# Software Deployment in the Field (Technical Debt and Data-Ops)

Outline and Notes
Gary Granger
SEA Conference, 2015

**Abstract**

In the Earth Observing Laboratory at NCAR, we develop and deploy many different instruments, for platforms ranging from miniature sondes to radars and aircraft, to areas all over the world to observe all kinds of phenomena. Every field deployment is in fact a custom and very complicated configuration of systems, software, instruments, and data streams. Besides the stories about writing software in some interesting situations, field projects present some (exciting) challenges for software engineering. The concept of technical debt gives an interesting perspective on these challenges and on the costs incurred while under the pressure of a field deployment. Some ideas from dev-ops can be borrowed to help deal with these costs and to optimize data collection in general, in that the more automated we can make system deployment, data collection, and data processing in the field, the more we can mitigate the costs of manual customization, intervention, and "real-time development".

This talk will present a few approaches to dev-ops and "data-ops" for field deployments, and frame some questions about how to know which approaches to use when. In particular, the talk will describe a python package used for managing streams of data files for the Integrated Sounding System, and the ways this package has evolved to test a data-ops idea of continuous data processing, similar to continuous integration and release for software.

> *Nothing in this talk has been approved by EOL. I am not speaking for any other EOL software engineers, I am speaking merely from my own experience and perspective.*

## I. Introduction

When people ask me about my job at NCAR, I tell them I'm a programmer, and I develop and deploy software for weather instruments, and I enjoy getting out from behind my desk to support software and instruments in the field. In fact, I have had some fascinating and educational experiences away from my desk.

So this is about what I've learned, how it relates to some current software engineering concepts like technical debt and devops, and my work-in-progress to apply those concepts to one field platform which I support, the Integrated Sounding System. Much of that work-in-progress uses python, of course.

I will focus specifically on software engineering challenges for field project deployments, and I hope

that will be interesting, but I also expect that most of it will have familiar parallels in your own developments. I get excited about good clean code which solves real-world scientific problems in interesting ways, so I care about process and infrastructure which supports developing that kind of code, python or not.

First, I will set the stage with some stories of coding in the field. EOL field projects usually involve many instruments and platforms, deployed for weeks to months at a time, to observe some particular weather phenomenon of interest to the project's principal investigators.

# II. Stories from the Field (or, mistakes from my past)

## A. Story of Zebra at TOGA-COARE

One of my first field experiences at NCAR was for the TOGA-COARE project in 1993, still a large project by any measure, and I was working at the ops center in the Solomon Islands supporting a software system called Zebra.

TOGA-COARE slide

Before Jon Corbet became a Linux kernel developer and editor of LWN, he developed Zebra. Zebra provided visual integration of a variety of kinds of instrument data, and configuration for a set of typical visualizations, for both real-time and post-project, because every field project was different.

Zebra plot graphic

So here I was in Honiara, on the island of Guadalcanal, supporting scientific software, I had this sense of adventure and excitement for field developments, seeing some of my code in use and meeting scientists' needs in real-time.

Well, I was young and optimistic, and I saw this shortcoming in the Zebra datastore that I thought I'd fix, right there, halfway around the world in the second floor of the malaria research center, the day before I was leaving to go wander around New Zealand for two weeks completely out of touch. I knew I was taking a risk because I went ahead and made a copy of the executable before I recompiled it, and I thought if there was a problem someone would probably be able to find the backup. Probably I gave the backup copy some clear and precise extension like *.pre-garys-datastore-fix*. And then I left, thinking how totally cool it was to write code in the field.

## B. Other field stories

I have since had many other interesting and fun stories of developing in the field.

NOAA P3 circling OKC at low altitude, feeling sick and trying to test and debug software.

GAUS truck

VORTEX driving through the plains, writing code on a laptop sitting shotgun in the MGAUS pickup.

Because you don't *really* know how reliable your cellular broadband connection will be and how to optimize the transmission of sounding data in real-time from a dead spot in the middle of Kansas.

CLIMODE images

CLIMODE: sitting at a computer and my chair sliding backwards away from the computer when the ship rolled.

