Stewardship of the MOOSE Ecosystem

Derek Gaston, Idaho National Laboratory, derek.gaston@inl.gov

Casey Icenhour, Idaho National Laboratory, casey.icenhour@inl.gov

Fande Kong, Idaho National Laboratory, fande.kong@inl.gov

Logan Harbour, Idaho National Laboratory, logan.harbour@inl.gov

Guillaume Giudicelli, Idaho National Laboratory, guillaume.giudicelli@inl.gov

Introduction

For the past 14 years, Idaho National Laboratory (INL) has led the development of the open-source Multiphysics Object-Oriented Simulation Environment (MOOSE) simulation framework [1, 2]. At its heart, MOOSE is a finite-element and finite-volume framework built to accelerate the development of parallel, multiscale, multiphysics simulation tools. Written in C++, it has been adopted for simulation of a wide range of physics and engineering problems, including nuclear reactor simulation [3], earthquake predictions [4], bio-inspired vascular networks [5], frontal polymerization [6], mining [7], additive manufacturing [8], and material science [9]. To demonstrate the reach of the MOOSE project, over 500 papers have been published using MOOSE, the primary two MOOSE papers have garnered over 1,000 citations, and the Department of Energy Office of Nuclear Energy (DOE-NE) Nuclear Energy Advanced Modeling and Simulation (NEAMS) program utilizes it as the basis for its advanced reactor modeling and simulation tools. Today, there is an ecosystem of over 30 actively developed MOOSE-based tools within the DOE laboratories that rely on the continuous improvement of the framework.

From the beginning, the MOOSE project has maintained rigorous software quality and development processes that have grown with the project and helped guide the stewardship of the ecosystem. Without these processes and management practices, the project could not have seen its current level of success. The very first commits to MOOSE reference the first design tickets (kept in Trac at the time – now in GitHub) and, to this day, every pull request merged into MOOSE does likewise. Continuous integration (CI) – still fresh in 2008 – was utilized from the beginning of the project and continues to play a prominent role, with continuous delivery (CD) joining it along the way. From the beginning, the code was created in a self-documenting way, blossoming over time into a documentation system called MooseDocs that automatically and continuously rebuilds the websites and documentation for MOOSE and MOOSE-based tools after every merge, ensuring accurate, up-to-date information. The testing system (MooseTest) and CI/CD software (CIVET) are custom developed along with a host of other tools, including a graphical user interface (Peacock) and a Make-based build system (MooseBuild). Altogether, these capabilities have eclipsed the original goal of a C++ simulation framework and have become an end-to-end platform for multiphysics software development.

Of particular importance to the stewardship of the MOOSE ecosystem is the comprehensive CI/CD system. The MOOSE project aimed to avoid typical problems in scientific library/software development – namely, the "integration/upgrade hell" associated with disjointed library/application development. This is where applications get stuck using old versions of libraries because they are unable to cope with changes made to the library while the application is under development. For this reason, MOOSE and all MOOSE-based tools were originally kept within the same Subversion (SVN) repository. This enabled CI across the entire ecosystem and the applications to be updated simultaneously (within the same commit) with changes being made to the framework. Having CI across the ecosystem meant that modifications to MOOSE could be instantly checked against all downstream applications and flagged (and later kept from being merged) if those changes caused test failures.

In 2014, MOOSE made the jump from internal DOE project hosted with SVN to external open-source project hosted on GitHub. While the tickets and commits were easy to migrate, it was unclear how to continue doing CI across the entire ecosystem. After trying multiple existing solutions, we eventually

developed our own CI tool [10] that later became the open-source project, CIVET [11]. CIVET plugged into the GitHub and GitLab webhooks to watch for new pull requests (changesets) and worked with a small amount of hardware on loan from the INL HPC to automatically test the changes against all downstream applications. Over the years, this effort has grown to automatically test 30+ repositories against each other as each library/application is developed. Today, the system utilizes 2,500 dedicated processors to run over 20 million tests per week and helps shepherd over 80 pull requests a week into the various codes.

This commitment to quality and automation has permeated the MOOSE ecosystem from the beginning and has informed our choices of how to grow the community and expand capability. Modularity, extensibility, and pervasive testing allow hundreds of researchers to work productively with the codes and framework every day. The team has responded to a comprehensive selection of questions from the RFI, retaining the original numbering for clarity. These responses, found below, reflect this point of view and provide examples of how large-scale scientific software development ecosystems can cope with complexity and growth.

Response

1. *Software dependencies and requirements for scientific application development and/or research in computer science and applied mathematics relevant to DOE's mission priorities:*

Development of high-performance, parallel, multiphysics frameworks such as MOOSE necessitates the use of many libraries. These range from low-level libraries such as Message Passing Interfaces (MPI) to higher-level capabilities like graphical user interface (GUI) toolkits. Working in C++ provides a lot of flexibility regarding library selection and abstraction, providing a comfortable level of flexibility to change libraries if the need arises. Figure 1 shows the primary layers of MOOSE dependencies.

