Time Stamps

We had roughly 8,000 algorithm tests that were run against the flight software, which in total produced about 4TB of results. We were managing each of these test data result sets independently and manually. The test itself was required to save off the data in a self-describing directory and the test engineers were responsible for delivering the data to the science team. As the data was copied to various computers it became difficult to identify which was the "official" data set. Also, if issues were discovered, the data set was too large to easily transfer for quick analysis and manual intervention was needed to isolate the offending data.

With the data server architecture we changed from identifying the test data results as a directory of data to identifying it as a start time and stop time (e.g. "20160602140532" represents the June 2, 2016 at 2:05 p.m. and 32 seconds). A customer request for test results became a data archive query. The official version of the test results was the data stored on our data server.

The biggest impact of this change came with how we on the flight software development team began to identify anomalies in our bug tracking system. Anytime an issue was found, the issue report ended up specifying the exact time stamp of the problem. From then on, any engineer working on the problem had only to issue a simple data server query to pull just the data they needed to analyze the problem.

ATLAS Example: Continuous Testing

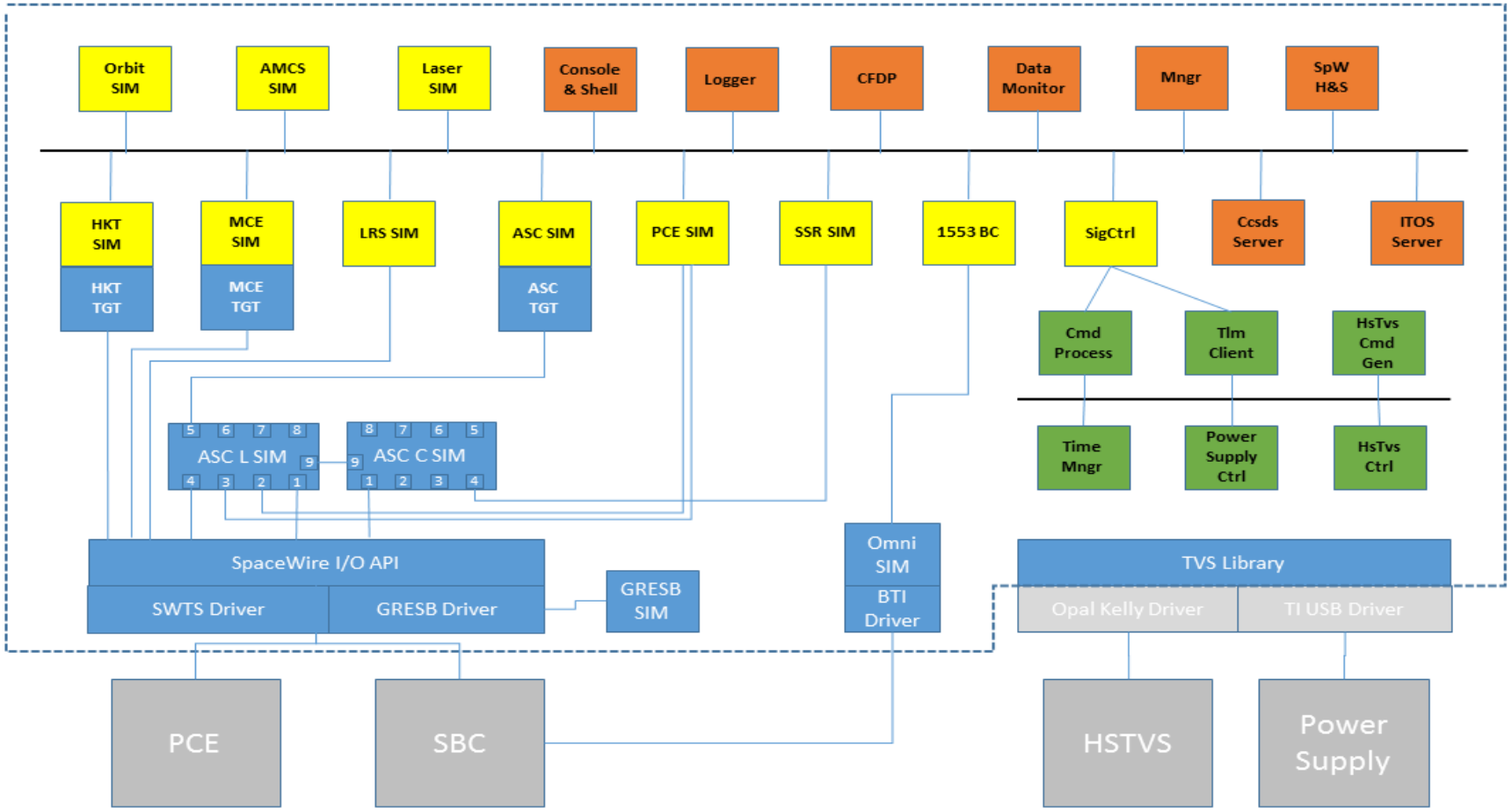
As a result of the data server always being on, continuously collecting everything, we realized that we could utilize time on the string that wasn't being directly used for a test and use it for continuous testing.

