Simple, Secure, Internet Delivery of MOOSE-based Applications

July 2021

Jason Miller¹, Logan Harbour¹, Robert Carlsen¹, Andrew Slaughter¹, Brandon Biggs², and Cody Permann¹

¹Computational Frameworks
²High Performance Computing
DISCLAIMER

This information was prepared as an account of work sponsored by an agency of the U.S. Government. Neither the U.S. Government nor any agency thereof, nor any of their employees, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness, of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the U.S. Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the U.S. Nuclear Regulatory Commission.
Simple, Secure, Internet Delivery of MOOSE-based Applications

Jason Miller\textsuperscript{1}, Logan Harbour\textsuperscript{1}, Robert Carlsen\textsuperscript{1}, Andrew Slaughter\textsuperscript{1}, Brandon Biggs\textsuperscript{2}, and Cody Permann\textsuperscript{1}

\textsuperscript{1}Computational Frameworks
\textsuperscript{2}High Performance Computing

July 2021

Idaho National Laboratory
Computational Frameworks
Idaho Falls, Idaho 83415

http://www.inl.gov
Page intentionally left blank
SUMMARY

Application packaging and distribution are the final steps for delivering software to end-users; both are frequently neglected when creating scientific software. Commercial businesses rely on electronic distribution systems that have rendered disk drives obsolete. Still, national laboratories continue to rely heavily on removable media to distribute and limit access to controlled applications. With increasing concerns of unauthorized copying of sensitive applications, a modern distribution system that utilizes cryptographically secure communication and authentication protocols has been developed. This new distribution system will secure the chain of custody for nuclear software while simultaneously simplifying access to these tools.

This report summarizes four primary advancements made toward the secure distribution of Nuclear Energy Advanced Modeling and Simulation (NEAMS)-developed, Multiphysics Object Oriented Simulation Environment (MOOSE)-based applications: application installation, package distribution, automated package building, and distribution of documentation.

NEAMS is currently developing more than ten separate applications based on the open-source MOOSE Framework. Distribution of these applications has primarily been accomplished by distributing source code, with end-users compiling the applications themselves. This work created a mechanism where MOOSE applications can be installed in a similar way to any other software. This allows both administrators and end-users simplified access to runnable executables.

With this new installation capability, it was then possible to rethink distribution. A new, secure capability for delivering MOOSE-based applications over the internet has been created. This system requires unique cryptographic tokens for authentication, greatly securing the custody chain for software. Once granted access, installation of any NEAMS code can be accomplished with these terminal commands:

```
conda install ncrc
crc install ncrc-bison
```

After these two commands (and authenticating) the BISON application will be securely downloaded from Idaho National Laboratory (INL)'s servers, installed, and ready to use.

To enable this new distribution capability to be successful, the open-source Continuous Integration, Verification, Enhancement, and Testing (CIVET) Continuous Integration (CI) capability was augmented to add Continuous Delivery (CD). CD enables the automated building and packaging of MOOSE-based applications as they are modified by development teams, ensuring that our customers can obtain up-to-date versions of the software at any time.

The need for instruction on how to use these applications was addressed through modifications to the MOOSE documentation system. The MooseDocs capability, which enables robust documentation of MOOSE-based applications, has been extended to allow both for the installation of documentation and the packaging of documentation with installed applications.

Together, these enhancements form the core of a new, secure distribution mechanism for nuclear simulation tools. In concert with the Nuclear Computational Resource Center (NCRC) NEAMS-developed applications will now be straightforward to obtain securely.
## Contents

**Summary**

1 **INTRODUCTION**

2 **INSTALLABLE APPLICATIONS**
   2.1 Shared Library Linking .......................................................... 6
   2.2 File Path Handling ................................................................. 6
   2.3 Testing ..................................................................................... 6
   2.4 Installed Files ......................................................................... 8

3 **THE CONDA PACKAGE MANAGER SYSTEM**
   3.1 Conda Deficiencies .................................................................. 10
   3.2 Conda Two-factor Authentication ............................................ 12
   3.3 End-User System Interaction ................................................... 12

4 **CIVET: CONTINUOUS INTEGRATION SYSTEM**
   4.1 Continuous Delivery Capability ............................................... 14
   4.2 Automated Builds and Version Control .................................... 15

5 **MOOSEDUCS: APPLICATION DOCUMENTATION**
   5.1 Installable Documentation ....................................................... 18