Picture of U-Haul trailer at METCRAX

Sitting inside seatainers and even U-haul trailers. Coding can happen anywhere.

## C. Integrated Sounding System

Nowadays, one of the field platforms I support is called the ISS, the Integrated Sounding System.

It is a good case study for all the field development issues I'm talking about: it is a highly configurable instrument network and system of software that must be deployed in all kinds of environments with all kinds of instruments.

Slide of ISS on land at Diego

ISS on a trailer at T-REX

ISS on a ship in DYNAMO

And thanks to our scientist Bill Brown for the cool diagrams.

# III. Technical Debt in the Field

## A. The Nature of Software Development in the Field

A field deployment is driven by the observations the scientists need to record with the instruments they can deploy. So every deployment requires custom configuration and integration, and most of that configuration and integration happen--where else?--in software. And it seems to be inevitable in field deployments that some of the configuration and integration happens in the field, and software development will often happen in a time crunch and in some unusual environments.

Clearly the less development and configuration that must be done in the field the better, but assuming that some will always be unavoidable, I think it is useful to look at the software development done in the field in terms of technical debt.

## B. What is technical debt?

It turns out that in 1993, right around the time of the TOGA-COARE project, Ward Cunningham contributed an experience report to the 1992 OOPSLA conference about the development of some financial software, and the report describes a debt metaphor for software development.

> Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object- oriented or otherwise.
>
> Ward Cunningham

This metaphor came to be known as technical debt.

The idea has evolved and many people have expanded on it, and so Ward has since tried to clarify the metaphor as he intended it. His technical debt metaphor is not about what happens when you write bad code. From his perspective, even if you always write good code, you still take on debt because that code will need to be refactored in the future as your understanding of the problem evolves and expands.

From the perspective of field deployments, I'll add that the code also needs to be maintained, configured, and deployed, for each and every successive field deployment. **In other words, writing the code is never the end, writing the code always implies taking on debt.** Code isn't an investment that sits in the bank and keeps giving you returns, it's the opposite. It's a mortgage on which you keep paying interest or else it becomes useless.

The kind of debt that Ward describes is not necessarily bad debt. And in fact you can read lots of articles justifying cases where you should take on debt. Martin Fowler gives the example of choosing between two solutions to a problem, call them the quick-and-dirty solution and the slow-and-clean solution. If you really need a quick solution, then you choose quick-and-dirty, but you accept a larger amount of technical debt, accepting that someday you'll have to refactor the dirty code into something cleaner.

Technical debt is what we have when we borrow against our future development time.

So I find the quick-and-dirty versus slow-and-clean dilemma to be especially relevant to field projects, but there are several pressures I'd like to mention which push us to take on debt, usually without even thinking of it as debt. And naturally all programmers have probably experienced analogous pressures in their developments.

## C. Pressures to take on debt

Here are some of the pressures we have to take on technical debt for a field project:

### i. Mission Creep

Or, since we're already going to be there, or incremental cost, or once-in-a-lifetime opportunity.

There is pressure to make the most of an expensive deployment: you send equipment, instruments, and people half way around the world for 1-2 months ("some limited window, and within that window to study some limited number of occurrences of whatever phenomenon you're there to observe") to observe a specific phenomenon -- you do everything you can to get it right. You may never get another

chance to have all these instruments in this place at this time to observe these conditions.

### ii. Live Testing

Often software changes are just much more convenient to test during a field project. Sometimes the software was written specifically for that field project and that combination of instruments and data, so that's when the code gets really tested. **The field project is our best integration test.** Therefore fixes have to happen in the field, and the time crunch can dictate the kind of fixes we can make.

### iii. User Interaction

Obviously the field deployment is usually the first time all the instruments and all the data are online together recording real observations, so of course that's when the scientists and investigators start to learn how well the deployment is working and what other features they wish they had.

### iv. Thrill and excitement

I'll be honest, it's much more exciting for me to make changes in the field, because that's challenging and that's where I see immediate gratification. I deploy a change and I get to see it in action right away, it immediately starts helping people.