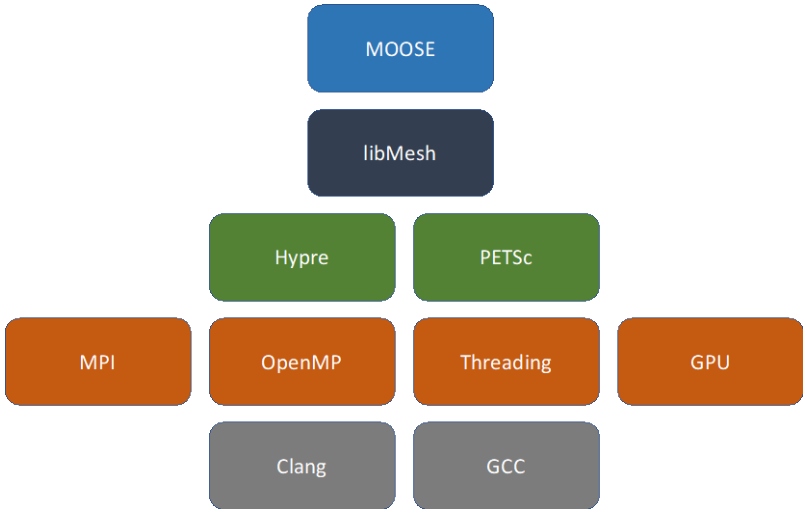

Figure 1: Primary layers of MOOSE dependencies.

The lowest level libraries/tools we utilize are C++ compilers. As far out as we can project, it appears as though development is maintaining its pace for both Clang and GCC, with both rapidly adopting new C++ Standard capabilities. Next is MPI, where the field does seem to be shrinking. Some larger supercomputer companies are still shipping their own MPI implementations, but many of these have been discontinued over the years. That said, OpenMPI, MPICH, and MVAPICH remain active projects, and the MPI standard holds them all to a consistency that allows us to move freely among them. Due to this flexibility, we don't actively track the versioning of MPI or compilers (beyond minimum requirements).

Just above MPI, we have a layer of numerical libraries and solvers, with the two most important being Hypre [12] and PETSc [13,14,15]. Both are primarily developed within national laboratories (Lawrence Livermore National Laboratory (LLNL) and Argonne National Laboratory (ANL), respectively), have existed for more than 15 years, and are still in active development. They have continued to evolve their APIs over the years, meaning that we are used to keeping up with the changes they make. One potential risk is that both libraries are pushing for more GPU capability, causing more extensive changes to their APIs than usual. Still, it is good that they keep up with computing architectures to enable use of next-generation machines.

The libMesh [16] finite element library sits just above the solvers and provides discretization, finite-element infrastructure, and parallelization. libMesh is a C++ library that abstracts away much of the detail of parallel vectors and solvers, allowing for significant flexibility to change the supporting libraries if needed. However, libMesh itself is incredibly intertwined with the MOOSE library, meaning that MOOSE cannot be separated from it. Therefore, continued libMesh development (to maintain the software and add required features) is critical to the MOOSE project. While several MOOSE developers are also libMesh experts, we have also chosen to directly employ the lead libMesh developers for the last ten years, recently hiring the primary maintainer. In this case, the risk was too significant to allow the possibility of losing the libMesh library.

Several smaller libraries are utilized by MOOSE, including utilities for regular expressions, XML, and JSON reading/writing, directory traversal, and unit testing. These libraries are small enough and have compatible enough licenses to simply copy their source into a "contrib" directory directly within MOOSE. There is no need to update them or keep up with APIs due to their small, targeted nature; therefore, the risk is minimal.

One potential risk is that another programming language that is less compatible with C++ becomes incredibly popular for numerical library developers, causing activity in libraries we can use to fall away. So far, this hasn't been the case. For instance, many AI/ML libraries (such as PyTorch [17] and TensorFlow [18]) are being developed to be utilized by Python. Still, they also offer backend libraries that are easy to link to and use from C/C++. Even Julia has a C interface, ensuring that our C++ library development can continue to capitalize on numerical library developments for many years to come.

Additional dependencies come from the necessary development and input file creation support in modern integrated development environments (IDEs) and text editors. Custom MOOSE plugins in the Atom IDE improve the user experience for developing C++ objects and MOOSE input files. Existing Atom community plugins enhance cohesion with Git version control, Python, etc. Using the concept of language servers, a similar development is currently undertaken for the Visual Studio Code IDE. Creating these plugin capabilities is typically a one-off, single-person project, and their maintenance is difficult when deprecation occurs.