Therefore, we developed a dynamic orbit simulation which simulated the environment the flight software would operate in as if it were in orbit at the current wall-clock time. Then with some simple monitors and baseline configuration scripts we started leaving the test string "operational" any time it wasn't being used for a specific test.

As the project ramped down and the string was less utilized we experienced a 14 day period of time in which the string was left in "continuous" mode. On day 12 of this period an anomaly occurred which our test system caught. All data was recorded and we had everything needed to resolve the anomaly.

Lesson #2: Desktop Simulation Environment

We started building a highly capable desktop simulation environment from day one.



Lesson #2: Desktop Simulation Environment

- The entire development and verification string had a high fidelity port to the Linux desktop.

- This included flight code, simulations, and GSE. Code was written in a portable way ***from day one***
  - Immune to big endian vs. little endian processors
  - Immune to 32-bit vs 64 bit processors
  - No direct memory access (registers accessed through APIs)
  - Hardware drivers properly layered in the code

- The port of the flight code went down as low as possible. Code was refactored until the abstraction layers needed to get it to run on Linux were as small as possible.

- All hardware GSE had corresponding software simulations so that the test bed code that simulated things like the orbit and the reflectivity of the earth could run in Linux without the physical hardware being present.

- We fostered a plug and play string architecture. Our design goal was that any hardware should be swappable with a simulation, and vice versa. This allowed for hybrid setups which we used extensively during integration with real hardware.

- We developed an in-house marshalling solution (which we called htonp) that auto generated code from preprocessed header files and supplies packet conversions to network byte order for all transactions over a bus or network.

Lesson #2: Desktop Simulation Environment

Advantages to the desktop simulation:

- Developing in Linux with gdb, valgrind, profilers, and other desktop tools was much easier than in the embedded environment.
- Everyone has a full string available to them on their desktop for development of their code and test procedures
- Marshalling allowed for a single database definition and allowed us to use the flight ground system unchanged when running flight test procedures.

How does this apply to DevOps:

- Just like people are an expensive resource, so are physical hardware strings… minimizing their use for development maximizes their availability for testing
- The more you can exercise behaviors of the code the more stable and reliable the code will be… we never replaced official testing with testing on the simulator, but the simulator testing exposed bugs in the code early in development that would have been hard to find on the embedded system.

ATLAS Example: Plug and Play

Early Algorithm Development: Using the desktop Linux simulation of our code, we took one of our strings and teamed up with the optical team and supported the optical lab they used to develop the beam steering mechanism. They could have done everything with MATLAB and a computer, but we wanted to integrate early with them, so they used both the flight code running on Linux and then on the flight code running on the Rad750 with everything but the optical components simulated by our simulator as a part of their optical development system.

