Deploying and Tracking Software with the NCCS Software Provisioning tool (NSP)

Asa Rentschler
rentschleraj@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Nicholas Hagerty
hagertynl@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Elijah MacCarthy
maccarthyea@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Edwin F. Posada
posadacorref@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Abstract

The National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory has a long history of deploying ground-breaking leadership-class supercomputers for the U.S. Department of Energy. The latest in this line of supercomputers is Frontier, the first supercomputer to break the exascale barrier (10^{18} floating-point operations per second) on the TOP500 list. Frontier serves a wide array of scientific domains, from traditional simulation-based workloads to newer AI and Machine Learning workloads. To best serve the NCCS user community, NCCS uses Spack to deploy a comprehensive software stack of scientific software packages, providing straightforward access to these packages through Lmod Environment Modules. Maintaining a large software stack while also including multiple new compiler releases each year is a very time-consuming task. Additionally, it is not straightforward to provide a software stack alongside existing vendor-provided software such as the HPE/Cray Programming Environment (CPE), and existing CPE, Spack, and Lmod integration does not allow for multiple versions of GPU libraries such as AMD's ROCm to be used. To address these challenges and shortcomings, NCCS has developed the NCCS Software Provisioning tool (NSP), a tool for deploying and monitoring software stacks on HPC systems. NSP allows NCCS to quickly and effectively provision software stacks from the ground up using template-driven recipes and configuration files. NSP is successfully deployed on Frontier and several other NCCS clusters, enabling the NCCS software team to quickly deploy software stacks for newly-released compilers, expand current software offerings, better support GPU-based software, and monitor Lmod module usage to identify unused software packages that can be removed from the software stack. In this work, we discuss the shortcomings of the previous CPE, Spack, and Lmod usage at NCCS, provide further details on the implementation and structure of NSP, then discuss the benefits that NSP provides.

CUG 2025, New York, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts

• **Software and its engineering** → **Software usability**; **System administration**; *Software version control.*

Keywords

Software deployments, High-Performance Computing, Spack, Configuration as code

ACM Reference Format:
Asa Rentschler, Nicholas Hagerty, Elijah MacCarthy, and Edwin F. Posada. 2018. Deploying and Tracking Software with the NCCS Software Provisioning tool (NSP) . In *Proceedings of Cray User Group 2025 Computing Horizons (CUG 2025).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL) hosts the Oak Ridge Leadership Computing Facility (OLCF), which is tasked with providing world-class computational resources and technical expertise for researchers investigating the largest problems in science. OLCF's current production computing resource is the exascale supercomputer, Frontier, an HPE/Cray EX powered by AMD Instinct MI250X GPUs and an HPE Slingshot 11 interconnect [3]. Frontier debuted in May 2022 with a ground-breaking achievement of 1.102 quintillion floating-point operations per second (FLOPs) on the High-Performance Linpack (HPL) benchmark, the first to officially break the 1-exaFLOP barrier on HPL [1].

The HPE/Cray Programming Environment (CPE) on Frontier provides access to two vendor-provided compilers and one open-source compiler – CCE (HPE), ROCm (AMD), and GCC (open-source). For each of these compilers, CPE provides various computational libraries such as Cray MPICH, libsci (HPE's BLAS/LAPACK implementation), and HDF5, among other libraries [6]. CPE is a long-standing and well-known offering on HPE/Cray supercomputers, but the many software and technological advancements made during the U.S. Department of Energy Office of Science and National Nuclear Security Administration's joint investment, the Exascale Supercomputing Program (ECP)[1], has created an explosion of scientific software beyond the offerings of CPE. Since ECP's end,

[1]https://www.exascaleproject.org

the Extreme-scale Scientific Software Stack (E4S)[2] [10] has curated and sustained many of the software packages from ECP, and many E4S packages are provided on Frontier. As such, it is necessary for many HPC centers to provide a more exhaustive software stack in addition to CPE through an HPC-oriented package manager, such as EasyBuild[3] or Spack[4] [4, 5].

Providing a software stack that leverages existing vendor-provided components while providing additional software packages in a flexible and reproducible manner poses many challenges. These challenges include creating long configuration files to define existing vendor-provided components, maintaining configuration files that have hard-coded versions and paths, providing multiple versions of each software, and periodically reviewing which center-provided software is still relevant. For example, Frontier's original Spack configuration file was more than 1800 lines long while only providing software for two ROCm versions and three CCE versions. At the time of this writing, Frontier provides nine different versions of ROCm (considering only X.Y versions, e.g., 5.7) and eight different versions of CCE (counting bug-fix versions such as 15.0.1, since they are in separate CPE releases). Installing outdated or unused software packages also contributes to the excessive length of Spack configuration files.