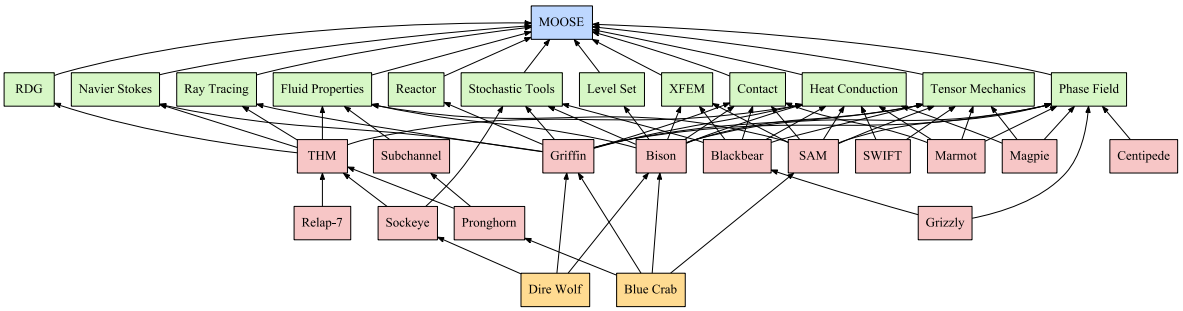

Figure 2: Dependencies within a subset of the MOOSE ecosystem.
Blue: MOOSE, green: physics modules, red: applications, orange: coupling applications.

MOOSE itself is a dependency for all downstream applications within the MOOSE ecosystem. Due to the high modularity of MOOSE-based codes, some of these applications also depend on several other applications, creating internal dependency chains, as shown in Figure 2. Changes in MOOSE could break any part of the chain, so pull requests to MOOSE are tested by running the entire test suite of the whole ecosystem in a single configuration. Note that directly "pushing" to a branch is programmatically forbidden in all repositories, requiring all changes to go through the pull request mechanism, and therefore testing, before they can be merged. As shown in Figure 3, if the pull request is accepted, it is placed in a "next" branch. This branch is tested against a larger number of configurations of the whole ecosystem (for example, with a different version of PETSc). If it passes, it moves to the "devel" branch. If it fails, the MOOSE development team and application developers collaborate to make the applications compatible with the merged changes. GitHub/GitLab dramatically enhances the process, facilitating communication and code iteration. Once a MOOSE change reaches the main branch, a submodule update is automatically attempted for downstream applications within the ecosystem. If unsuccessful, a second round of collaboration occurs on the affected applications, and changes are made with application developers. If an issue is identified with the testing process (i.e., a gap in coverage), changes are made to earlier rounds of the CI/CD process to ensure proper treatment of rare cases. This process is similar for internal and external applications. All internal applications and a subset of external ones (e.g., those submitted directly to the MOOSE development team for monitoring) are tested. This level of coordination and collaboration, underpinned by automated processes, is key to stewardship of the ecosystem, ensuring that all developers and users can continue to operate unhindered as development progresses.

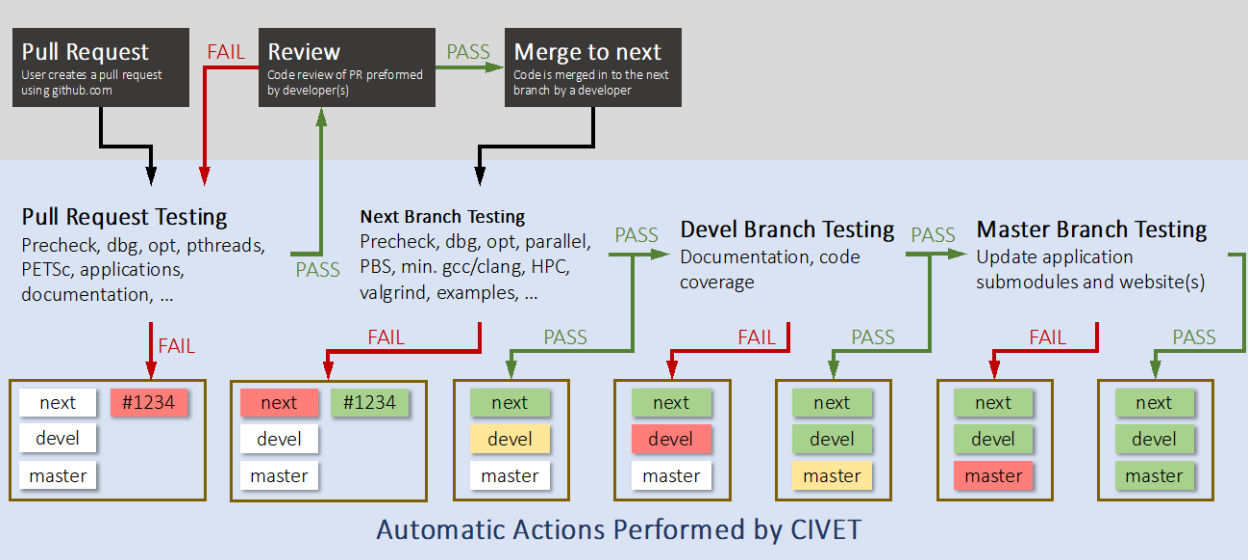

Figure 3: Flowchart of the review and merge process for pull requests.