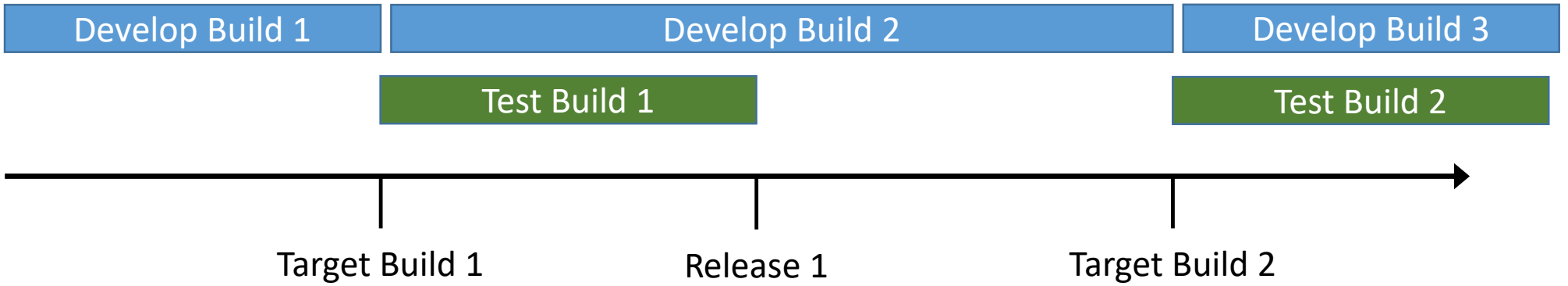
Hardware Troubleshooting: We troubleshot one of the hardware cards in the main electronics box by interfacing our flight code to the board and replacing just that cards simulation with the real card.

Early Science Testing: We supported science testing with the main electronics box by running just our spacecraft simulation and allowing the rest of the instrument to be the real hardware.

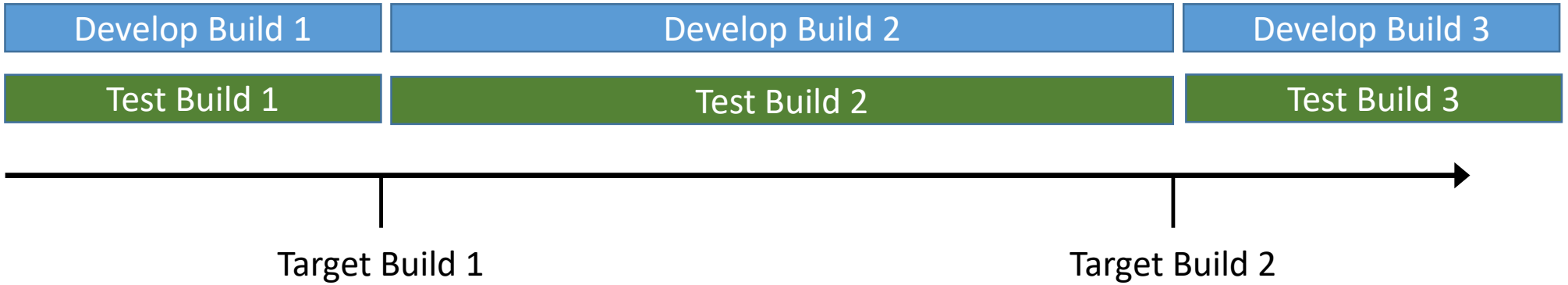
Board Level Thermal Cycling: We performed thermal cycling on the flight signal processing boards while the full flight software was running with no changes to the flight code. The C&DH (which usually runs on the Rad750), as well as the rest of the instrument was simulated in our desktop Linux simulation.

