The Challenges and Practices of Release Engineering

Diomidis Spinellis

**RELEASE-ENGINEERING TEAMS** are responsible for building and delivering software to the customer and for enabling its development on an industrial scale. This has always been a tall order. In modern software development, a number of factors make release engineering even more challenging.幸亏遵循了几项既定的实践让我们可以应对这些挑战，并以可靠、可信赖和高效的方式交付软件。

**Modern Challenges**

The first sign of trouble on the release-engineering front typically comes from the software’s scale. Software comprising hundreds of thousands to millions of code lines is increasingly common. This large size is typically further inflated through associated components and diverse configurations. Building, testing, and deploying such software can take a heavy toll on developers’ time and on computing resources.

Then comes the software’s scope. The multitude of available computing platforms, ranging from cell phones, to tablets, to desktops, to servers, to embedded software systems, gives us all wonderful choices and drives innovation. However, releasing software for many different devices can be a nightmare. The tailored builds increase complexity and gobble up CPU resources, and testing might require maintaining dedicated hardware. Release-engineering teams have to support cross-compilation tool chains, and many keep racks with diverse and exotic hardware devices as a testbed. As if this variability isn’t enough, consider that modern software is often deployed around the world to places with distinct languages, writing systems, fonts, customs, and regulations. These result in even more configuration and testing options.

Also, modern software typically depends on tens to hundreds of third-party components. This isn’t a matter of choice: many people believe that if your software doesn’t consist mostly of such components, you’re wasting resources by reinventing the wheel. Yet, managing these components is tricky. You need to combine stable, compatible versions, but you’re also forced to update components to address security vulnerabilities, hardware upgrades, or interface changes. You must set up reliable mechanisms to receive updates from suppliers, and you must integrate your changes with the third-party code. Finally, you must ensure legal compliance with all the software licenses of all the used components. Many of these concerns also apply to the tools...
you use to build and release your software.

A related problem is the management of long-lasting products. As more and more networked software gets embedded into physical devices that can last for decades, release engineers must have ways to build and deploy updates over this time span.³ Last year, Microsoft issued a patch addressing the WannaCry ransomware attack for Windows XP, a product whose first release was in 2001, whose last release was in 2008, and for which mainstream and extended support ended in 2009 and 2014, respectively. If you’re impressed by this feat, consider that some aircraft flying today had their avionics systems designed in the 1970s and that their software was built on minicomputers that are no longer in production.

Meanwhile, modern software development practices add to the demands on the release-engineering team. Agile development’s emphasis on working software and on responding to changing requirements necessitates frequent builds. In organizations practicing continuous delivery, these builds might become complete releases. Further upstream, the agile movement’s principles of sustainable development and unmitting attention to technical excellence prompt for the continuous integration of developer work into mainline builds, static-analysis checks, and tests.

**Best Practices Save the Day**

Fortunately, release-engineering teams can adopt a number of practices and tools to address the challenges they face.

First comes the across-the-board adoption of a revision control system. Aiming at a fully reproducible build environment, the corresponding repositories should contain everything required for the build. This means not only the software and associated components but also the complete build toolchain and even the OS. Remember, this might save your colleagues’ day when in 30 years your OS’s installation CDs are available only on display at the Computer History Museum.

Managing this efficiently requires careful planning and segregation. For example, you don’t want developers to synchronize on their laptop’s repositories the artwork heaps associated with the game they build, and you probably don’t want to maintain a separate repository with compiler and OS releases for each of the hundred components you build. On the other hand, you must set up a swift, reliable mechanism to track and incorporate into your repository the security updates of all the tools and components you use, even across long transitive dependencies.

Then comes build tooling. Each software development language and platform has its own peculiarities regarding the separate compilation of modules, the detection of dependencies, the linking of compiled elements, and the packaging of the result. Look at build tools such as Ant, Automake, Bazel, CMake, Gradle, Make, Maven, NAnt, Ninja, and npm, and adopt the tools best suited for the job at hand. Ensure that the tools you choose can handle the complete build cycle, including static code analysis, testing, documentation generation, and unattended operation on a build server. Resist the temptation to delegate build automation to the developers’ IDE; the IDE build features are simply not cut out for the demands of industrial-scale release engineering.

---

**Welcome New Editorial Board Members**

George Fairbanks is a software engineer at Google. For the past few years he’s been working on model-minded development, a synthesis of object-oriented analysis and design and software architecture. Fairbanks received a PhD in software engineering from Carnegie Mellon University. He’ll be taking over from Eoin Woods as the editor of the Pragmatic Architect department. Contact him at gf@georgefairbanks.com.

Mika Mäntylä is a professor of software engineering at the University of Oulu. His research interests include empirical software engineering, software testing, software maintenance, mining software repositories, and behavioral software engineering. Mäntylä received a DSc in software engineering from the Helsinki University of Technology. He’s IEEE Software’s associate editor for software testing and maintainability. Contact him at mika.mantyla@oulu.fi.
Treat the configuration files you write for build automation just as you would treat any other software. Make them modular, maintainable, portable, and efficient. Apply the DRY (Don’t Repeat Yourself) principle by centralizing configuration options into a few files and delegating boilerplate into separate functions or scripts. For example, rather than repeating a long sequence of shell instructions in tens of makefiles, put them into a separate script that the makefiles invoke.

Provide your developers a consistent, feature-rich environment on which to write, build, and test the software. Many technologies can help you here. Start by adopting for the developer environment a virtual host or container. This can be based on Docker, Hyper-V, KVM (Kernel-Based Virtual Machine), VirtualBox, VMware, or a cloud-based offering. Then use a configuration management tool, such as Ansible, Chef, or Puppet, to specify and set up the developer tooling. For extra points, automate the creation of the virtual environment using Vagrant.

Ensure your builds’ reliability and trustworthiness by making them verifiably reproducible (see reproducible-builds.org). The gold standard here is for repeated builds of the same software elements to produce bit-identical results. Select tools that can handle unattended operation, and apply that principle to the whole build. Problems can surface here from tools that insist on human input, such as key phrases for code signing or the acceptance of a license during installation. Once you’ve created a fully automated repeatable build, make it run continuously, after each commit into the source code repository. There are many tools and services, such as Hudson, Jenkins, Team Foundation Server, and Travis CI, that you can adopt for this purpose.

Serve your customers (in or outside your organization) by delivering your software as a proper package rather than an executable file or an archive or ZIP file to be dumped at the installation’s location. The package, when used by the appropriate package manager or installation system, should handle installation, upgrades, removal, logging, operation simulation, dependency tracking, and the unique identification of the build that created it. In particular, test and verify early on that the upgrade process works reliably and is itself upgradable. Discovering problems in the upgrade process after the software has shipped can be really painful for both the release-engineering team and its customers.

In the end, a happy and productive release-engineering team needs only to appreciate the challenges of modern release engineering and diligently apply the well-known best practices and tools.

References
got flaws?

Find out more and get involved: cybersecurity.ieee.org