6 **CONCLUSIONS**
   6.1 Future Work ............................................................................. 21

**References**


ACRONYMS

2FA  Two-factor Authentication
CD   Continuous Delivery
CI   Continuous Integration
CIVET  Continuous Integration, Verification, Enhancement, and Testing
COG  Code Oversight Group
DOE-NE  Department of Energy’s Office of Nuclear Energy
HPC  High Performance Computing
INL  Idaho National Laboratory
LDAP  Lightweight Directory Access Protocol
MFA  Multifactor Authentication
MOOSE  Multiphysics Object Oriented Simulation Environment
NCRC  Nuclear Computational Resource Center
NE  Nuclear Energy
NEAMS  Nuclear Energy Advanced Modeling and Simulation
NFS  Network File System
NQA  Nuclear Quality Assurance
OSS  Open Source Software
RSA  Rivest-Shamir-Adleman
SMB  Server Message Block
SSL  Secure Socket Layer
TMP  Technology Management Plan
Page intentionally left blank
1 INTRODUCTION

This report summarizes activities performed by the Idaho National Laboratory (INL) software frameworks team to create and deploy an innovative application distribution and protection mechanism. The Nuclear Energy Advanced Modeling and Simulation (NEAMS) program has strengthened its cyber-protection posture in recent years to protect against inappropriate distribution of Nuclear Energy (NE) funded software. As part of that process, NEAMS released a Technology Management Plan (TMP) document describing the rigorous expectations of the national laboratories funded by NE in terms of licensing and the handling of their respective software products. The TMP provides a recommendation for distributing codes based on a graded risk approach to various types of licensees. In response to these expectations, deployment technologies were evaluated based on their ability to supply the necessary capabilities to support the TMP. Additionally, software enhancements along with new development structures were formulated to implement a modern deployment methodology for NEAMS software. The resulting product represents a shift in deployment methods for laboratory software.

The national laboratories have had a long history of licensing and distributing software developed at their respective institutions. Historically, the primary mechanism for delivering licensed software has been done on physical media mailed directly to the licensee. In many cases, once received, this software is void of any copy-protection mechanism, making it easy for the end-user to install it and use it as they see fit. In other cases, the software may have had limited key-based protection, but as part of the activities undertaken by this team, a brief analysis showed these systems falling short of modern cryptographic standards for protection. Some mechanisms relied on security through obscurity, meaning knowledge of the security mechanisms themselves could circumvent the security. While the distribution of physical media required the licensee to list a valid physical location so the media could be received, it was concluded that minimal code protection existed to prevent unauthorized use.

The NEAMS TMP describes a graded approach to reduce the risk of unauthorized use of sensitive software. An example set of access levels are briefly summarized here:

- **Level 0**, No Access: User access to software in any form is not granted.
- **Level 1**, Remote Execution (default): Users have the ability to execute protected (non-copyable)
application binaries that are installed on a controlled-computing system. This includes console-based and web-based solutions.

- **Level 2, Installable Binary Access:** User access to binaries managed through one of the program’s approved code distribution methods. This level may prove most convenient for many licensees as they can authenticate the downloaded executable for software releases or developmental versions.

- **Level 3, Header and Library Access:** User access to binaries, libraries, and application headers is made available to facilitate the development of “pluggable” objects. New objects may be built and compiled by the end-user and linked at run-time to the application. For example, a new material model may be developed for the BISON nuclear fuels performance code for evaluation. The material model is coded, compiled, and linked with the BISON headers to provide a BISON executable with the new material model. No repository access is granted.

- **Level 4, Full Source Access:** Complete access to the source code is provided to software developers that have completed all training required to contribute or approve changes to the code repository.

For the last several years, the NEAMS program has been granting Level 4 access to most licensees. Advanced simulation software that is designed to run on supercomputers often needs to be compiled from source. This, coupled with the fact that the base framework (Multiphysics Object Oriented Simulation Environment (MOOSE)) is available as Open Source Software (OSS), made it natural to operate in this fashion. No additional burdens were placed on the development team as every user was given full instructions on obtaining, building, and running the software. This also meant capable end-users could potentially troubleshoot and extend the software directly themselves. In addition, users could always view the source to understand how certain calculations were performed. The obvious downside to this approach was that it immediately gave every licensed user a complete copy of the source code as well as the ability to build and install that software on any system worldwide.