Lesson #3: Testing to Trunk

Old Way

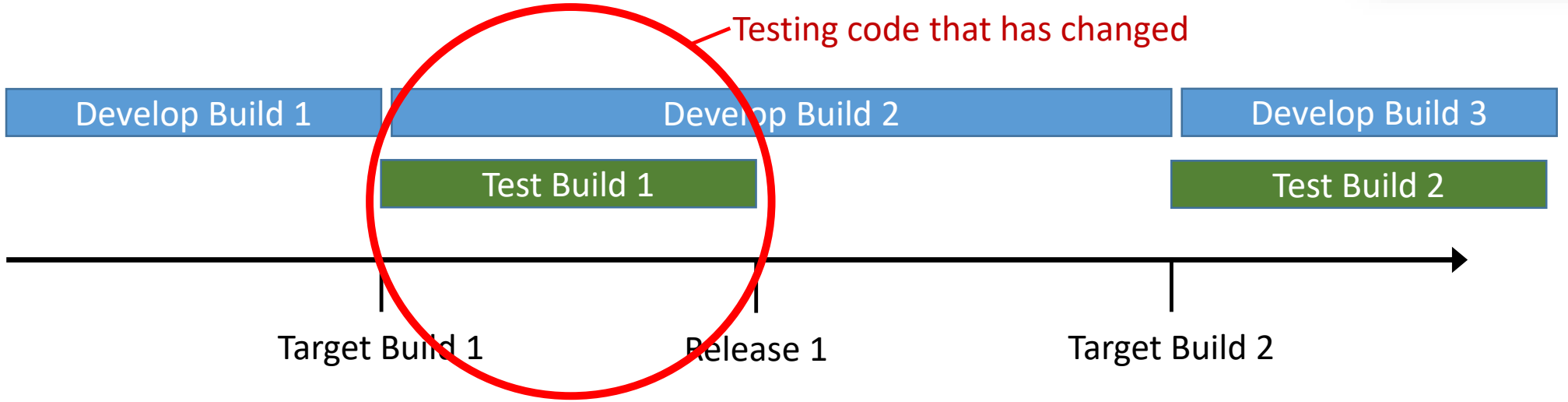


New Way

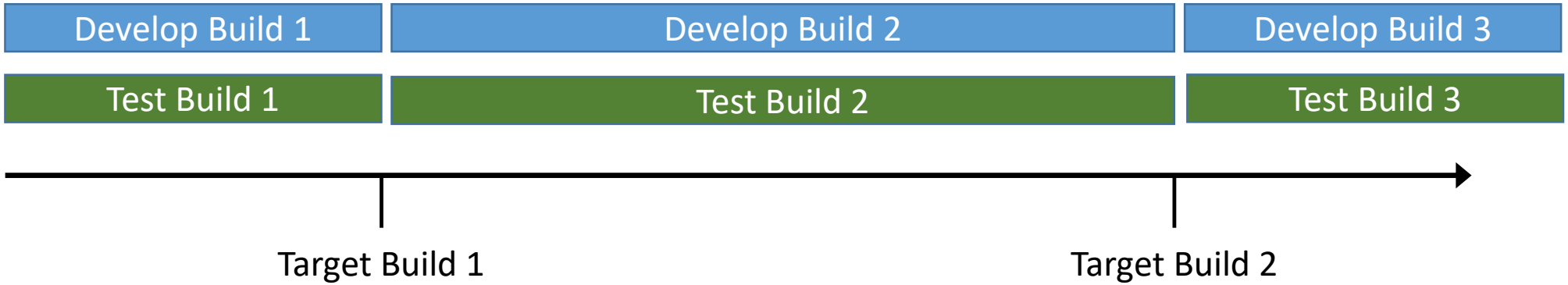


Lesson #3: Testing to Trunk

Old Way



New Way



Lesson #3: Testing to Trunk

Workflow:

- We still maintained independent testers, but it was not a separate team – there was just the software team that included both developers and testers.
- The software team had only one schedule – in order to release a build, both the code and the testing had to be done. As a result flight software and test procedures/infrastructure was developed together as a single product.
- Every day a tester would checkout the trunk and make sure any testing they did was integrated with the latest code.
- Testers and developers worked together from the beginning. Requirement and design arguments that usually happen well after the code is written and integrated, were happening right away. Issues were often found within 24 hours of being introduced.