In this work, we present the NCCS Software Provisioning tool (NSP), a Configuration as Code (CaC)-focused tool designed to improve reproducibility, implement version control, and enable automation of software deployments, as well as provide tracking capabilities for software usage. NSP enables greater reproducibility and efficiency through the use of Ansible, which is an automation tool commonly used to synchronize and maintain configuration files on HPC systems. NSP extends this common use case of maintaining configuration files by adding roles to perform common tasks like installing Lmod [8], setting up Spack environments, and installing compilers, among other roles. We focus on two primary functions of NSP: integration with Spack (Section 3) and integration with Lmod (Section 4). Section 5 provides a brief overview of other NSP capabilities and Section 6 presents opportunities for automation through Continuous Integration (CI).

2 Building Blocks of NSP

The NCCS Software Provisioning tool (NSP) is designed with Configuration as Code (CaC) principles in mind to ensure consistency, version control, and automation. As shown in figure 1, NSP leverages Ansible "roles" to perform various tasks, including installing compilers, synchronizing configuration files, or deploying tools, such as Spack and Lmod, while Jenkins will enable automation.

This section provides further discussion of Ansible, Spack, and Lmod, three of the primary components of NSP. NSP integration with CI tools such as Jenkins is under development and is discussed further in Section 6.

[2]https://e4s.io
[3]https://easybuild.io
[4]https://spack.io

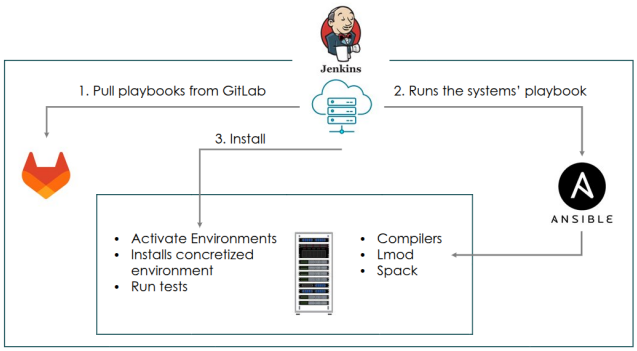

Figure 1: Overview of the target NCCS Software Provisioning Tool (NSP) design

2.1 Ansible

Ansible[5] is an open-source automation tool commonly used by system administrators for automating operations like provisioning and synchronizing configuration files [7]. Ansible is built upon the concept of a "playbook", which is a file that defines the order in which Ansible operations should be performed. These operations are defined through Ansible "roles", which are ordered lists of Ansible "tasks". An example of a single role could be installing and configuring Lmod, and the tasks within that could be (1) checking if Lmod is already installed, (2) downloading Lmod, (3) untarring the Lmod download, and so on.

Roles are named by a directory name, and all files within that directory contribute to defining tasks, providing template files, and setting default variables for those tasks, among other capabilities. Each role must provide at minimum a directory named `tasks` with a file named `main.yaml` in it, which is where Ansible starts execution of a role. From there, `main.yaml` may call other YAML files in the `tasks` directory. For example, a role for Lmod may have the following tasks:

- `tasks/main.yaml` – the main execution file, which checks if Lmod is already installed and calls `install.yaml` if not
- `tasks/install.yaml` – instructions for downloading and installing Lmod
- `tasks/lua_bootstrap.yaml` – instructions for installing Lua, a pre-requisite of Lmod

These task files may consume variables set by the user (or the default variables) and may issue commands to evaluate expressions in a Jinja2-style template file.

2.2 Spack

Spack[6] is a package manager developed by Lawrence Livermore National Laboratory, which supports both supercomputers and local Linux or Mac workstations [4]. Spack is used both by system administrators and users to compile complex scientific software stacks. Prior to NSP, Spack was used to provide a large software stack of more than 100 applications (not counting software compiled as a dependency of other software) for each set of compilers on Frontier.

[5]https://docs.ansible.com
[6]https://spack.io

These software stacks were organized into "environments", which are software stacks that are provisioned through configuration files, rather than interactive commands. This software is made available to users by Lmod Environment Modules, which is discussed later in this section.

NSP does not replace Spack but rather builds on top of it, using Ansible to create and maintain multiple distinct Spack environments while implementing version control on the Spack configuration files and improving efficiency through templates. This usage is discussed in detail in Section 3.