SABL slide

I have lots of examples of this in my own experience, but here's one example from working on the SABL LIDAR instrument deployed on our C130 aircraft. I was building and deploying embedded vxworks code while flying over the Great Lakes, testing the laser pointing. It was live testing, the instrument was on the plane, in the air, receiving the real live navigation data feed, vibrating and bouncing over the Great Lakes. And I admit it was fun and exciting, and I wasn't thinking twice about changing the code and recompiling during flight. I was having so much fun I took a selfie at 5,000 feet, and that was before selfie was even a word. Don't I look like I was having fun?

Was I thinking then about taking on debt? No, but I should have been. Any code changes made in the air would need to be evaluated later, refactored, and tested. Because I couldn't just ask the plane to fly the next time I wanted to verify my changes still worked.

## D. Technical Debt for Data

While I'm at it, technical debt is also incurred for data. The emphasis for the field project is to record as much as possible, take measurements in the field with as many instruments as you can field and fund, with the intention of later on, after the project, dealing with the format changes and extra research and processing that will be required to reconcile and finally publish those data.

## E. System configuration in the field is also technical debt

Technical debt applies to system administration too. For example, we tweak system configurations to add services and capabilities in the field: firewall and routing, printers and other hardware, third-party

software installations, desktop configurations, user accounts. We can quickly do it manually in the field, but then the next time we need that same capability we'll have to repeat the manual configuration, after trying to remember how we did it last time. The longer-term solution is to write it up in a script which can get its parameters from a central configuration.

So like software, system configuration changes can immediately incur some amount of debt.

### F. Other sources of technical debt in the field

It can be hard to justify spending time on normal software development practices: no time to write tests, or track changes, or write documentation, or log issues in an issue tracker.

Technical debt in a field project is inevitable: sometimes you must deploy the expedient fix during the project to keep it going and put off all the checks until later. It means adopting a feature now and assuming it can be supported for all future field deployments.

The features are not just software features: it includes all the system features accumulated from all the field projects up to now.

Users (and developers also) expect to use those features again. They have a long memory. Also, it is very likely a different user will come up with the same need but with a slightly different deployment configuration.

Accepting instruments and software developed outside EOL is like immediately taking on a huge debt with unknown risk.

These are examples of choices we must make for every field project, and neither choice is right or wrong. Sometimes it is impractical to document an issue. Sometimes a feature really doesn't seem like it would ever be needed again. **The point is that we must make it a conscious choice, and understand the debt implications for whatever choice we make.**

## IV. Reducing debt for field deployments

The basic idea is to let the software do as much of the customization and configuration work as possible, and borrow ideas from agile software development practices and devops.

### A. Design to Deploy

**Software is not working until it is deployable**. It is not enough to compile the program and verify that it does what is needed, it must be automatically and consistently deployable (and even updatable), without requiring repeated manual configuration.

For the ISS, some of the software is third-party processing software, or scripts that scientists have written, so I've been writing python wrappers to those programs to adapt them to the ISS environment. The python code can collect the configuration parameters and apply them to the command line or environment, whatever the program needs.

## B. Test Deployments

So how do we know when a software system has been deployed successfully? **Unit tests should be deployed with the software and should run successfully on the field system.** But we don't need to stop at unit tests. **We can also check that all the dependencies exist on the system, like outside tools or libraries, and that those tools work, and a sane configuration exists.**

As an example, some of the ISS data processing depends on IDL, and so the ISS system configuration script makes sure IDL is installed. (It should also make sure IDL has a license, but I haven't done that yet.) The Twelve-Factor App suggests not relying on any system dependencies, rolling all external tools into the deployment system. In practice, I get around that on linux by declaring which RPMs and external software I need and automatically installing those when configuring a system. Either way, it's worth adding some automatic mechanisms to confirm a deployment. If you run something manually to make sure an install worked, then script it instead.