Lesson #3: Testing to Trunk

Challenges:

- The technical demands/skill set of the tester is significantly greater. Since everything is changing all the time, they need to fully understand the system at both the development level and the system level.
- When coordination between the development and test effort break down, there is greater inefficiency in the test effort as it “chases” the development effort.
- There are inefficiencies when a piece of code changes multiple times due to developer instigated changes. Tester may drive the changes in which case testing to trunk is good as it speeds this cycle up.

Benefits:

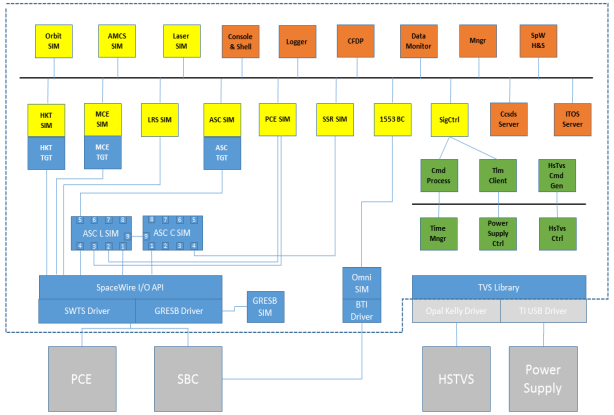
- Every line of code written has had the maximum amount of test time possible – since it started being tested after the first day it was written.
- Collaboration/coordination between development and test team greatly improved.

Lesson #4: Owning the GSE

The development of the real-time simulation of all interfaces, components, and environments needed by the flight software for development and verification was done by our flight software team, and was an effort equal in scope to the flight development.

The test bed development greatly exceeded the scope and complexity of the flight software development. The flight SLOC was around 90K where as the GSE SLOC was around 500K. But the significantly looser development practices for the GSE caused the cost of the efforts to be about the same.

The number one external dependency on schedule and budget of every software effort I've led has not been the requirements but the system needed to develop and test the code. We cannot afford to give control of that dependency to anyone else.



Lesson #5: STOL testing language

- We use the STOL scripting language which comes as a part of the ITOS ground system to write all of our build and system tests.

- STOL has been successfully used on flight missions at Goddard for generations.

- Programming languages manage complexity.
  - Is it expressive? How many lines of code does it take to express a desired behavior?
  - What is its approach to scoping variables?
  - Does support a layered architecture? How is lower level logic encapsulated in higher level blocks?
  - Does it support concurrency and distributed processing?
  - Is there community support?
  - Is it performant when it needs to be?

- The goal of automating potentially thousands of tests, executing them in a distributed system, managing failures and collecting/reporting results is not well supported by the STOL language.

- The ITOS implementation of the STOL language does not have functions.

Lesson #5: STOL testing language

- Here is a list of programming languages that are globally viewed as robust and expressive:
  - Python
  - Ruby
  - Lua
  - Go
  - Java
  - Swift
  - Scala
  - Clojure

- We should have chosen one.

Lesson #6: Manual data versioning

- The data server running in our lab stores all the data as opaque blobs of CCSDS packets.

- When we go back to analyze past data the question becomes: what was the format of the packet at that time. We managed this manually – i.e. you look up the VDDs, find the version that mapped to that time period, and do a pull of the database from SVN at that time.

- This made it impractical to go back in time… the result was that we rarely go back in time.

Lesson #6: Manual data versioning

- Other systems I've used run a database with time indexed meta-data. So tools can query a database with a time period and get back the packet definitions they need to process the data.

- Or we could have stored version information in the data server itself along side the data we were collecting.

Lesson #7: The plight of timeouts in command verification

- If you have a false alarm rate of 1 out of a 1000 runs and you are running a few tests a day manually, then false alarms are a small problem.

- If you have the same false alarm rate, but are running a 1000 tests a day, then you are investigating false alarms every single day.

- The single largest source of false alarms in our test system was a telemetry verification timing out. The root cause was either a bug in the test procedure where it got “out of synch” or a network delay that was unexpectedly long.