An early focus was placed on API stability to minimize dependency risk to MOOSE-based applications. The MOOSE APIs are the principal interface applications use to connect to MOOSE objects and systems. When changes occur to APIs, current development practice dictates "deprecating" it (that is, discouraging the use of a previous version of code over another that supersedes it in capability or function) over a set timeframe – sometimes comprising several months. This gives application developers ample time to adjust their code. Deprecations within input files can also be checked using MooseTest options, and CIVET testing dashboards can be configured to show deprecation warnings. Test input file consistency has been achieved for a given version of MOOSE (and applications) and dependencies by storing and testing input files in the same repository. Thus, coupled with the CI systems, there is assurance that a given input file will work as expected whenever it is needed. More recently, MOOSE-based projects have risen that serve as benchmark libraries, where a set of input files are tied to various subsets of MOOSE, MOOSE modules, and MOOSE applications. As these inputs are stored separately from the applications they depend on, more care must be taken when proposing significant code changes that affect these projects. A "two-way" testing paradigm is adopted between the project itself (checking base integrity of the input files) and more intensive checks of those inputs within upstream applications. One example is the Virtual Test Bed repository, discussed in more detail in the next section on software integrity.

2. *Practices related to the security and integrity of software and data:*

Traceability and integrity are paramount when building software for licensing nuclear reactors. The regulations surrounding nuclear reactors require nuclear safety software to follow the ASME Nuclear Quality Assurance (NQA-1) software quality process. NQA-1 certification requires extensive documentation of the development process. Modifications of the code must reference a design document, be featured in a test, and reference an issue. The design document is usually the user documentation for the modified (or created) object. The issue serves as a forum for developer and user stakeholders to discuss the proposed change and explains why it is being suggested and its impact. This is enforced automatically by the test suite on pull requests.

Reproducibility of results is achieved using the git submodule system. The MOOSE commit hash is uniquely tied to a set of submodule commits for its dependencies, including libMesh and PETSc. The code can then be recompiled in the exact configuration that a simulation was performed at any point in the

future, assuming MPI distributions and other dependencies can also be reverted to their former state. In practice, extensive backward compatibility in libMesh, PETSc, and other dependencies is leveraged to avoid reproducing the entire environment. The process is similar for MOOSE applications, which are also tied to a unique MOOSE commit via a git submodule in their repository. The process is facilitated by having the documentation for the framework and applications checked into the repository with the codes, making the documentation revertible to how it was at the commit of interest. This is also true of all the other tools making up the platform, including the build system and GUI – all can be perfectly reverted to a point in the past. Past investigation of previous MOOSE builds can also be performed using the testing infrastructure, which records and logs metadata for all tests and builds, permanently storing proof that any code changes satisfied the testing requirements.

Their respective test and validation suites ensure the integrity of the downstream applications. These range from unit tests to complete calculations compared against experimental data. It is further secured using the open-source Nuclear Reactor Innovation Center (NRIC) Virtual Test Bed (VTB) [19]. This repository holds numerical benchmarks, full-scale multi-application calculations, and other target analyses relevant to nuclear science and engineering. Each model in the VTB should always be functional, as they are customer-facing models. This is ensured using two-way testing. When each model is modified or added to the VTB, it is tested against the relevant applications using syntax checking and limited regression tests. Regressions tests are limited to comparing a few integral and local data points over a few time steps because of the cost of these simulations. In parallel, the relevant VTB inputs are included in the test suite for every pull request in MOOSE or any MOOSE-based application. Pull requests may not be merged to any code before a patch is proposed for the VTB. Git submodules are also used in the VTB to record application versions that are guaranteed to run the input files at any point in time.

It is essential to protect the privacy of applications/input files/developers while still enabling the large degree of cooperation, collaboration, and testing necessary within the MOOSE ecosystem. To enable this, CIVET can restrict the view of testing/outputs to only those authorized to work on those projects. It utilizes GitHub/GitLab authorization technology to directly authenticate for test dashboards. Even in the case of testing downstream applications due to an upstream library change, if the upstream developer does not have access to the downstream application, the upstream developer will simply see a “pass/fail” mark for that application’s tests, without the ability to see any details. Failure, in this case, will then require communication and cooperation with the downstream application to either create a patch that will allow that application to handle the changes or modify the upstream patch to enable the application’s tests to pass.

3. *Infrastructure requirements for software development for scientific and high-performance computing:*

The primary infrastructure for MOOSE is the hardware and software surrounding the repositories, testing, and distribution. The CI hardware needs far outweigh the hosting/distribution needs. To meet the integration needs of the code base, the MOOSE project employs a diversity of hardware, from laptops, workstations, and powerful servers to high-performance clusters consisting of thousands of CPU cores for daily tests. The diversity of hardware hosted locally at INL is essential to ensure that the code base works smoothly on the platforms required by the large user community with diverse backgrounds. The reliability and accessibility of the testing hardware is critical, with downtime significantly disturbing the software development process and impacting the whole ecosystem. Many potential factors can deteriorate the hardware accessibility, including network outages, power connections (within the data center or to the grid), or unexpected hardware issues. The existing testing capability is constantly under

pressure from the growing testing need. Regular hardware upgrades are required within the dedicated pool of testing hardware, as are local cluster resources.