With the complexity of the simulation software being developed through NE laboratories, a
temporary workaround to giving away the full source was devised and implemented quickly as a proof of concept in Fiscal Year 2019. The INL would build binaries on its High Performance Computing (HPC) systems and allow licensees to obtain HPC accounts to run directly on INL hardware. Simple UNIX permissions would be set on those binaries to prevent casual copying of those files while still allowing users to execute applications and obtain results. While this method worked for the short term, it created several new maintenance problems and exposed the difficulty in using many of the INL's advanced tools with incomplete documentation, test problems, or examples.

In Fiscal Year 2020, Department of Energy’s Office of Nuclear Energy (DOE-NE) established the Nuclear Computational Resource Center (NCRC) at the INL with the purpose of streamlining the licensing and remote access of DOE-NE simulation software. The goal was to manage license requests and the end-to-end pipeline of getting laboratory software to users. Additionally, the program has worked to resolve existing deficiencies uncovered by users with access to the pre-built binaries created in the previous year. The Code Oversight Group (COG) discovered that many licensees would apply for this new, lower-rigor access mechanism only to request full source (Level 4) access later. The developers and users were provided similar feedback: without sufficient or, in many cases, any documentation or examples, the remote access was unusable. Work on the remote access workflow related to creating binaries and the scheduling and rotation of builds was all performed by this team and is covered in this report. Work-scope associated with workflow streamlining was handled by NCRC and is beyond the scope of this report.

Efforts continued into Fiscal Year 2021 to create a streamlined, secure installation capability for all MOOSE-based, NEAMS-developed software. The goals were to allow for local installation of binaries for satisfying the requirements of access levels 2 and 3 in a secure manner. To achieve this, a new installation capability was added to MOOSE. Using this new installation capability, the Conda package manager is then extended to securely distribute pre-compiled, binary executables. Continuous Delivery (CD) capability was added to the Continuous Integration, Verification, Enhancement, and Testing (CIVET) Continuous Integration (CI) system to enable automated builds of these pre-compiled packages on a scheduled basis. In addition, the MooseDocs documentation system in MOOSE was enhanced to facilitate the installation of automatically-built documentation. Together these enhancements form a simple, secure, Internet-based delivery of applications.
to nuclear vendors and other partners. The following sections detail the development of this distribution capability.
MOOSE is a framework for creating multiphysics applications. The framework is modular and extensible and has inherent capability for coupling simulations across temporal and spatial scales. Much of the NEAMS program’s focus relies upon MOOSE since advanced reactor designs often contain features that require coupling across physical systems. The MOOSE architecture enables direct source development and deployment, which includes the ability to couple applications together. This is accomplished through minor changes to the build system and results in a new combined application. With this design in mind, a similarly generic solution for installation of applications was needed.

Software requires documentation to guide users and often includes examples to aid the user in getting started. A known deficiency for MOOSE-based applications, when access was provided via a binary executable, was the inability of a new user to understand how to begin. Limited access users did not gain access to documentation or the application’s test suite. In particular, the test suite is the best way to verify the software is functional and provides a myriad of operational examples to aid in learning the application.

The agile development process of MOOSE, in union with the need for on-demand functionality from researchers, has driven the framework to be distributed as source code with direct compiling occurring for each user. When distributed as source code, the framework and applications are compiled in place, with the final binary executable located alongside the source code it was built from. This is considered a “non-installed” state, as the binary has not been placed where the operating system or other users can access it without having access to the source code. To enable the graded access levels mentioned above, it was necessary to develop a formal way to deploy and provide access to MOOSE-based applications without the need to compile directly.

A formalized installation process was developed in this work that allows all moose-based applications to be installed using a standard procedure that includes documentation and the test suite for applications. In this instance, “installing” means placing the compiled application (and related files) in a separate location from the source code. Achieving this functionality required overcoming a few obstacles—shared library linking, file path handling, testing, and identifying and copying critical files. Each of these issues is discussed further in the following sections.
2.1 Shared Library Linking

When compiling an application from source code to create a binary executable, there are three primary outputs: the executable, static libraries, and shared libraries. A library is a set of compiled code that executables can ”link to” to run the routines contained inside. Static libraries are linked when the executable is compiled, while shared libraries are linked ”dynamically” when the operating system runs the executable. Shared libraries are preferred due to reduced memory and disk space usage.