2.3 Lmod

Lmod [7] is a module system with support for both Lua and TCL Environment Modules, designed to enhance support for hierarchical modules. That is, modules that depend on other modules and are provided in a tree-like directory structure. Lmod is developed by Texas Advanced Computing Center (TACC) and is packaged as part of the HPE/Cray Programming Environment (CPE) on Frontier. Using Lmod, users have the ability to load one of several available versions of CPE, as well as any modules that OLCF provides. Lmod commands can be also modified or added to using Lmod hooks provided in `SitePackage.lua`. Lmod hooks are callable functions that are run at the time of certain actions, such as during a `module load`, `module unload`, or `module spider` command.

As is the case with Spack, NSP does not replace Lmod, but rather provisions it, enabling centralized version-controlled installation recipes and configuration files. This usage is discussed in detail in Section 4.

3 The Spack Role

The Spack Ansible role is a meta-layer to set up Spack and deploy all necessary configuration files for Spack environments. In this section, we provide details on the two functions of the Spack role, provisioning Spack and creating Spack environments, and provide further details on configuring Spack to deploy module hierarchies that are compatible with the Lmod hook.

3.1 Provisioning a Spack Installation

The first function of the Spack role is to provision a Spack installation. At the most basic level, this is done by providing a Git commit hash that the Spack role checks out once Spack is downloaded. However, NSP also supports providing Git-compatible patch files with modifications to be made to Spack source files once the specified Git commit hash is checked out. These patch files may modify specific packages or modify Spack source files. This functionality is critical to providing fixes to specific versions of Spack. In NSP, the patch files are also version-controlled through Git, allowing fine-grained control over local fixes.

3.2 Templated Spack Environments

Ansible features Jinja2 templates that enable creating files from a template using variables. NSP leverages this by defining a CPE manifest of variables for the Spack role that contains the list of the CPE-provided packages and versions for a given CPE release. For instance, Listing 1 shows a portion of the manifest for packages from the CPE 24.11 release. The CPE manifest variables are then used to generate Spack environment configuration files. Specifically, Listing 2 illustrates how the file `packages.yaml` is generated by interpolating the variables in the template `packages.yaml.j2`. One advantage of using templates is the standardization of available configuration options and centralization of the configuration file format. By defining all variables used in the templates in a single location, templates also help to avoid code duplication and ensure consistency across multiple configuration files.

```
...
# Cray Packages
cray_dyninst: 12.3.4
cray_fftw: 3.3.10.9
cray_hdf5: 1.14.3.3
cray_hdf5_parallel: 1.14.3.3
cray_mrnet: 5.1.4
...
```

Listing 1: CPE 24.11 manifest in NSP's Spack role.

All environments created by the Spack role are independent and deployed in separated locations, as shown in Listing 3. This ensures that each environment can be removed or modified without impacting the others. In this way, if an environment fails testing or support for portions of that environment is dropped, it can be easily removed and/or redeployed without disrupting the rest of the software stack. This independence-driven approach greatly simplifies adding new packages and environments.

Once the Spack role provisions Spack and creates the environments, the environments are ready to be concretized and installed. Although each environment installs software to an independent location, all environments generate modules to a common location using custom projections in Spack.

3.3 Spack Module Projections

Spack supports generating an Lmod module file hierarchy automatically, but that may only use what Spack calls "virtual" specs as part of this hierarchy (ie, MPI library or BLAS/LAPACK library). By this limitation, the AMD ROCm toolkit and the NVIDIA CUDA toolkit cannot be included in the hierarchy because ROCm and CUDA are not virtual specs. Additionally, when Spack creates a module hierarchy, it adds short hashes to the paths that make it difficult to integrate with external vendor-provided modules. Using the Lmod hook described in Section 4, NSP dynamically adds paths to `MODULEPATH`, but that requires that Spack generates the module file hierarchy including non-virtual specs (like ROCm/CUDA) and generates paths without hashes. To do this we disable Spack's hierarchy by setting `hierarchy:: [ ]`.

