One Script to Rule Them All: Unifying Build Processes Across Platforms

A whitepaper for the 2020 Collegeville Workshop on Scientific Software, focussing on Developer Productivity.

Jason M. Gates, Josh Braun, David Collins; *Sandia National Laboratories*

The Electromagnetic Plasmas in Realistic Environments (EMPIRE) team at Sandia National Laboratories focuses on next-generation electromagnetic/electrostatic/fluid dynamic code development. The codebase consists of seven actively developed main repositories, is built on top of Kokkos and Trilinos, and is intended to run on everything from Linux desktops to high-performance computers (HPCs) to bleeding edge testbed platforms. A year ago, our `BuildScripts` repository was a bit of an unorganzied mess. It had grown organically over time, with bash scripts here and there for running on different platforms, with different configurations, etc. In addition to what had been collected in the repository, developers also had their own scripts lying around for setting up their environment and configuring the code. In the summer of 2019, we undertook an effort to unify our build process across platforms and create a "one build script to rule them all," so to speak. The need for modern scientific codes to consistently build and execute in such diverse computing environments, and to migrate swiftly to new environments, points to the need for such a unified build system. Investing the time, money, and energy in developing such an infrastructure pays dividends in productivity, both the scientific developer and the DevOps engineer.

Language

We chose Python as the scripting language (specifically version 3.6+, in order to take advantage of f-strings; see PEP 498) for two primary reasons: documentation and unit testing. Python has agreed-upon conventions for documenting your code with *docstrings* (see PEP 257). We decided to utilize the Google docstring format to document all modules/classes/functions, and then use Sphinx (the documentation engine behind ReadTheDocs and the Python documentation) to generate easy-to-use HTML documentation for our build infrastructure. Emphasis was also placed on keeping functions small, contained, and as readable as possible. The other substantial benefit of Python is the ability to unit test with pytest. Though unit testing in bash is possible, it's not nearly as friendly or full featured as all that pytest provides. The consistent documentation, unit testing, and code style (see PEP 8) significantly reduce the burden on the DevOps engineers maintaining this infrastructure.

Replicability

The ability to easily replicate a build and debug a failure was at the forefront of our minds as we designed this script, so we built in a number of useful features to make that process as painless as possible.

Since cloning, configuring, and building a code ultimately requires interacting with the shell, we built what we've called our `Logger` utility. Any print statement or shell command is executed via a `Logger` object, which captures useful information:

- The command being executed
- A message describing why the command is being executed (encouraging good documentation)
- The start time and duration
- The current working directory
- `stdout` and `stderr`
- The return code

When the `Logger` is finalized, all this information is processed into a HTML log file so you have a thorough record or what was done, when, where, and why, which significantly eases debugging.

While the `Logger` can show you what happened last time, it doesn't provide you with the ability to do the exact same thing again. To that end, we built in a replay capability. When the script is run, it will drop a `.build_empire_replay` file containing a `dict` of all the options the script was run with, whether passed in at the command line, or set by default. If you need to replay exactly what heppened last time, simply pass the replay file to the script.

To ensure you know exactly what was done, the script finishes by printing a high-level execution summary:

- What script was run
- What machine it was run on
- What command line arguments were specifically passed to it
- What was cloned where
- What was configured, where, and how
- What was built and where
- What was tested and where, including a high-level results summary
- Which SHA1s were used
- How long everything took

While knowing what was done is helpful, something notoriously difficult in our context is ensuring you have the right environment established before doing anything. For this our script builds on top of the work done by the Advanced Technology Development & Mitigation (ATDM) DevOps team within Trilinos (see here and here). In any build or install directory you'll find a `load_matching_env.sh` script. Source it, and you'll get the exact same envi-

ronment that was used to create whatever build/install directory you're in. If you need to need to use somebody else's build, whether created by another developer or Jenkins, source that script and then you're off to the races.