One potential solution to these issues could be cloud computing, taking advantage of the near-infinite scalability and flexibility. However, cloud computing works best when utilization is sporadic. After a recent cost comparison, it was clear that local hardware was significantly cheaper due to the near-constant load on our testing hardware. Local hardware also affords extra security due to firewalls and access controls.

The GitHub/GitLab hosting of MOOSE and the broader ecosystem provides an increasing array of features that help the team manage the project. The GitHub platform offers flexible changeset feedback interfaces, allowing interaction with our testing system and human reviewers. CIVET is triggered for every change via the GitHub API. Beyond code, efficient and effective communication between collaborators and users provided by this infrastructure is vital. Users can report a MOOSE bug or ask for a new feature by creating a GitHub "issue," wherein developers will try to understand and clarify the requirement. The project team can also assign a team member who has the right expertise to handle that ticket. Once the developer has a changeset for the ticket with a potential solution, the user can comment on code changes and try the submitted code branch against their use case. Developers rely on dependable feedback from users to improve the code quality. The discussion inherent in this development cycle and GitHub/GitLab-based infrastructure is an asset for the project, especially when it needs to be referred to during later parts of the lifecycle of the software.

Another essential feature of software hosting tools like GitHub and GitLab is allowing the project manager to assign different roles to different developers and users. Some developers have the right to merge a changeset, and others might only be able to comment on the code. Code branches, such as the main release branch, can also be protected from "force pushing" (or forced code modifications). This enhances code security and integrity. A remaining challenge for MOOSE is that GitHub does not currently provide some extended, fine-level controls over user roles. For example, a user cannot simply be given the right to assign reviewers and labels (e.g., as an integration specialist) without giving them some level of write permissions to the code repository, which they might not need. This introduces some inconvenience to the process, but overall, software hosting tools such as these are vital to our software development workflow.

4. *Developing and maintaining community software:*

Scientific software tailored for specific research purposes or only for internal use often includes tailored development and maintenance approaches. The application area or domain almost always influences feature development, verification, and validation. The software development process may be governed by internal development or usage paradigms and held accountable to only a handful of internal hardware and software platforms. This can reduce effort and lend focus to development, maintenance, and support plans. As a scientific software project gains stakeholders and applications with various (and sometimes competing) needs, the largest source of additional developer effort lies in supporting an ever-wider application base on an increasingly diversified number of platforms.

For example, the MOOSE ecosystem currently holds a complex set of more than 30 interdependent libraries and applications (some of which are shown in Figure 2), with platform support over various versions and distributions of Linux and macOS. Changes to the core framework require extensive regression testing across all supported platforms to ensure that code changes do not lead to software breakage while delivering new and improved features in a timeframe that enables the community to perform their research effectively. This leads to millions of tests being run per week for proposed

changes and is untenable for individual developers to perform. However, these additional challenges are not unique to scientific software and are primed for automation. Thus, MOOSE has utilized CIVET, a custom-built, open-source CI/CD tool for the last 14 years.

CIVET automatically tests every code change before it can be merged, tests integrations of multiple code changes, promotes code changes to release branches, builds binary packages, distributes binaries to INL HPC systems, publishes binaries via the Conda [20] package management system (to promote consistency across platforms), and updates documentation on project websites based on the new changes. Automated systems such as CIVET save person-years during development and release cycles, ensuring bugs aren't being introduced over time and across all development branches of the project. However, using a CI/CD tool doesn't eliminate all developer effort in ensuring code function and compatibility. Significant challenges remain, such as test portability, testing system stability, platform diversity, and performance test consistency.

Developing scientific software for use by a larger community also adds importance and challenges to effective communication amongst both developers and users. Reusing tools from the wider software development space, such as GitHub/GitLab, dramatically improves the quality of communication among developers. Submitted code changes can be commented on directly before merging into the main codebase, instead of having in-person "integration meetings" to perform those tasks. The MOOSE team has also utilized real-time communication platforms, such as Slack, to give code teams consistent communication capabilities across organizations and the world. User communication tools, like GitHub "Discussions" forums, excel over traditional software mailing-lists, resolving issues and questions with a more consistent, searchable, and organized experience. Establishing easy and effective avenues of communication alleviates developer effort in gaining project feedback and monitoring user needs. Fresh perspectives and ideas resulting from communication are hugely beneficial for project improvement. However, resolving the increased frequency of requests, issues, and questions can take a significant amount of developer time, especially for complicated applications. This competes with the need to deliver scientific advances and improve the project.