[7] https://lmod.readthedocs.io/en/latest/

```
packages:
    ...
  cray-dyninst:
    buildable: false
    externals:
    - spec: cray-dyninst@{{ cray_dyninst }}
      modules:
      - cray-dyninst/{{ cray_dyninst }}

  cray-fftw:
    buildable: false
    externals:
    - spec: cray-fftw@{{ cray_fftw }}
      modules:
      - cray-fftw/{{ cray_fftw }}
...
```

```
packages:
    ...
  cray-dyninst:
    buildable: false
    externals:
    - spec: cray-dyninst@12.3.4
      modules:
      - cray-dyninst/12.3.4

  cray-fftw:
    buildable: false
    externals:
    - spec: cray-fftw@3.3.18.9
      modules:
      - cray-fftw/3.3.18.9
...
```

Listing 2: A template `packages.yaml.j2` (left) next to the fulfilled configuration file `packages.yaml` (right) for CPE 24.11.

```
/frontier/
├── cpe24.11-cpu
│   ├── bootstrap
│   ├── compilers.yaml
│   ├── config.yaml .3 packages.yaml
│   ├── spack.lock
│   └── spack.yaml
└── cpe24.11-gpu
    ├── bootstrap
    ├── compilers.yaml
    ├── config.yaml
    ├── packages.yaml
    ├── spack.lock
    └── spack.yaml
```

Listing 3: File structure of the `cpe24.11-cpu` and `cpe24.11-gpu` Spack environments created by NSP.

```
default:
  roots:
    lmod: # root module path
  enable:
    - lmod
  arch_folder: false
  lmod:
    core_compilers:: []
    all:
      autoload: none
    exclude_implicits: true
    hash_length: 0
    hierarchy:: [ ]
    projections:
      ^llvm-amdgpu ^mpi:
        ↪  "{^mpi.name}-{^mpi.version}/rocm-{^llvm-amdgpu.version}/
        ↪  {compiler.name}-{compiler.version}/{name}/{version}"
      ^cuda ^mpi: "{^mpi.name}-{^mpi.version}/cuda-{^cuda.version}/{co
        ↪  mpiler.name}-{compiler.version}/{name}/{version}"
      ^llvm-amdgpu: "rocm-{^llvm-amdgpu.version}/{compiler.name}-{comp
        ↪  iler.version}/{name}/{version}"
      ^cuda: "cuda-{^cuda.version}/{compiler.name}-{compiler.version}/
        ↪  {name}/{version}"
      ^mpi: "{^mpi.name}-{^mpi.version}/{compiler.name}-{compiler.vers
        ↪  ion}/{name}/{version}"
      all: "{compiler.name}-{compiler.version}/{name}/{version}"
```

Listing 4: Module Configuration for Spack

To generate a module hierarchy without hashes we use the `projections` feature in Spack, which can define a path for a module such that it does not use hashes and includes non-virtual specs like ROCm. The Spack environment's `autoload` parameter must also be set to `none` because projections are considered by Spack as part of the modules' name. If `autoload` is not set to `none` dependent modules will try loading prerequisite modules with the full projection string. For the majority of cases disabling `autoload` is acceptable because Spack uses `RPATH` to direct binaries to the correct libraries regardless of the modules for those libraries being loaded.

4 The Lmod Role

The Lmod Ansible role in NSP is an end-to-end solution for deploying and customizing Lmod installations. The role may install Lmod from scratch or configure an existing Lmod installation, as in the case of CPE. To configure Lmod, the role renders templates for various Lmod configuration files such as `admin.list`, `lmodrc.lua`, or perhaps most significantly `SitePackage.lua`, which holds NCCS site-custom hooks. The three NCCS site-custom hooks presented below in Sections 4.2, 4.3, and 4.4 are critical to supporting both CPE-provided modules and NCCS-provided modules.

4.1 Challenges of Integrating Site Modules with CrayPE

One of the most popular features of Lmod is its ability to track the dependency relationships between modules. This is achieved through the Lmod module function `prepend_path`, which adds paths to the shell environment variable `MODULEPATH`. This variable holds a list of paths which are used by Lmod to search for modules. Any time a path is added to `MODULEPATH` through a call to `prepend_path`, the module adding that path is considered by Lmod as the parent in the module hierarchy for any modules found in that path. For example, suppose there are two different compilers, `A` and `B`, each exposed through their respective modules. A module is provided for the popular Molecular Dynamics software, LAMMPS [9]. A user loads `lammps` while compiler `A` is loaded. If the module for compiler `A` were to be switched with that of compiler `B`, a corresponding `lammps` module would automatically be searched for in the module hierarchy under module `B`. If found, the `lammps` modules would be swapped to maintain consistency with the currently-loaded compiler. If no corresponding `lammps` module is found, the `lammps` module would be marked as "inactive" due to the removal of its providing module, compiler `A`, and no suitable replacement provided by compiler `B`.

Despite this powerful capability, there is one significant limitation to Lmod's ability to record module dependencies. If calls to `prepend_path` are made dynamically (e.g. change based on environment variables), Lmod is not able to predict the possible paths that could be added. This means that Lmod can see the modules that would be added in the current environment but all other possible modules are invisible because they exist in a location Lmod doesn't know to search for. Thus, by default, for Lmod to work reliably and account for all modules, calls to `prepend_path` must be static (i.e. hard-coded).