MOOSE-based applications utilize shared libraries for both framework and application functionality. In order to support deployment, it was necessary to change how these applications are linked to allow for the compiled application to execute in a different directory from where it was compiled. This was accomplished using platform-specific tools—install_name_tool for macOS and patchelf for Linux-based systems. These tools modify the shared libraries search path to correspond to the installed location. This capability was integrated into the MOOSE build system and work automatically for all MOOSE-based applications.

2.2 File Path Handling

MOOSE includes functionality that provides the ability to read, write, or modify files. These capabilities are utilized for meshes, solutions, post-processed data, and material data. Many of these data types have built-in directory paths for the simplification of input files. Traditionally, these paths were based on the location of the source code.

However, for a MOOSE-based application to be successfully installed, the application must be able to handle data files residing in non-standard locations. This required modifying MOOSE to locate files relative to the currently executing application binary and to detect whether it is running in an installed or non-installed configuration. It was also necessary to update the build system to install the executable, libraries, configuration, and other special files.

2.3 Testing

Several Python utilities have evolved to become an integral part of the MOOSE ecosystem including a graphical user interface, animation scripts, plotting scripts, and critical pieces of the build
system. One of the most important Python utilities is the testing tool, which enables MOOSE-based applications to develop a suite of tests, ensuring that applications continue to work correctly as the framework and applications are developed. The testing tool plays a pivotal role in the NQA-1 development process, followed by the framework and applications. It was designed for execution of tests in a non-installed build. Thus, modifications were required to be able to run a MOOSE-based application in either an installed or non-installed configuration.

Additionally, to provide an integrated experience for users, the ability to execute tests was integrated into MOOSE-based applications directly. Previously, the testing tool was executed directly by users. However, with installed applications, it would be unclear where the testing tool is located. A newly developed capability allows MOOSE-based application users to supply a special option, \texttt{--tests}, to execute tests, as shown below:

```
[username@host]$ moose_test-opt --tests
```

By default, tests that exist in the current directory are executed. If no local tests are found, then any relevant installed tests are executed. All available testing options from the testing system are available from the executable using the \texttt{--tests} option, as follows:

```
[username@host]$ moose_test-opt --tests -h
       [--ignore [caveat]] [-j [int]] [-s] [-c]
       [--color-first-directory] [--heavy] [--all-tests] [--g GROUP]
       [--not-group HDT_GROUP] [--dirfile [DIRFILE]] [-l LOAD] [-t]
       [--longest-jobs LONGEST_JOBS] [-s] [-i INPUT_FILE_NAME]
       [--libmesh-dir LIBMESH_DIR] [--skip-config-checks]
       [--parallel [PARALLEL]] [--n-threads THREADS] [--recover]
       [--recoversuffix RECOVERUFFIX] [--valgrind]
       [--max-fails MAX_FAILS] [--longest-jobs LONGEST_JOBS]
       [-c] [-d] [-parallel-mesh] [--distributed-mesh]
       [-max-fails MAX_FAILS] [-r REG_EXP] [--failed-tests]
       [-check-input] [-no-check-input] [--spec-file SPEC_FILE]
       [-g] [-no-report] [--show-directory] [-e directory]
       [-f FILE] [-x] [-sep-files-ok] [-a] [-include-input-files]
       [--teetharness-unitest] [--yaml] [--dumpe]
       [--no-trimmed-output] [--no-trimmed-output-on-error]
       [--results-file RESULTS_FILE] [-pbs name] [-pbs-pre-source]
       [-pbs-project] [-pbs-queue] [-pbs-node-class]
       [-queue-clean-up] [-queue-project] [--queue-queue]
       [queue-queue]
```

In this manner, the installed, built-in testing functionality can be used to verify correct exe-
cutable behavior (i.e., by running the installed tests) and run user-created tests.

The process of running tests generates files, thus is problematic for users running installed tests as they will often not have the necessary file system permission to add or modify files in the installation directory. Hence, the ability to copy the tests into a location with necessary permissions was included in the testing tool. This is invoked using the --copy-tests option, which creates a copy of the installed tests for an application into a local directory (named [app_name]_tests). The tests can then be run using --tests option. For example, the copy and test execution can be performed as follows:

```
[username@host] $ moose_test-opt --copy-tests
Tests successfully copied into ./moose_test_tests
[username@host] $ moose_test-opt --tests
```

This capability for copying installed tests partially alleviates the issue mentioned earlier where Level 1 (executable only access) users were previously without files that helped them get started using the codes. Now, running with the --copy-tests option will provide those users with an extensive suite of example input files and geometries that are known to work with the binary they have access to.