As the user base for a scientific software project grows, consideration of the user experience can lead to significant effort. Users range widely in software usage and development experiences – from compilation and command-line experts to relative novices – making basic installation and usage challenging to manage and improve upon. In recent years, this has been enhanced by the rise of multiple software packages made to address complex software environments and facilitate interface accessibility. Package management software such as Conda [20] and Spack [21] makes it easier for users of all kinds to obtain working versions of complex software within a consistent operating environment. Jupyter [22] and Open Ondemand [23] facilitate a user-friendly interface to software and HPC resources for application users and domain scientists. However, a sizeable remaining gap and area of additional effort is the need for graphical user interfaces (GUIs) in many scientific software projects. Clickable menus and buttons for operating a software tool are standard features in commercial products. This usability gap is often not addressed in scientific software projects or only addressed on the "input file" level. An impediment to improving this is often person-hours and expertise. Design and construction of a GUI not only involves interactivity considerations (the placement of a button/menu or other aspects of how a user interacts with the GUI) but issues of accessibility, ensuring that researchers with visual or other impairments can effectively use the software. Consideration of these needs is often secondary to the primary function and feature set of a project itself. Still, funding and expertise in this area are vital to wider community usage of any software product. Exploration of GUI development practices in other software development areas should also be a focus, keeping in mind the unique needs of scientific computing.

Funding these user/community support efforts is extremely difficult. It is often done either on developers' own free time (evenings/weekends) or as a small effort on the side of paid technical work. As mentioned previously, as the userbase grows, this support method becomes untenable. In recent years, the MOOSE project has successfully secured three forms of funding for infrastructure/support activity. Firstly, INL utilizes its own money to maintain the open-source project itself. This includes time for essential external (non-DOE) user support, maintenance of supporting capability (such as documentation/websites/GUIs), essential repository management (for instance, ticket assignment and triaging for externally originating tickets), and maintenance work in the underlying libraries (libMesh and PETSc). Secondly, the primary DOE project that utilizes MOOSE and NEAMS pays for two different kinds of support activities: maintenance of the CI/CD system (including the hardware and the software distribution mechanisms) and application support for the applications being built within NEAMS. Outside of support activities, NEAMS also funds forward-looking MOOSE enhancements for the future benefit of the project. There is approximately equal funding for the upkeep costs (infrastructure, maintenance, and community support) as there is for costs related to direct enhancement of the MOOSE framework. This type of direct program support is required for stable funding of foundational software.

5. *Challenges in building a diverse workforce and maintaining an inclusive professional environment:*

Recruiting and retaining highly skilled software developers of any kind will continue to be difficult into the future. As software engineering is a highly desired skill in the modern economy, pay, benefits, and flexibility in work schedule and work location are critical considerations for any employer. Given the specific needs of a scientific software package, software development skills must also go together with engineering or scientific knowledge, making the possible talent pool even smaller.

Recruiting underrepresented and/or underserved communities adds another layer to the challenge. Current biases in recruitment or talent pipelines established over many years can cause issues in sourcing diverse candidates upfront. During the interview process, conscious and unconscious bias, and lack of diverse representation on hiring committees can discourage candidates from considering a potential offer. Once hired, a workplace without an already diverse workforce or diversity-minded employee mentoring and resources might also discourage employees from underserved groups.

The practice of developing widely used scientific software also requires some thought regarding inclusion and equity. Development and collaboration communities tied to software often develop unique "personalities" that can be (consciously or unconsciously) very welcoming to some groups and not to others, exacerbating previously established majority groups. The names of software components might be unwelcoming – for example, the use of "master" vs. "main" for the name of the primary software code branch. Software packages, tools, and platforms might not be welcoming or accessible to those with visual impairment or other disabilities.

At INL, we have leveraged networks established via university collaboration and partnerships to attract more diverse talent referrals for interns, post-docs, and staff. Employee resource groups (focusing on LGBTQ+, international, and other underrepresented staff) are often promoted. On the MOOSE team, to decrease unconscious bias in the interview process, structured interview question guides are determined before the interview process begins and are based on desired job qualifications or quantifiable code expertise. Community expectations and guidelines are set and posted within the MOOSE code development and collaboration process, especially for user interaction and support. Poor behavior is called out and actively discouraged.

Another aspect of retaining scientific software developers is facilitating and promoting career progression. Traditionally, the software itself has not been seen as an important product in many areas of science, with the result of the code application often being what is cited and promoted. This has been slowly changing, with recent efforts such as the *SoftwareX* Elsevier journal [24] and the *Journal of Open Source Software* [25] allowing for better recognition of the importance of software by making it citable, and thus enabling better career progression. This practice needs to be expanded to ensure recognition.

8. *Management and oversight structure of the stewardship effort:*

Management and oversight of the scientific computing ecosystem, with many competing interests and institutions, is an arduous task. However, some inspiration might be taken from the broader engineering and software development industries. Standards organizations – such as IEEE, ISO, and W3C – have long used many working groups to bring together private/public institutions, governments, and individual experts. A "Software Stewardship Working Group" bringing together all stakeholders – DOE, national labs, universities, software standards bodies, and HPC vendors – could lend focus and some level of management, testbed standardization, and infrastructure planning as the nation turns toward Exascale computing.