This limitation impacts NCCS systems because CPE makes extensive use of dynamic calls to `prepend_path`. While this approach does have advantages like cutting down on the total number of module files, it creates challenges offering site-provided modules alongside CPE. For instance, the `cray-mpich` module provided by CPE dynamically detects which compiler is loaded, and constructs the appropriate CPE module path for use with that compiler. In order to tie in a second module hierarchy, such as the modules provided by Spack, the CPE-provided compiler and `cray-mpich` modules would need to be modified to add the additional paths to the constructed module path. These CPE-provided compiler and MPI module files are already very complex and can change between CPE releases, so any modifications to these module files could bring unexpected behavior.

4.2 Enabling a Dynamic Software Stack

NSP leverages a combination of Lmod hooks and Spack projections to deploy site-provided modules alongside CPE. The use of Spack projections is discussed in Section 3.3, but in short, we eliminate the use of hashes in Spack's projections by disabling the default projections. Once Spack projections no longer include these hashes, we can implement a hook in `Lmod` to dynamically add/remove paths from `MODULEPATH` based on the currently loaded modules, including CPE-provided modules. This hook runs every time a module is loaded or unloaded. This solution bypasses modifying the CPE modules on HPE systems and is extremely flexible being used on non-HPE systems at NCCS as well.

Listing 5 is an example configuration in NSP for our hook.

```
NSP_LMOD_spack_modules: "{{ path to spack modules }}"
NSP_LMOD_hierarchy:
  compiler:
    members: [ 'cce', 'amd', 'gcc-native' ]
    paths:
        - { path: '|compiler.name|-|compiler.version|', weight: 20 }
        - { path: '|mpi.name|-|mpi.version|/|compiler.name|-|compiler.ve⌋
            ↪ rsion|', weight: 30 }
        - { path: '|gpu.name|-|gpu.version|/|compiler.name|-|compiler.ve⌋
            ↪ rsion|', weight: 40 }
        - { path: '|mpi.name|-|mpi.version|/|gpu.name|-|gpu.version|/|co⌋
            ↪ mpiler.name|-|compiler.version|', weight: 50 }
  gpu:
    ...
  mpi:
    ...
 NSP_LMOD_nv_mappings:
    # '%s' substitutes in the value provided by the module
    amd/6.2.4: { name: 'rocmcc', version: '%s' }
    gcc-native/13.2: { name: 'gcc', version: '%s' }
```

Listing 5: Example Hook Configuration

The first variable defined is `NSP_LMOD_spack_modules`, which tells the hook the root directory in which to look for Spack-generated module files.

The second variable, `NSP_LMOD_hierarchy`, tells the hook how to interpret the Spack module hierarchy. In this example, the Spack hierarchy consists of a `compiler`, `gpu` (such as NVIDIA's CUDA or AMD's ROCm toolkit) and `mpi`. Each of these hierarchy components lists a set of "members", which are the names of modules that belong to that component. For example, CPE provides compiler modules named `cce`, `amd`, and `gcc-native` on Frontier. All of the module files defined as members for a single component should not be able to be loaded at the same time (i.e., through Lmod's `family` function). In addition to the list of members for each component, the configuration also defines a list of paths which should be evaluated when any module from the list of members is loaded/unloaded. For each path, if all components required for that path are currently loaded, then that path is appended to `NSP_LMOD_spack_modules` and the full path is added to `MODULEPATH`. A similar process is used when modules are unloaded to remove paths from `MODULEPATH`. A weight parameter is also provided that tells Lmod the priority of each path, which is used when searching these paths for modules.

The final variable defined above is `NSP_LMOD_nv_mappings`. Sometimes Spack uses a different name for a compiler than CPE uses for the module. For instance, as shown above, CPE has a module called `amd` which provides the AMD compilers but Spack knows this compiler as `rocmcc`. `NSP_LMOD_nv_mappings` addresses this by mapping a module name and version to the name that Spack uses for its projections.

4.3 Using an Lmod hook to improve Lmod's `spider` command

As previously discussed, dynamically adding paths to `MODULEPATH` via `prepend_path` breaks `Lmod`'s ability to track the dependency relations between all modules. The most notable effect this has is with `module spider`, which uses the module dependency tree to search all available modules on a system. With dynamically added paths some modules will unfortunately not be found by `Lmod` giving users a limited set of the modules that are actually available. Once again we use a custom hook to overcome this issue. Our hook runs whenever `module spider` is invoked and uses the spider cache generated by `Lmod` in combination with a cache of our Spack modules which we generate. The NSP spider seeks to keep the same output format as Lmod's spider command but leverages NSP's knowledge of the spack projections to make sure all available modules are displayed.