### 2.4 Installed Files

Installed MOOSE-based applications follow a standard file structure. The files are separated into the standard Unix bin, include, lib, and share directories. All application-specific installed files either contain the application name or are inside a sub-directory containing the application name. This will help facilitate installing and using several MOOSE-based applications in a single environment. For example, the following file structure is created for the installed open-source Mastodon application:

```
|-- bin
| |-- exodiff
| |-- mastodon-opt
```
```bash
|-- moose_test_runner

|-- include
  |-- moose
    |-- MooseConfig.h

|-- lib
  |-- mastodon
    |-- libhit-opt.0.dylib
    |-- libmastodon-opt.0.dylib
    |-- libmastodon_test-opt.0.dylib
    |-- libmodule_loader_with_tm_st_con-opt.0.dylib
    |-- libmodule_loader_with_tm_st_con_test-opt.0.dylib
    |-- libmoose-opt.0.dylib
    |-- libpcre-opt.0.dylib
    |-- libstochastic_tools-opt.0.dylib
    |-- libstochastic_tools_test-opt.0.dylib
    |-- ...

|-- share
  |-- mastodon
    |-- doc
    |   |   +-- ...
    |   |-- test
    |       |-- mastodon-opt -> ../../../bin/mastodon-opt
    |       |-- testroot
    |       |   |-- ...
    |   |-- ...

  |-- moose
     |-- moose_config.py
     |-- python
         |-- ...
```
3 THE CONDA PACKAGE MANAGER SYSTEM

Conda is an open-source Python-based environment and package manager, supporting several versions of Linux, Apple MacOS, and Microsoft Windows. It is designed to work with developer environments as a tool for installing binary applications. It can be installed to any location where a user has write access, therefore it does not require elevated privileges for use. This makes Conda user-friendly when operated alongside other similar products or within existing developer environments.

One of the principle capabilities within Conda is a dependency resolution system. This powerful system will automatically resolve all of the dependencies (other libraries/executables) needed by an application when it is installed. Resolving the dependencies involves locating suitable versions of the dependencies, downloading, and installing them. This powerful system simplifies the installation of software with complex dependencies.

MOOSE-based applications have complicated dependency chains. They not only depend on MOOSE but also other MOOSE-based applications and libraries developed within the national laboratory system such as PETSc[2], Hypre[3], and MPI[4]. To simplify installation, Conda was chosen as the tool for installing MOOSE-based tools and plays a pivotal role in package distribution and environment management.

3.1 Conda Deficiencies

Conda is primarily designed to distribute open-source applications. Pushing the tool to deliver controlled application revealed several deficiencies. While Anaconda [5], a commercial entity that supplies Conda-based services, supports security for Conda packages via their cloud services, it is not FedRAMP [6] approved. It was therefore decided to establish self-hosted, private servers for distributing protected packages on behalf of the national laboratory system.

Conda is further hampered by its inability to handle the permissions necessary for hosting multiple controlled packages. Additionally, Conda does not have a server-side component where these deficiencies could be addressed. Conda expects packages to be available as a directory service, providing difficulties for granular control of applications. Several readily available products could provide directory services, including local files systems, Server Message Block (SMB), and Network File System (NFS).
work File System (NFS), or the Internet gold standard Secure Socket Layer (SSL) over hypertext transfer protocol. After considering the requirements, a well-established set of secure software (Apache with Lightweight Directory Access Protocol (LDAP) authentication) was leveraged to control access to the packages.

Conda was not intended to decline users access to packages, and it fails in unrecoverable ways if subjected to such restrictions. Users could receive erroneous error messages if communications are interrupted due to access restrictions, as opposed to normal network failure mechanisms. The mechanism used to work around this issue will be described in 3.2.

Another deficiency of Conda is poor dependency resolve time. Conda now contains millions of packages, and working through the dependency chains for complex software can take several minutes. To work around this, we will use ”Mamba” [7], a rewrite of the Conda front-end that was explicitly developed for faster execution.

```
$ conda install mamba
$ mamba install moose-tools moose-libmesh
```

Utilizing Mamba, dependency resolve times drop from 40 minutes to less than two in our testing. Mamba was created with the scientific community in mind, a community that commonly creates large complex packages. Due to the lack of an authentication mechanism, as discussed in 3.2 it cannot be used at this time. However, contact has been made with the Mamba developers, and improved authentication capabilities can be added in the future.