Lesson #7: The plight of timeouts in command verification

- Later in the test effort we implemented a scheme that attached to our full rate data and verified all commands through a command echo packet. It was significantly more reliable – yet our flight software design did not support the command echo packet being sent in nominal flight modes.

- Question is – are the reasons we use the command accept/reject counter still valid for the mission we are working on. For ATLAS, the answer was no – we could have used a different scheme – we just didn't think about it until much too late.

Lesson #8: The behemoth test procedure

- On ATLAS we had 556 test procedures, making up 262,500 lines (includes comments and blank lines)
- A lot of our build test scripts were over a 1,000 lines; our largest was over 10,000 lines.
- They were hard to review, hard to fully understand, hard to debug
- If a change was made to a large procedure, it could have a ripple effect
- We had a lot of cases where a large procedure had dependencies inside and outside of the test that we were not aware of.
- Getting ***all of these large test procedures*** that tested dozens of requirements each and took sometimes hours to run themselves, ***to work all the time*** without hiccups, was challenging.

Lesson #8: The behemoth test procedure

- Later in the project we had to implement a large set of algorithm tests (~8000 tests mostly made up of permutations on about 20 underlying test cases).

- From the very beginning the test engineer in charge developed a test framework that read in small test descriptions and executed each test. The framework was responsible for:
  - Test setup (making sure things were in a known original state)
  - Test execution
  - Test reporting

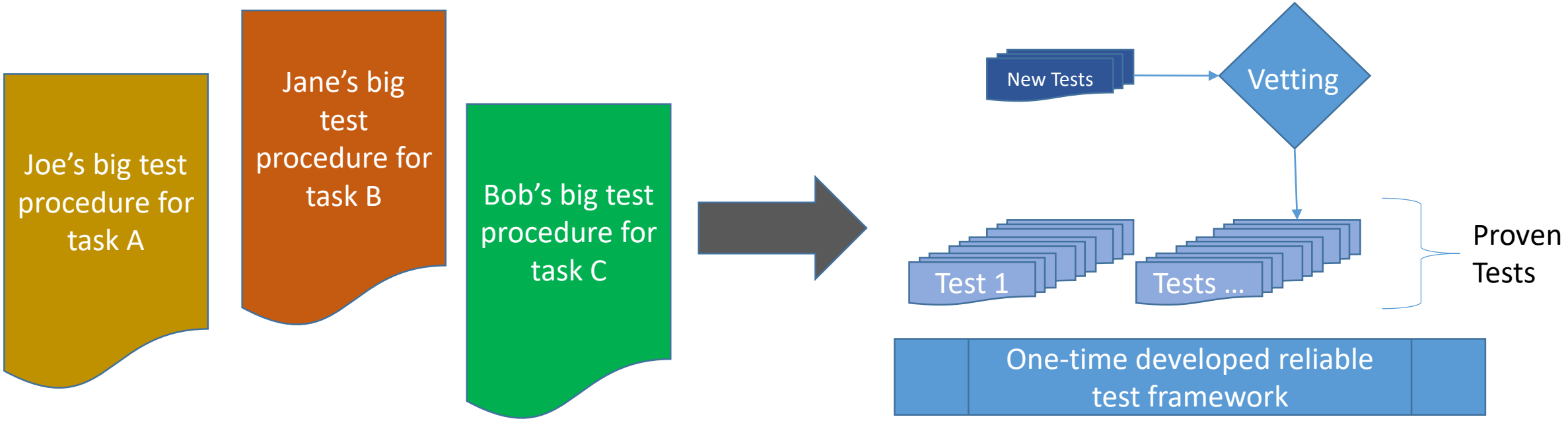
- We were able to run thousands of these tests all the time with no hiccups.

- When a test asserted itself as failing, we could drive down right to that specific test, understand immediately what we were looking at and resolve the problem there.