4.4 Tracking Module Usage with Lmod

While NSP greatly accelerates the deployments of large software stacks, it is also critically important to optimize the software offerings to only what is used. This can be done through user surveys, formal software request mechanisms, or other request-focused pathways, but NSP enables a passive module tracking capability that logs information to a database each time a module is loaded.

Logging module usage is done through another Lmod hook, similar to the hook that handles `MODULEPATH`.

Lmod provides an example of logging `module load` usage to `syslog`, but on many NCCS systems, users do not have permission to log messages to `syslog` so NSP sends data to a RESTful API that curates and forwards data to a MariaDB database. From there the data can be visualized in a Grafana dashboard or queried from other locations to perform analysis. Both the API and the database reside on an NCCS-managed OpenShift[8] cluster. This database-based logging enables more powerful queries than those that are accessible with `syslog`, and integrates cohesively with existing monitoring infrastructure.

The REST API is written in Python using the Fast API[9] package. The REST API validates incoming data, collects multiple `module load` calls into a single request to the database, and enforces rate limits on a per-user basis. For example, if a single user runs `srun -n 8000 module load rocm`, the REST API will only send up to 100 records to the database from this user in a single minute. This prevents a single user from flooding the database with redundant loads. The REST API also reports how many messages were dropped by this rate-limiting filter and which user they were from.

The NSP Lmod logging hook uses the Linux `curl` utility with a 3-second timeout to send messages to the REST API while protecting against long delays if the API cannot be reached. Upon receiving a message, the REST API instantly returns an HTTP 202 "Accepted" status code, which indicates that the message has been received and is being processed. Once the HTTP 202 code is returned, the `curl` process will terminate and the client side will be free to continue executing Lmod functions. On the REST API server, the rate-limiting and validation is performed, then the message is sent to the database.

A performance scaling study was performed to ensure that database-based logging did not significantly increase the overhead of Lmod operations. A 9000-node job was used, in which one process on each node loaded the same module five times. This was designed to simulate a worst case scenario of 45,000 module loads in a short period of time from 9,000 clients. Figure 2 shows the distribution of the measured latency of each `module load` command. There is a slight increase in `module load` latency observable with the logger, on the order of 10% runtime, or 2 seconds. However, the minimum time is effectively equivalent, with a difference of just 0.01 seconds, demonstrating that under ideal conditions, the logger does not add substantial overhead. These results were determined to be acceptable for operations, and the module logging has been enabled on Frontier.

As noted before, tracking software usage is critically important to optimize the software stack. It usefulness is also apparent when determining the impact of deprecating old versions of software. A good example of this is deprecating CPE versions on Frontier. Because porting applications to a new compiler or GPU library version can cause code breakages or performance regressions, many users opt to remain on a known stable version of CPE. At NCCS we support this by keeping multiple CPE versions available; however, eventually certain CPE releases must be deprecated and removed.

[8]https://www.redhat.com/en/technologies/cloud-computing/openshift
[9]https://fastapi.tiangolo.com

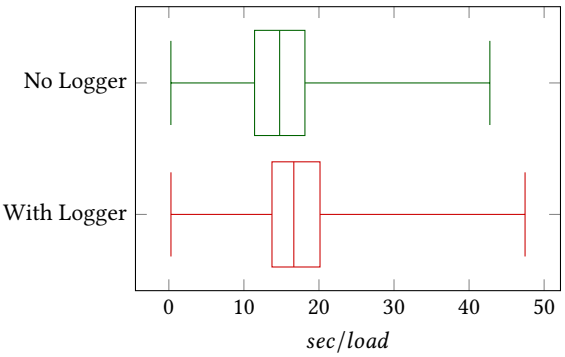

Figure 2: Box plot measuring the overhead of module logging

Having module usage data allows us to identify the impact of removing a CPE and target our efforts helping users move to newer CPE releases.