With these issues, the best course of action was to encapsulate the authentication mechanism and the Conda interactions inside of a new tool. The ”NCRC utility”, a simple wrapper script to accomplish these tasks was developed and open-sourced as part of this work. This utility handles access to controlled applications by adding an authentication mechanism. The utility also provides the flexibility of changing the access mechanism from Conda to Mamba without impacting users. The authentication mechanism employed within this utility is detailed in the next section.
3.2 Conda Two-factor Authentication

For a web application to be externally facing, INL requires the use of Multifactor Authentication (MFA)—a mechanism that utilizes two or more authentication methods before a user is granted access. For this MFA implementation, Two-factor Authentication (2FA) agent for Apache webserver was used. The first authentication token is a unique 8-digit “pin” that each user creates when setting up an INL HPC account. The second is a hardware or software Rivest-Shamir-Adleman (RSA) token which provides a cryptographically secure, one-time code. The code is a 6-digit number that changes every 60 seconds. The combination of pin and token is provided to the RSA Apache agent by the NCRC Utility, and access to the user’s approved applications is granted.

The 2FA was implemented with Conda through the use of a cookie captured during a successful authentication attempt by the NCRC Utility. The utility is available publicly from a self-hosted Conda channel named ”idaholab.” The cookie allows for several minutes of connectivity, after which the user will have to authenticate again to run additional commands that require access to the secure servers.

The NCRC utility wraps Conda to augment its existing capabilities. When maintainers wish to add a new Conda channel protected by 2FA, a new channel prefix otherwise unrecognized by Conda is used, ”rsa://“. When channels with these prefixes are encountered, the utility handles authentication and hands a transformed URL back to Conda using supported SSL syntax along with an authentication token or cookie. Conda then processes the request without any further modification. With the cookie in place, Conda behaves like it would with any other server. Access control is handled by existing LDAP permission technology, which allow for fine-grained control without the development of yet another system.

3.3 End-User System Interaction

With much of the technical detail for how these tools work discussed in the previous sections, a demonstration of how end-users make use of these tools is shown here:

The NCRC utility itself must be installed on each end-user system. Fortunately this is easily accomplished by using Conda directly and must only be done once:

```
$ conda install ncrc
```
An example of installing a package using the NCRC utility:

```
$ ncrc install ncrc-bison
Username: johnsmith
PIN+TOKEN: *******
```

If authentication was successful, the NCRC utility will use the native Conda installation routines and present the user with the appropriate activate command:

```
# To activate this environment, use
#
# $ conda activate ncrc-bison
#
# To deactivate an active environment, use
#
# $ conda deactivate
```

The user will then activate the environment as shown above and may use the NCRC application:

```
$ conda activate ncrc-bison

# Run the local binary on the end user’s system
$ bison-opt -i <input file>
```

Regular updates may then be obtained by using the following command:

```
$ ncrc update ncrc-bison
```

Information on how these regular updates are produced and delivered is covered in Section 4.
4 CIVET: CONTINUOUS INTEGRATION SYSTEM

The NEAMS program is currently developing more than ten simulation tools based on the MOOSE Framework. Each of these applications and the framework has a dedicated development team which is modifying the code daily to achieve program objectives. As the code changes it is critical that the applications continue to function correctly and interoperate with both the framework and the other applications. In addition, the rigorous NQA-1 software development process must be followed. To ensure this development process works smoothly, a software system called CIVET continuously analyzes every change proposed to each application and runs tens of millions of tests every week. For this work the CIVET system has been augmented for automated packaging of applications, a capability widely known as CD. Simply put, CD is use of automated pipelines and distribution mechanisms to deliver regular application updates all the way to the end user.