Now if a developer runs into a problem and they need help with it, they can file an issue in GitLab, include the script execution summary, attach the replay and log files, and a teammate should be able to reproduce the problem in no time. No more time wasted to, "What machine were you on? What was your environment? How did you configure? What's the error you're seeing?" It's all right there, and it works at the push of a button.

Modularity

While we fully intended for this script to be used by all EMPIRE users and developers, we also designed it with Jenkins in mind. "Amplifying feedback loops," the second of The Three Ways of DevOps, was a guiding principle when designing our top-level Jenkins Pipeline architecture. As such, we wanted the ability to fail as fast as possible, and to pick up where we left off if necessary. This meant designing our pipeline, and the underlying Python script, with modularity in mind. Any given stage would need to be able to be executed independently of the others. While a user might want to run `build_empire.py --stage clone configure build ctest pytest vvtest extest`, the modularity built in gives Jenkins the ability to fire off the various test stages in parallel, for instance. And while Jenkins probably always wants to start by cloning the repositories, a user would likely want the ability to point to local clones with work in progress, and that flexibility is afforded here as well.

Flexibility

As mentioned previously, the EMPIRE code consists of a number of repositories, and as such the team consists of a number of sub-teams that handle different collections of those repositories. When doing daily development, a team member will likely only be dealing with a small subset of the repositories, so it'd be ideal to not have to build everything all together. At the same time, most of our Jenkins testing exists to ensure that everything builds together all the time. By default, therefore, the script will turn on all the repositories at once, but the user has the ability to turn on only the repositories they're working with.

Another significant boon has come from building in the ability to grab whatever git references (branches, SHA1s) you desire. By default, the script will check out `develop` in all our repositories, but the flexibility is there to, e.g., check out the branch you're working on in a particular repository, or in multiple repositories. This also affords us the flexibility to easily check out the SHA1s that were used in a previous run of the script, even if run by another developer, as given in the script execution summary. As an added bonus, this allowed us to stand up a

merge request testing infrastructure relatively easily.

A final significant design consideration was we wanted the ability to run the script in dry-run mode. When doing so the script would go through all the motions, but just tell you what it was going to do instead of actually doing it. For the application developer, that means you have the ability to see what you're about to execute before doing so, so you don't have to wait an hour or two to find out you ran the wrong thing. For the DevOps team, that means we have the ability to functionally test the infrastructure in dry-run mode and see if things look right in minutes, compared to running one of our pipelines for real, which would take up hours on multiple machines.

Discussion

We started building our "one script to rule them all" a year ago. A small subset of developers began using it last fall, contributing feedback to improve the feature development and polish. We officially rolled all this out to the team after integrating it into our Jenkins Pipeline infrastructure in January. Members of the team have enjoyed the ability to get up and running on any of the machines we support (seven different platforms, with a variety of configurations) easily. It's as close as we can get to Staples' "easy button" for a high-performance scientific codebase consisting of multiple repositories intended to run on next-generation architectures.

For the DevOps team, this has been an absolute game-changer. It enabled us to modularize our Jenkins Pipeline infrastructure, spread our testing out across multiple machines, and fail as quickly as possible. That, in turn, has made developers more responsive to failures, so as a team we spend less time debugging and more time developing the scientific capabilities of the code. The maintainability and extensibility improvements have also accelerated the pace of development on our DevOps infrastructure. We've been able to stand up additional pipelines to, e.g., run more in-depth testing, or deploy builds to a variety of platforms, and we were able to do so relatively quickly. These improvements mean the code is generally more stable, and the team is able to spend less time configuring and building the code and more time doing real work.

The effort to unify our build process across platforms was substantial, requiring a number of months of work before starting to reap the benefits. The productivity gains for scientific developers and DevOps engineers alike are substantial, which tells us our hypothesis at the beginning—that investing the time, money, and energy in developing such a unified build infrastructure would be well worth the effort—was spot on. Given the need to continuously be productive in our ever-evolving and increasingly diverse environment, we see it as the only way to go.

Acknowledgements

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. SAND2020-6668 C