5 Other Roles

At NCCS, there are several cases in which a software is not installed by Spack. The most common example is GCC, which is installed in a short and easily-found prefix such as `/sw/system_name/gcc/13.2`. In these cases, a role is developed in NSP to install and configure the software. NSP maintains uniformity between these roles when possible by requiring the use of several specific Ansible variables such as the installation path or scratch/build directory. Additional roles have been developed for installing specific software including:

- `gcc` - installs the GNU Compiler Collection (GCC)
- `llvm` - installs the LLVM compiler
- `miniforge3` - installs the Miniforge conda installer
- `codee` - installs the Codee code inspection tool
- `oneapi` - installs the Intel OneAPI toolkit
- `cuda` - installs the NVIDIA device toolkit, CUDA

NSP automates the deployment of these packages and the generation of a corresponding module file.

NSP also provides a `files` role for deploying various configuration files and the `init` role for installing shell scripts to set up the NCCS software stack upon login. These roles are crucial for maintaining consistent user environments across the wide range of NCCS systems.

6 Future Work

NSP has demonstrated great value to production operations by successfully deploying the NCCS software stacks on Frontier and several other clusters. Looking forward, additional roles will continue to be added as further opportunities are identified. One larger campaign currently underway is to enhance NSP with Continuous Integration and Continuous Deployment (CI/CD) tools. For this effort, two open-source CI/CD tools have been selected: Jenkins and Jacamar CI. These two CI tools are discussed further in the remainder of this section.

6.1 Automation with Jenkins

Jenkins is a popular cross-platform, open-source CI/CD tool for automating tasks related to building, testing and deploying software. It allows predefined steps known as jobs, to be executed based on a trigger. While a trigger can be an event in a Version Control System (VCS), it can also be based on a time interval. Jenkins' functionality can be extended to accommodate a wide variety of projects through custom plugins. For projects that require functionalities that are not available within Jenkins, it is possible to develop a custom Jenkins plugin.

For the purposes of building and deploying software through NSP, existing Jenkins plugins provide sufficient functionality. Git/GitLab, Ansible and "Publish Over SSH" plugins will be relied upon for automation of NSP. Our software deployment requires that Jenkins integrates with Git for source control management (SCM), Ansible for building software, and Publish Over SSH for establishing connections to NCCS systems.

Our Jenkins automation approach sets up a free style project that uses poll SCM to trigger builds and deployments to NCCS systems. To accomplish this, we setup a Jenkins, Git/GitLab, Ansible integration that uses Publish Over SSH plugin within Jenkins to tunnel onto the host system for software deployment. The Publish Over SSH plugin allows username and password authentication as well as public key authentication. We utilize the username and password authentication method since this approach is for software deployment on open enclave NCCS systems. The username and passwords are passed to Jenkins through its credential configuration option where credentials are encrypted to enhance security.

Once connection is established with the remote NCCS system, the free style project kicks off by Poll SCM triggers enabled through the Jenkins Gitlab plugin. Since the GitLab repository is private, we setup a Jenkins credential for establishing connection with the repo. This ensures commits are picked up by the job and builds are automatically triggered based on schedules setup for poll SCM.

Though an Ansible Jenkins integration is setup through the Ansible plugin on Jenkins, we prefer an automated install on the target NCCS system for our software deployment needs. Hence, we utilize Execute commands over SSH to install Ansible on the target NCCS system. Through the Jenkins job, our installed Ansible runs the playbooks from the repository and spack is finally initialized to build and deploy the software.

6.2 Enhanced-Security Automation with Jacamar CI

Frontier resides in a more-restrictive enclave within NCCS called the "moderate" enclave. Moderate enclave NCCS systems require a more sophisticated authentication approach than is possible with our Jenkins server. As such, we are investigating using Jacamar CI for automating NSP integration. Jacamar CI is attached to the Gitlab repo just like a traditional Gitlab CI pipeline and authenticates to the host system to automate jobs through the `Jacamar-auth` application.

Jacamar CI is an open-source project originating from the Exascale Computing Program (ECP) Continuous Integration team, and serves as a bridge between traditional GitLab CI pipelines and HPC resources [2]. Jacamar CI ensures facility security requirements are met, which makes it suitable for authentication to our moderate NCCS systems. Once authentication to the moderate enclave NCCS system is established, the same CI pipeline as described above for Jenkins can be executed.

7 Conclusion

The NCCS at ORNL deploys an extensive software stack that supplements the offerings of the CPE to best support the large OLCF user base. Providing a software stack that seamlessly integrates into an existing vendor-provided stack like CPE poses many challenges. The Lmod environment module packaging of CPE is complex and subject to change without notice, which motivates the development of a non-invasive method for combining the two software stacks.

NCCS has developed the NSP tool to ensure reproducible and efficient software deployment, and to enable the unification of the CPE and NCCS-provided software stacks. NSP is a Configuration as Code focused tool written in Ansible that implements version control, improves the automation and reproducibility of software deployments, and enables tracking capabilities for software usage.