A significant part of that planning (and a method to improve the coordination among DOE user facilities) could focus on generic testing and capability infrastructure. Open-source, cross-platform, operating-system-level virtualization technologies such as Singularity [26] could be harnessed to provide consistent individual environments and encapsulation of testing processes. This ensures integrity and reproducibility. In the MOOSE ecosystem, while CIVET manages the testing procedure and integrates code changes, it uses Singularity "under the –hood" to provide testing environments based on multiple versions of Linux (CentOS and Ubuntu variants) on INL HPC systems. Multiple Singularity instances comprising multiple possible software configurations enable a wide breadth of testing and a consistent environment for individual users who might want to also use instances of those containers on INL HPC systems.

Trust in the security and integrity of these centralized resources will be critical to adoption. Trusted execution environments, such as AMD-TEE and SGX for CPU execution and Telekine [27] for GPUs, can be used to enable the execution of codes/input files without users and developers worrying that sensitive information will be leaked on the backend. Similarly, increased integrity can come from remote attestation, which allows centralized resources to prove that they are uncorrupted and running approved software and hardware.

9. *Assessment and criteria for success for the stewardship effort:*

Metrics for success in a scientific software project, especially one hosted on an open platform such as MOOSE, can be challenging to ascertain. Usage and downloads of the code cannot be directly controlled. They might not be easily monitored, so external developments and successes often come to the development team's attention via direct user engagement and published works. When choosing metrics to determine day-to-day success (and trends over time), those specific to the code hosting platform and CI/CD system are generally preferred. Measures of impact on scientific fields and application areas often use the number and frequency of published work and external engagement metrics.

The gateway into MOOSE for a new or established user/developer on the GitHub hosting platform is generally the MOOSE "Discussions" forum. As the official portal for support, comments related to installation, getting started, model development suggestions, and bugs are presented to the community for assistance. The number of discussion submissions can be an excellent measure of community

engagement, along with the number of "answered" Discussions that indicate resolved threads. Similarly, users and developers are encouraged to submit tickets (also called "Issues") containing formal bug reports, feature requests, and general ideas for improvement. The MOOSE code repository currently holds over 8,200 closed and 1,400 open tickets. These numbers and the number of tickets closed through code change merges (submitted via "Pull requests," or PRs) are another essential code and community health metric. The number of PRs merged might be the best metric of code health and improvement alone, as many stagnant public software projects still receive tickets and community ideas but do not receive regular code updates. MOOSE often has over 50 PRs up for review and testing at any given time, leading to millions of tests run per month through the CIVET CI/CD tool. The high number of tests run not only suggests the high number of proposed code changes but also the breadth of the MOOSE ecosystem, considering that CIVET tests all internal INL software projects alongside submitted external projects and applications.

While the metrics for day-to-day success often revolve around the code repository, the impact of MOOSE and its ecosystem on the application domains it serves is often measured externally. Citations for publications using MOOSE or MOOSE-based applications in Technical Reports (generally national laboratory or industry collaborations) or software-focused journals such as *SoftwareX* and the *Journal of Open Source Software* are monitored and promoted on the MOOSE website. The number and frequency of these publications are additional indicators of platform use and effectiveness. The number of users visiting the MOOSE websites (about 1,500 unique visitors per week) and downloading tools (about 100 visitors and 70 unique clones per day) – monitored using Google and GitHub analytics, respectively – as well as where they come from (e.g., search engines, GitHub links, etc.) help the development team predict future resource needs stemming from community growth as well as the breadth of reach MOOSE might have in digital spaces. Further, when MOOSE trainings are performed, event registration is used to give insights on the number of attendees, institution/organization breakdown, and relative level of development experience among the most active community members.

References

1. Gaston, Derek, Chris Newman, Glen Hansen, and Damien Lebrun-Grandié. "MOOSE: A parallel computational framework for coupled systems of nonlinear equations." Nuclear Engineering and Design 239, no. 10 (2009): 1768-1778. https://doi.org/10.1016/j.nucengdes.2009.05.021.
2. Permann, Cody J., Derek R. Gaston, David Andrs, Robert W. Carlsen, Fande Kong, Alexander D. Lindsay, Jason M. Miller, John W. Peterson, Andrew E. Slaughter, Roy H. Stogner, and Richard C. Martineau. "MOOSE: Enabling massively parallel multiphysics simulation." SoftwareX 11 (2020): 100430. https://doi.org/10.1016/j.softx.2020.100430.
3. Williamson, Richard L., *et al*. "BISON: A Flexible Code for Advanced Simulation of the Performance of Multiple Nuclear Fuel Forms." Nuclear Technology 207, no. 7 (2021): 954-980. https://doi.org/10.1080/00295450.2020.1836940.
4. Bolisetti, C., A. S. Whittaker, and J. L. Coleman. "Linear and nonlinear soil-structure interaction analysis of buildings and safety-related nuclear structures." Soil Dynamics and Earthquake Engineering 107 (2018): 218-233. https://doi.org/10.1016/j.soildyn.2018.01.026.
5. **Garg, Mayank, Jia En Aw, Xiang Zhang, Polette J. Centellas, Leon M. Dean, Evan M. Lloyd, Ian D. Robertson et al. "Rapid synchronized fabrication of vascularized thermosets and composites." Nature communications 12, no. 1 (2021): 1-9.**
6. Goli, Elyas, Ian D. Robertson, Harshitt Agarwal, Emmy L. Pruitt, Joshua M. Grolman, Philippe H. Geubelle, and Jeffery S. Moore. "Frontal polymerization accelerated by continuous conductive elements." Journal of Applied Polymer Science 136, no. 17 (2019): 47418.

