Accelerating the Path from Dev to DevOps

DINAH MCNUTT

Dinah McNutt is a release engineer at Google. She has a master's degree in mechanical engineering from MIT and has worked in the fields of system administration and release engineering for more than 25 years. She's written articles for numerous publications and has spoken at technical conferences (including chairing LISA VIII). mcnutt@google.com

My first lesson in release engineering occurred more than 20 years ago. I was working for a start-up company, and we discovered that we could not reproduce the build we had shipped to customers. This meant we could not send out a patch for this release and our only solution was to force all our customers to upgrade to the new version. I was not directly involved in the events that got us into this situation, but I certainly learned from it.

I've spent most of my career in a system administration role at start-up companies and have learned a lot about software development and releasing products. Twelve years ago, I fell into a release engineer position when the company I was working for needed someone to do the work, and I discovered I loved it. All the skills that made me a good system administrator (problem solving, attention to detail, etc.) were directly applicable.

What Is Release Engineering?

Release engineering (or releng, pronounced "rel-eng" with a soft g) is like the old story of the blindfolded people and the elephant. You may get a different answer depending on whom you ask. But, because this is my article, I get to describe my elephant.

In a perfect world, a release process looks like:

- Compile
- Test
- Package
- Release

A real release process looks more like what is shown in Figure 1.

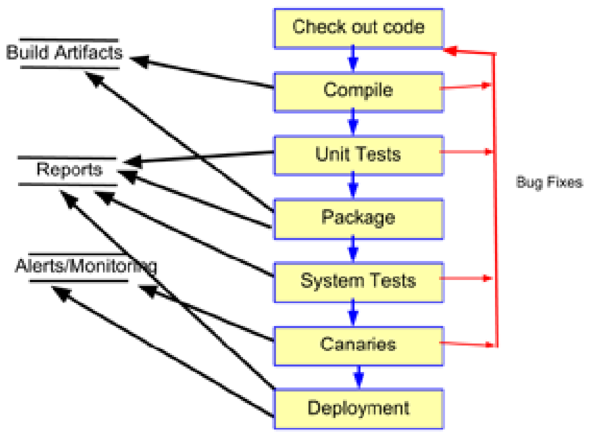

Figure 1: A real world release process

I'm not going to go into the details of Figure 1 because the point is to show that most release processes are complicated. However, here are some terms from the figure that you might not be familiar with:

- *Build artifacts.* By-products of the release process (i.e., log files, test results, and packaged binaries). Basically, it's everything you want to save from the release process.
- *Canaries.* Testing new software on a small number of machines or with a small number of users.

As the tagline to this article says, releng accelerates the path from development to DevOps by bringing order to the chaos shown in Figure 1. How do we do that?

Building Blocks

My eyes usually glaze over when I hear people talk about velocity, agility, delivery, auditing, etc. Those concepts are the attributes and results of good releng practices but are not where I like to start when I talk about releng.

Instead, here are the things I care about:

- *Release engineering from the beginning*. Releng is often an afterthought. Companies should budget for releng resources at the beginning of the product cycle. It's cheaper to put good practices and process in place early rather than have to retrofit them later. It is essential that the developers and release engineers (also called releng) work together. The releng need to understand the intention of how the code should be built and deployed. The developers shouldn't build and "throw the results over the fence" to be handled later by the releng. It's OK to outsource the implementation of your releng processes, but keep the ownership and design in-house.
- *Source code management*. Everything needs to be checked into a source code repository. It's not just about code. Configuration files, build scripts, installation scripts, and anything that can be edited and versioned should be in your SCM. You need to have branching/merging strategies and choose an SCM system that makes these tasks easy. I personally think you should have different strategies for ASCII and non-ASCII files (like binaries). I am not a fan of storing binaries with source code, but I do think it is reasonable to have separate repositories for those types of files. (This is one of those topics in which even members on the same releng team do not agree!)
- *Build configuration files*. The releng should work closely with the developers to create configuration files for compiling, assembling, and building that are consistent and forward thinking (e.g., portable and low-maintenance). Do they support multiple architectures? Do you have to edit hundreds of files if you need to change compile flags? Most developers hate dealing with build configuration files, but a releng can make their lives easier by taking the lead in this task.
- *Automated build system.* You need to be able to build quickly and on-demand. The build process needs to be fully automated and do things like run tests, packaging, and even deployment. Your build system should support continuous and periodic (e.g., nightly) builds. A continuous build is usually triggered by code submissions. Frequent builds can reduce costs through early identification (and correction) of bugs.
- *Identification mechanism*. There should be a build ID that uniquely identifies what is contained in a package or product, and each build artifact needs to be tagged with this build ID.
- *Packaging*. Use a package manager. (As I have said repeatedly, tar is not a package manager.) You have to plan ahead for upgrades, handling multiple architectures, dependencies, uninstalls, versioning, etc. The metadata associated with a package should allow you to determine how the binaries were built and correlate the versions to the original source code in the source code repository.
- *Reporting/Auditing*. What was built when? Were there any failures or warnings? What versions of the products are running on your servers? Logs, logs, and more logs. (We like logs.)
- *Best practices*. What compile flags should you use? How are you versioning your binaries so you can identify them? Are you using a consistent package layout? Can you enforce policies and procedures?
- *Control of the build environment.* Do your tools allow you to put policies in place to ensure consistency? If two people attempt the same build, do they get identical results? Do you build from scratch each time or support incremental builds? How do you configure your build environments so you can migrate your tool base to newer versions yet still be able to support and build older versions of your code?