By utilizing templates, NSP greatly reduces the amount of code required to provision NCCS software stacks and configuration files, while also greatly increasing the efficiency with which new stacks can be deployed. This template-driven approach also allows NCCS to deploy multiple GPU-enabled stacks seamlessly.

NSP's two key functions are (1) integration with Spack (Section 3) and (2) integration with Lmod (Section 4). NSP exposes version-controlled configuration options to install and configure a Spack installation and create Spack environments, greatly accelerating the process of deploying new Spack environments. These new Spack environments are created from templates, which ensure that each Spack environment is consistent in format with previously-deployed Spack environments. These Spack environments are made available to users through Spack's Lmod module projections. NSP also configures Lmod and implements several Lmod hooks, which are callable functions that run at the time of certain actions, such as a `module load`. NSP leverages Lmod hooks to dynamically make modules available based on which CPE components such as compiler, MPI library, and device library (i.e., AMD's ROCm) are loaded. This enables CPE and the NCCS-provided stack to be successfully integrated. An additional Lmod hook has been developed to enable the logging of `module load` and `module unload` commands, which provides NCCS with new insights into software usage patterns.

NSP has been successfully deployed on NCCS' primary production resource, Frontier, as well as several other clusters at NCCS. NSP has greatly accelerated the addition of support for new CPE releases on Frontier and provides a reproducible method for deploying future software stacks on NCCS machines.

Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] 2022. TOP500 List June 2022. https://www.top500.org/lists/top500/2022/06/.

[2] Ryan Adamson, Paul Bryant, Dave Montoya, Jeff Neel, Erik Palmer, Ray Powell, Ryan Prout, and Peter Upton. 2024. Creating Continuous Integration Infrastructure for Software Development on US Department of Energy High-Performance Computing Systems. *Computing in Science & Engineering* 26, 1 (2024), 31–39.

[3] Scott Atchley, Christopher Zimmer, John Lange, David Bernholdt, Veronica Melesse Vergara, Thomas Beck, Michael Brim, Reuben Budiardja, Sunita Chandrasekaran, Markus Eisenbach, Thomas Evans, Matthew Ezell, Nicholas Frontiere, Antigoni Georgiadou, Joe Glenski, Philipp Grete, Steven Hamilton, John Holmen, Axel Huebl, Daniel Jacobson, Wayne Joubert, Kim Mcmahon, Elia Merzari, Stan Moore, Andrew Myers, Stephen Nichols, Sarp Oral, Thomas Papatheodore, Danny Perez, David M. Rogers, Evan Schneider, Jean-Luc Vay, and P. K. Yeung. 2023. Frontier: Exploring Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (<conf-loc>, <city>Denver</city>, <state>CO</state>, <country>USA</country>, </conf-loc>) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 52, 16 pages. doi:10.1145/3581784.3607089

[4] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. 2015. The Spack package manager: bringing order to HPC software chaos . In *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12. doi:10.1145/2807591.2807623

[5] Markus Geimer, Kenneth Hoste, and Robert McLay. 2014. Modern Scientific Software Management Using EasyBuild and Lmod. In *2014 First International Workshop on HPC User Support Tools*. 41–51. doi:10.1109/HUST.2014.8

[6] Hewlett Packard Enterprise 2025. *HPE Cray Programming Environment documentation.* Hewlett Packard Enterprise. https://cpe.ext.hpe.com/docs/latest/index.html.

[7] Lorin Hochstein and Rene Moser. 2017. *Ansible: Up and Running: Automating configuration management and deployment the easy way.* " O'Reilly Media, Inc.".

[8] Robert McLay, Karl W. Schulz, William L. Barth, and Tommy Minyard. 2011. Best practices for the deployment and management of production HPC clusters. In *State of the Practice Reports* (Seattle, Washington) *(SC '11)*. Association for Computing Machinery, New York, NY, USA, Article 9, 11 pages. doi:10.1145/2063348.2063360

[9] Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in 't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, Ray Shan, Mark J. Stevens, Julien Tranchida, Christian Trott, and Steven J. Plimpton. 2022. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications* 271 (2022), 108171. doi:10.1016/j.cpc.2021.108171

[10] James M. Willenbring, Sameer S. Shende, and Todd Gamblin. 2024. Providing a Flexible and Comprehensive Software Stack Via Spack, an Extreme-Scale Scientific Software Stack, and Software Development Kits. *Computing in Science & Engineering* 26, 1 (2024), 20–30. doi:10.1109/MCSE.2024.3395016