4.1 Continuous Delivery Capability

To facilitate packaging MOOSE-based applications on a regular automated basis, it was necessary to add new scheduling functionality to the CIVET server. Many solutions exist for running processes on regular schedules. Rather than invent a new syntax for this purpose, a commonly used syntax in the Linux world was adopted. This syntax is is part of the “cron” system [8]. A few cron syntax examples and the associated scheduled times are provided in Table 1.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 09-18 * * *</td>
<td>Run every hour, from 9:00 AM to 6:00 PM, everyday</td>
</tr>
<tr>
<td>30 23 * * 2-6</td>
<td>Run at 11:30 PM, every weekday</td>
</tr>
<tr>
<td>30 07, 09, 13, 15 * *</td>
<td>Run at 07:30, 09:30, 13:30 and 15:30</td>
</tr>
<tr>
<td>*/7 3-6 2, 3, 4 4 *</td>
<td>Run every 7 min. from 3-6 am on April 2nd, 3rd, and 4th</td>
</tr>
</tbody>
</table>

A preexisting Python library was used for interpreting this syntax. To specify an event to occur periodically, the “scheduler” option is added to the corresponding recipe file. A CIVET recipe is a set of commands that are executed based on some trigger, such as building and testing a set of source changes when a set of changes is proposed by an application developer. The inclusion of
the "scheduler" option automatically promotes the recipe from otherwise being strictly "event-based" to "scheduled" as well. This capability allows for maximum reuse of existing recipes. For instance, the same recipe may be used to build a binary containing the very latest changes from a developer as well as producing regular tested updates for delivery to an end user.

```plaintext
[Main]
build_user = moosetest
name = schedule1
scheduler = 30 23 * * 2-6

[Test]
script = scripts/run_cmd.sh
abort_on_failure = True
allowed_to_fail = False
RUN_CMD = run_test_script.sh
```

A web-based dashboard was created on the CIVET server for monitoring these scheduled jobs as well as initiating manual runs. This dashboard allows for clear differentiation between the event-based and scheduled jobs. Scheduled jobs, by definition, will be performed on a subset of the commits in the repository where event-based jobs have previously been triggered. It is unlikely a job that has previously passed as an event based trigger will fail, but it is still a possibility; so the dashboard allows developers to monitor those cases do determine the conditions leading to the failure.

![Cron-Scheduled Recipes](image)

**Figure 1:** Screenshot of the new periodic job scheduling dashboard for CIVET.

### 4.2 Automated Builds and Version Control

Using CIVET and GitLab, there will be two builds involved: scheduled and event-based. The scheduled builds are intended for binary distribution where users do not have access to the repository. These builds are for users with binary-only levels of access; the event-based builds are for testing and CI.
The flowchart in Figure 2 describes the process of creating a Conda application package. Applications are under continuous development (i.e., multiple event-based builds daily). Creating a package for every change would over populate a Conda channel with packages and confuse users with large numbers of available packages. Instead, for unofficial intermediate releases, the scheduling feature in CIVET, as discussed in Section 4, is leveraged to create packages on a predetermined schedule—in this case, weekly. When CIVET triggers a package creating job, the latest available version of the software for that date is used.

Figure 2: CIVET Conda package creation flowchart.

Figure 3: INL Testing Architecture.
Figure 3 shows INL’s testing architecture. INL maintains several servers representing different operating systems and configurations for testing all MOOSE-based codes on a variety of architectures. Each of these servers communicates with two CIVET servers to perform CI operations. One of these servers is primarily used for OSS applications while the other handles internal, export-controlled applications. Both can respond to events (i.e., pull requests, pushes, and merges) from repository hosting services, including the time-based events generated internally for building new packages for deployment discussed in this section.
5 MOOSEDOCS: APPLICATION DOCUMENTATION

Documentation is a critical component for users to understand the use of an application. MOOSE-based applications utilize a common system for documentation: MooseDocs. MooseDocs is a stand-alone Python tool within MOOSE that includes the ability to convert Markdown, a common text formatting language, to HTML, LaTeX, and presentations. The entire system is extension-based and includes a parallel, multi-page parsing system. By design, creating new input and/or output formats is possible, and applications can create custom extensions to meet specific needs. The system has allowed for all applications to create documentation using a single system, enabled documentation to become a necessary part of development, and is utilized to meet the Nuclear Quality Assurance (NQA) software quality standard [9]. With respect to deployment of application documentation, two features were added to the system: (1) documentation was setup to be installable and (2) capability was added for controlling in-document source code access.