## C. Consolidate and Deploy Configuration

Field software should differ as little as possible across deployments. Instead, customization should be realized with configuration parameters, and access should be through a single interface. Essentially this means associating configurations with the field project rather than spreading it out across the software. Some settings need to be consistent across the deployed software, and some might be specific to one program but change across projects. Either way the goal is to deploy a single and consistent configuration across the range of software tools used in a field project, rather than deploying separate configurations for each tool. The configuration includes metadata and context information also, like the name of the project, host names, site names, and so on. Ideally such metadata are set in one place and one place only, since that way it cannot get out of sync.

### i. Avoid redundant configuration

If different parts of the system or software all need to know the name of the current field project, or the temperature range limits, then they should all get it from the same place.

**On the ISS, the configuration is stored in a text file in JSON format, and a python script can be used to set and query the parameters in that file. Both python code and bash scripts get their configuration settings from one place.**

### ii. Keep the Configuration in Revision Control

Deploying the configuration also means keeping the configuration consistent across sites and hosts, not just on one system. For the ISS, the configuration is under revision control. When a project configuration changes, the changes can be pulled at every site with a subversion update. One advantage to storing configuration settings in text formats like JSON or XML is that it is easy to manage changes and see differences in revision control systems.

Configuration includes system administration settings as well as software and metadata configuration.

When we deploy a network server, it always has custom iptables rules and DHCP and custom user accounts. Those are part of the configuration that should be deployable (and verifiable) automatically.

There are other benefits to keeping configurations in revision control, even if it's for a one-time project and will only be used on a single server. When configuration is in revision control, I can see a log of what's changed on the server and when. I can investigate the current settings even when I'm away from the site or the site is inaccessible, ie, the Internet link is down or the site is powered down. I can also deploy that same configuration to a test server and investigate a problem without having to change the production system.

### iii. Beware overlooked configuration sources

#### Crontab files

If software depends on an entry in crontab, then that is part of its configuration and it should install and manage those entries itself, as part of realizing its configuration. It should not be done manually.

#### BIOS settings

When we deployed a PC to Greeley for a 3-month project over the winter, I forgot to set the AC recovery option so the PC would turn back on after a power loss. Needless to say I was not happy with myself when the site had a power outage and the PC did not recover. I think my coworker that drove out there to push the power button was even less happy with me. Since then I've looked for ways to automatically confirm the BIOS setting when configuring a system, but so far to no avail.

#### Routers, webcams, and other devices

Download and commit the router configurations. As soon as the network appears to be down and the site cannot be reached, you start wondering what settings you left on the router, and how quickly you could restore the configuration if you had to replace it with a new one bought off the shelf at the closest computer store.

## D. Apply devops to system configuration

Devops can help mitigate the technical debt from system configuration tweaks. If you have to tweak a system to accommodate a new field project configuration, then instead of doing it manually, add the tweak to a script or to a provisioning system like puppet, and make it dependent upon a configuration parameter.

ISS bash and python scripts can retrieve configuration parameters and then configure the system accordingly. For example, set the network parameters, enable or disable ports and services, create data and log directories, install packages.

**First automate everything that can be automated, then automate everything else.**

**On the ISS data manager...**

System Configuration slide

Scripts ask questions of the operator and set configuration parameters in the JSON configuration, and then the scripts modify the system according to the configuration. I checkout the ISS source tree, install it, and then run a script which steps through all of the checks and modifications needed to turn that computer into the ISS data manager for that site and that project. It's been very useful, but now it seems rather primitive. It would be nice to implement more of it in python and add a web interface for the configuration.

## E. Automate failure detection

More and more of our systems take advantage of nagios and other monitoring, so a software engineer gets an email or text message when a nagios check fails. Once you've coded up the detection of a problem, you can use it all the time, everywhere it makes sense. Otherwise you end up manually checking for problems during the project. That sounds to me like paying interest on a debt I didn't need to accept. Automated checks save time in the field and make it less risky to do change code and system configurations in the field.

The ISS has several python scripts which specifically check on important data streams and log files and report problems to nagios.