Lesson #8: The behemoth test procedure

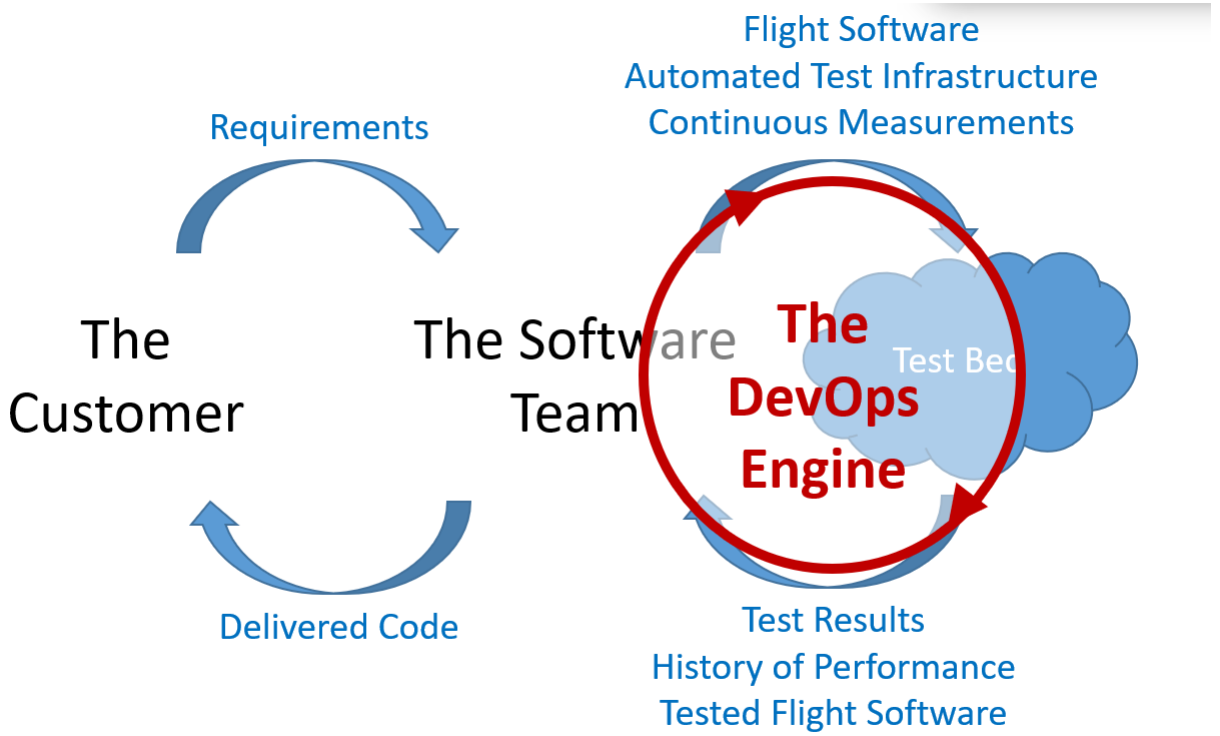
- This is a solved problem:
  - COSMOS has the Test Runner
  - cFS has a branch that implements a Lua build test framework
  - Unit test frameworks abound that implement this type of functionality

- Managing the test scripts goes from a technical problem of understanding the intricacies of each of the test procedures to a management problem of maintaining the list of “good” test blocks that can safely be inserted into the build test list.



Lessons Learned from a DevOps Approach

- **Invest in the engine that produces the product**
- **Design it so that it can be automated** (computers run tests, not humans)
- **Design it so that it can be measured** (maximize the amount of time each behavior in the code has been measured)
- **Good things**: a data centric test bed, a desktop simulation environment, testing to trunk, owning the GSE
- **Bad things**: the STOL testing language, manual data versioning, ambiguous command verification, the behemoth test procedure.



Acronym List

ATLAS – Advanced Topographic Laser Altimeter System
CRC – Cyclic Redundancy Check
EEPROM – Electronically Erasable Programmable Read Only Memory
GSE – Ground Support Equipment
PID – Proportional-Integral-Derivative
PROM – Programmable Read Only Memory
RPC – Remote Procedure Call
SLOC – Source Lines of Code