7. Qu, Q., A. Wilkins, R. Balusu, J. Qin, and M. Khanal. "Floor seam gas emission characterization and optimal drainage strategies for longwall mining." CSIRO, ACARP project C26050. https://www.acarp.com.au/abstracts.aspx?repId=C26050
8. Lindsay, Alexander, *et al*. "Automatic Differentiation in MetaPhysicL and Its Applications in MOOSE." Nuclear Technology 207, no. 7 (2021): 905-922. https://doi.org/10.1080/00295450.2020.1838877.
9. Aagesen, Larry K., Daniel Schwen, Michael R. Tonks, and Yongfeng Zhang. "Phase-field modeling of fission gas bubble growth on grain boundaries and triple junctions in UO2 nuclear fuel." Computational Materials Science 161 (2019): 35-45.
10. Gaston, Derek R., John W. Peterson, Cody J. Permann, David Andrs, Andrew E. Slaughter, and Jason M. Miller. "Continuous Integration for Concurrent Computational Framework and Application Development." Journal of Open Research Software 2, no. 1 (2014): 1-6. https://doi.org/10.5334/jors.as.
11. https://github.com/idaholab/civet
12. Falgout, Robert D., Jim E. Jones, and Ulrike Meier Yang. "The design and implementation of hypre, a library of parallel and high performance preconditioners." In *Numerical solution of partial differential equations on parallel computers,* 267-294. Springer: Berlin, Heidelberg, 2006.
13. Balay, Satish, *et al.* "PETSc Web page." Accessed December 10, 2021. https://petsc.org/
14. Balay, Satish, *et al.* "PETSc/TAO Users Manual." ANL-21/39 – Revision 3.16. Argonne National Laboratory, 2021.
15. Balay, Satish, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries," in *Modern Software Tools in Scientific Computing*, 163-202. (Birkhäuser Press, 1997).
16. Kirk, Benjamin S., John W. Peterson, Roy H. Stogner, and Graham F. Carey. "*libMesh*: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations." Engineering with Computers 22, no. 3-4 (2006): 237-254. https://doi.org/10.1007/s00366-006-0049-3.
17. Paszke, Adam, *et al*. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In *Advances in Neural Information Processing Systems 32*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, 8024-8035. Curran Associates, Inc.
18. Abadi, Martín, *et al*. *TensorFlow: Large-scale machine learning on heterogeneous systems*. 2015. https://www.tensorflow.org
19. Abou-Jaoude, A., D. Gaston, G. Giudicelli, B. Feng, and C. Permann. 2021. "The Virtual Test Bed Repository: A Library of Multiphysics Reference Reactor Models using NEAMS Tools." in *Transactions of the American Nuclear Society: Winter Meeting*. American Nuclear Society.
20. Anaconda Software Distribution. Conda, version 4.11.0. Austin, TX: Anaconda 2021. Accessed December 10, 2021. https://www.anaconda.com
21. Gamblin, Todd, Matthew P. LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and W. Scott Futral. 2015. "The Spack Package Manager: Bringing Order to HPC Software Chaos." In *Supercomputing 2015 (SC'15),* Austin, TX, November 15-20, 2015. LLNL-CONF-669890
22. Kluyver, Thomas, *et al.* "Jupyter Notebooks – a publishing format for reproducible computational workflows." In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, edited by F. Loizides and B. Schmidt, 87-90. IOS Press, 2016.
23. Hudak, Dave, Doug Johnson, Alan Chalker, Jeremy Nicklas, Eric Franz, Trey Dockendorf, and Brian L. McMichael. "Open OnDemand: A web-based client portal for HPC centers." Journal of Open Source Software 3, no. 25 (2018): 622 https://doi.org/10.21105/joss.00622.
24. https://www.journals.elsevier.com/softwarex
25. https://joss.theoj.org/

26. Kurtzer, G. M., V. Sochat, and M. W. Bauer. “Singularity: Scientific containers for mobility of compute.” PLoS ONE 15, no. 5 (2017): e0177459. https://doi.org/10.1371/journal.pone.0177459.
27. Hunt, Tyler, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. “Telekine: secure computing with cloud GPUs.” In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation, 817-834. USENIX Association, 2020.*