# V. Applying devops to data: data-ops

So I also mentioned that data can be a source of technical debt. We are constantly adding new instruments and new data streams. Recording new instruments for one project likely means recording that instrument for future projects. And it never ends with just recording the instrument data, because those data will need to be checked for quality and reformatted into publishable formats. The more of that that can be automated, the better.

Unlike some of the other tools we've heard about for doing interactive scientific research with data, this is all about making the data management non-interactive.

So "data ops" is for data what devops is for software: automating the data processing and consolidating the configuration as much as possible, so that **publishing a dataset is as turnkey as deploying a server**. Whatever transformations must happen to the data, whatever software must run and with whatever parameters, should all be automated. Like a Makefile for data: identify the dependencies, the transformations (like Makefile rules), the output formats and metadata (the targets). Then "run the build" and "run the tests" specified in the data Makefile. If the processing chain, software, configuration, or inputs change in any of the dependencies, then the affected targets get regenerated.

Unlike for software builds, data builds should include documentation in the metadata for how the results are generated, when, and reasons for changes to the processing. All of that can be generated and updated more easily if the data processing configuration is consolidated into a "data ops" build system.

I have been trying to do something like this in a python data management framework written for the EOL Integrated Sounding System. The ISS is a good testing ground for this because it contains dozens

of disparate raw data streams which all feed into different derivations and processing software, and the outputs include artifacts like plots.

## A. Data-ops for the ISS

iss_catalog slide

The ISS uses a python application as a test implementation for this idea of data-ops, called the iss_catalog. The catalog script scans data files and builds up a database of metadata for each file. Different Python subclasses in the application can parse the different formats to extract sample timing and also data values. A processing engine builds lists of transactions which need to run on each category of datafile. A check engine runs checks across the data files and reports problems. Since the particular data file categories and timing differ from project to project, that information comes from the central JSON configuration file for each site.

It's not unusual for the ISS to generate a few hundred thousand data files across 50 categories of data file, so the catalog really helps bring down the incremental cost of adding new categories to manage and check.

Python Helps

Python has been a great platform for this implementation because it has been easy to interface with the rest of the system and all the functionality available in built-in and add-on packages. Metadata are collected and stored in a sqlite database, data files with custom formats can be easily parsed, netcdf files are read with the scipy netcdf interface, there is a module to read the JSON configuration files, the subprocess module allows for running and managing non-python data processing programs with almost seamless integration.

# VI. Conclusion

The technical debt metaphor has been useful to me in how I think about my choices for software developments. I would encourage anyone to learn more about it. It provides a useful analogy and (I think) good justification for continually improving your software development practices. Ward Cunningham even made the point that **technical debt fits nicely with the agile development processes, since ideally debt gets paid off quickly in successive iterations and not put off.**

Further, maybe, just maybe, **this is a useful concept to introduce to our stakeholders and managers.** When we as developers are faced with taking on technical debt, probably we aren't the only ones who are affected by that decision. Frankly, in EOL I think we can correlate technical debt with stress. When technical debt comes due, it's never at a good time. So we help ourselves out in the long run by encouraging development practices which cut down on technical debt.

Finally, back to the story of my epic mistake during TOGA-COARE. I took on technical debt by making a software change in the field without evaluating who would pay the debt, how, or when. And isn't that the worst kind of debt, the debt we accept but which someone else will have to pay later? I

left the Honiara ops center and went traveling through New Zealand, and my fellow programmer Michelle took the next shift and she had to pay off my debt. I finally called her a week later from Auckland, because I had a sinking feeling that maybe something hadn't gone quite right, and she confirmed my fears. The datastore had mysteriously stopped working, and she was left trying to figure out why. I hope she was able to forgive me for that. And for all the other programmers who have had to pay off debt that I should not have taken on, especially during field projects, I hope they can forgive me too. I'll keep trying to do something about it.

### References:


http://en.wikipedia.org/wiki/Technical_debt
http://c2.com/doc/oopsla92.html
http://c2.com/cgi/wiki?WardExplainsDebtMetaphor
http://martinfowler.com/bliki/TechnicalDebt.html
http://12factor.net/