5.1 Installable Documentation

As detailed in Section 2, the ability to install applications is critical for deployment of NEAMS-developed software. This new installation capability also was extended to include documentation as generated by MooseDocs. The process for installing the documentation is straightforward. Foremost, the main documentation for an application is a static website; as such, once created, it can be viewed from any system regardless of internet connectivity. It is also portable; all links within were updated to be relative to support installation. If an application is compiled, the documentation can be created using the commands in Listing 5.1. This command reads the Markdown files within the repository, converts them to static content, and writes to the desired location. During installation of an application, this command is automatically executed.

```bash
1. cd docs
2. ./moosedocs.py build --destination path/to/installed/docs
```

A key feature of MooseDocs is the ability to extend Markdown syntax. One extension, in particular, the “Listing” extension, included with MOOSE allows documentation to incorporate code snippets that are extracted directly from the application source. This allows documentation to reference actual code, helping ensure it remains current. For example, the listing extension can
Figure 4: The HTML rendered from the “listing” Markdown syntax shown in Listing 1 can be used to include portions of an input file. For example, Listing 1 is the custom Markdown syntax that will create a code listing in the documentation as in Figure 4.

Listing 1: Example use of the listing extension to import input file content into documentation.

By default, the contents of the entire file are loaded into the link shown at the bottom of Figure 4. This link will open a dialog that will show the full contents of the linked file. However, this feature may be disabled globally for the application documentation. In support of optionally including/excluding code, the “modal” extension was developed. This extension provides a single control point for including source code. As such, toggling off the “show_source” option within the extension, as shown in Listing 2, disables all complete source code.

Listing 2: Command for MooseDocs to build and install documentation for an application without including complete source code.

As NEAMS applications continue to gain use throughout the nuclear industry and beyond, the importance of quality, complete, and approachable documentation will be crucial for success. MooseDocs provides the critical ability to create, test, and maintain documentation as a part of application development. This allows for the integration of the documentation process with the application development process, creating a system that encourages continuous improvement of
documentation in a sustainable manner during all phases of a project.
6 CONCLUSIONS

This report summarizes the technical achievements made toward enabling simplified, secure distribution of NEAMS-developed codes. An installation capability that works with all MOOSE-based applications was created, allowing installed binaries to be placed in standard operating system paths where multiple users can then have access. The installation capability was then leveraged to create pre-compiled, binary packages that work within the Conda package management ecosystem. Enabling secure deployment, the NCRC utility was created which augments Conda to enable secure, two-factor authentication for accessing application binaries. The CIVET continuous integration system was extended to also perform continuous delivery through the automated creation of application packages. Finally, the documentation system within MOOSE (MooseDocs) was modified to allow documentation to be installed alongside applications.

A robust and secure distribution mechanism for nuclear applications is critical to the success of the NEAMS program. The work described in this report, combined with the deployment of the NCRC, represents a large step forward toward simplified, verified access to reactor simulation tools.

6.1 Future Work

Future work will continue with the testing and deployment of the additional multiphysics applications. While Conda is fully capable of handling any amount of dependencies, these larger applications require a large number of dependencies, and users making use of different combinations of applications may need several potentially conflicting copies of individual applications. To be clear, this is not a problem for end users since Conda is designed precisely to handle these kinds of conflicts in separate environments—a primary driver for selecting Conda as a deployment mechanism. Additional investigation is also occurring in improving the initial setup of new users, further lowering the barrier of entry. Continued adoption of the newer Mamba utility for the purposes of installation optimization is ongoing. Mamba is considered a drop-in replacement for Conda but as a newer utility, additional testing of the tool to see if there are any issues is needed.

Level 3 access is still being researched as this report is being written. Much of the infrastruc-
ture created for the Level 2 binaries is planning to be reused for Level 3 access, the differentiating characteristic being the inclusion of the header files necessary for end-user development. One of the remaining challenges with implementing Level 3 is the ability to deliver a complete build system through Conda. While this is theoretically possible with Conda, there are practical challenges and complications with delivering a build environment to end users on various platforms. Level 3 access, while an important capability in theory (the ability to protect proprietary algorithms that might appear in source implementation files), is only partially realized with the architecture of MOOSE and the limitations of the C++ language. MOOSE is “templated” (a C++ language feature) to support the plethora of advanced features. Template algorithms must appear in header files, so they can be compiled based on their specific end use. This ultimately means Level 2 or lower access should be preferred when there are any concerns about sharing any amount of source. Likewise, any developer of a MOOSE-based application would benefit from having full source (Level 4) access.
References