I've described the building blocks of release engineering. Through effective use of these building blocks, you can

- Continuously deliver new products (e.g., high-velocity)
- Identify bugs early through automated builds and testing
- Understand dependencies and differences between different products
- Repeatedly create a specific version of a product
- Guarantee hermetic build processes
- Enforce policy and procedures (this is a hard one—you at least need to be aware of violations and exceptions)

Sub-disciplines within Releng

Releng is an evolving discipline. It's going to be exciting to see how it changes over the next few years. At many companies, releng is just one of several hats worn. At LISA '13, I held a Birds-of-a-Feather session on release engineering. Several people attended who have a dual role as system administrator and release engineer. Because I come from a system administration background, that makes perfect sense to me!

However, at a large company like Google, we are starting to see specialization within the releng team that is dictated by product area and personal preference. Here are the sub-disciplines I have identified:

- *Tools development.* Extending and customizing our proprietary build tools; developing stand-alone applications to provide reporting on everything from build status to statistics about build configuration files.
- *Audit compliance.* This is no one's favorite task, but the Sarbanes-Oxley Act of 2002 dictates that controls must be put into place for applications that handle financial information. The controls include (but are not limited to):
  - Verifying all code that is under scope for SOX has undergone a code review (separation of duties)
  - Verifying the person who writes the code must not also own the build and deployment processes (separation of duties)
  - Embedding a unique ID that can tie the binary to the build that produced it (version verification)
  - Using a package manager that supports signatures so the package can be signed by the user who built it (builder and version verification)

  Release engineers work with developers and internal auditors to ensure that appropriate controls and separation of duties are in place.
- *Metrics.* We have several projects that provide releng-related metrics (build frequency, test failures, deployment time, etc.). Some of these tools were developed by members of the releng team.
- *Automation and execution.* We have proprietary continuous-build tools, which are used to automate the release process. Release frequency varies widely (from hourly to yearly). Typically, customer-facing applications are released more frequently in order to get new features out as quickly as possible. Internal services are usually updated less frequently because infrastructure changes can be more expensive. However, with effort, release processes can be developed which support frequent, low-impact changes.
- *Consultation and support.* The releng team provides a suite of services to development teams, which range from consulting to complete automation and execution of the releases.
- *Source code repository management.* We have a dedicated team of software engineers and administrators who work on our source code management system, but many of the release engineers have in-depth knowledge of the system. We even have engineers who transferred from the source code repository team to a releng team!
- *Best practices.* This covers everything from compiler flags to build ID formats to which tasks are required to be executed during a build. Clear documentation makes it easy for development teams to focus on getting their projects set up and not have to make decisions about these things. It also gives us consistency in how our products are built and deployed.
- *Deployment.* Google has an army of Site Reliability Engineers (SREs) who are charged with deploying products and keeping google.com up and running. Many of the releng work closely with SREs to make sure we implement a release process that meets their requirements. I spend just as much time working with SREs as I do Software Engineers (SWEs). We develop strategies for canarying changes, pushing out new releases without interruption of services, and rolling back features that demonstrate problems.

What's Next?

Here is what I expect to see over the next few years in the field of release engineering:

- More vendors entering the space (particularly cloud-based solutions). Look for productization around open source software (e.g., Git) and tools that will offer end-to-end release engineering solutions. The latter will probably be achieved through partnerships between vendors.
- Fuzzy lines between configuration management and release engineering (my prediction is that they will evolve into a single discipline)
- Standards organizations—ISO standards for releasing highly reliable software, SOX compliance standards, etc.
- Industry-standard job ladders
- College curriculums
- Industry-accepted best practices
- Industry-accepted metrics

I am excited to be chairing the upcoming URES '14 (USENIX Release Engineering Summit). As we are starting to put the conference program together, we wanted to be able to easily explain what release engineering is and why it is important (and timely) for USENIX to sponsor a summit on this topic. I hope this article has been a good introduction to release engineering and that my personal experiences have been educational. May all your software come from reliable, reproducible processes!