979-8-3503-8713-1/24/$31.00 ©2024 IEEE

A Deployment Tool for Large Scale Graph Analytics Framework Arachne

Garrett Gonzalez-Rivas, Zhihui Du, David A. Bader
Department of Data Science
New Jersey Institute of Technology
Newark, NJ, USA
{grg, zd4, bader}@njit.edu

2024 IEEE High Performance Extreme Computing Conference (HPEC) | 979-8-3503-8713-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/HPEC62836.2024.10938515

***Abstract*—Data sets have grown exponentially in size, rapidly exceeding the scale at which traditional exploratory data analysis (EDA) tools can be used effectively to analyze real-world graphs. This led to the development of Arachne, a user-friendly tool enabling interactive graph analysis at terabyte scales while using familiar Python code and utilizing a high-performance back-end powered by Chapel that can be run on nearly any *nix-like system. Various disciplines, including biological, information, and social sciences, use large-scale graphs to represent the flow of information through a cell, connections between neurons, interactions between computers, relationships between individuals, etc. However, to take advantage of Arachne, a new user has to go through a long and convoluted installation process, which often takes a week or more to complete, even with assistance from the developers. To support Arachne's mission of being an easy-to-use exploratory graph analytics tool that increases accessibility to high performance computing (HPC) resources, a better deployment experience was needed for users and developers. In this paper, we propose a tool specially designed to greatly simplify the deployment of Arachne for users and offer the ability to rapidly and automatically test the software for compatibility with new releases of its dependencies. The highly portable nature of Arachne necessitates that this deployment tool be able to install and configure the software in diverse combinations of hardware, operating system, initial system environment, and handle evolving packages and libraries in Arachne. The tool was tested in virtual and real-world environments, where its success was evaluated by an improvement to efficiency and productivity for both users and developers. Current results show that the installation and configuration process has been greatly improved, with a significant reduction in the time and effort spent by both users and developers.**

***Index Terms*—open-source framework, software deployment, large-scale graph data, graph analytics**

I. Introduction

Data sets being analyzed by data scientists, and scientists in general, have grown exponentially in size, rapidly surpassing the scale at which traditional exploratory data analysis (EDA) tools [9], [24] can be used effectively. Arkouda [17], [20] is an open source software framework that overcomes the scalability limitations of traditional EDA tools. Arachne [21] is a software package for Arkouda, empowering it to support the analysis of massive real-world graphs. Arkouda and Arachne are made up of a pair of user-facing Python interfaces (the Arkouda and Arachne clients), with a high-performance back-end (the Arachne server) written in Chapel [2], [3]. By leveraging Chapel, an open-source, high-level, parallel programming language designed for high performance computing (HPC), Arachne's server offers: the ability to be run in single or multi-locale configurations, native vendor-agnostic GPU acceleration, and distributed data structures. The Arachne server is highly portable and supports nearly all *nix-like systems [13]. It takes advantage of massively parallel algorithms written in Chapel to allow for the analysis of datasets on the order of hundreds of terabytes in a matter of seconds, with the performance of some algorithms scaling nearly linearly with up to several thousand nodes.

It is crucial for data analysis to preserve the 'human-thought loop', which is the ability for a data scientist to ask questions, execute the code representing those questions, analyze the results, and form new questions. Traditional EDA tools became ubiquitous because they enabled this loop, however, they cannot do so at scale. They either impose long wait times due to slow execution of code (e.g. Python, C/C++, Fortran, etc.) or require the user to implement intricate and specialized technologies (e.g. OpenMP [4], CUDA [22], etc.), making this style of continuous hypothesis testing inefficient, if not impossible. Arachne's Pythonic interface and its ability to take advantage of HPC resources allows for the computation of terabyte-scale data sets in real-time while writing code fundamentally no different than using traditional EDA tools. The Arkouda and Arachne Python clients use APIs (application program interfaces) designed to be familiar to data scientists, taking inspiration from the Python libraries currently used in 80% of data science workflows [16], and fitting seamlessly into the predominant EDA platform: Jupyter Notebooks [10]. Arkouda/Arachne was developed with the vision of empowering data scientists with better tools that incorporate a user interface they already know so they can take advantage of interactive, scalable EDA tools with ease.

Arachne is an incredibly powerful software package, but before any piece of software can be used, it must be installed, configured, and learned. Arachne's design paradigms make it exceptionally easy to learn, reducing the amount of time between starting to use the software and being productive with it. However, the process of deploying the software still poses a significant hurdle to would-be users, a challenge shared by many large open-source software packages. Within our own lab (the Arachne developers), it is not uncommon for new individuals joining the team to take up to a week or more to

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on April 10,2025 at 10:06:23 UTC from IEEE Xplore. Restrictions apply.

get Arachne running on their system. When collaborating with other labs, it often takes a significant amount of interaction over a considerable period of time before the software environment is set up and working. These issues in the installation and configuration of Arachne prevent perspective users from trying the software, hindering its ability to become more widely adopted and being a useful solution for more people. As Arachne is becoming one of the foremost tools for performing large-scale graph analysis in neuroscience, among other fields, Arachne is in need of a tool to address the problematic deployment process.

The major contributions of this work are as follows:

1) A fast and easy-to-use deployment tool facilitating a more straightforward trial of or migration to Arachne.
2) A way for Arachne developers and community contributors to quickly and automatically install different configurations of the software, reducing barriers to entry and accelerating testing.
3) Experimental results in virtual and real world situations that demonstrate improved efficiency when using a deployment tool for Arachne.

II. Analyzing the Problem

Arachne's deployment difficulties arise from a long, intricate, and complex installation process that is further complicated by the lack of adequate documentation[1]. The documentation to deploy Arachne is broken into two sections 'Installing Prerequisites' and 'Building the Arkouda Server with Arachne', which together take up less than a page. The user is instead instructed to follow certain subsets of instructions across different sections of Arkouda's documentation, which also contains several redirects to different sections of documentation, some of which are two years or more out of date and contain broken links. Assuming the user has no issue finding the instructions they need to follow, they still have to:

- Perform the installation of nearly two dozen packages (which varies between operating systems),
- Configure Python environments with Anaconda (and its installation if they don't have it on their system),
- Download, configure, and build Chapel (with different configurations for different use cases),
- Link the dependencies to Arkouda Makefile.paths (for the GNU Make build system),
- Build Arkouda with the Arachne (after going back to Arachne's documentation).

To get a running Arachne environment through this process requires the user to skip over several steps of the Arkouda documentation (which they are never explicitly told to do), use different versions of certain dependencies than specified in the Arkouda documentation, and adjust commands being run to fit their system (such as the number of concurrent jobs when building Chapel).

Although it is possible to complete the installation by being very careful, reading everything thoroughly, and having a good understanding of (and experience with) everything that needs to be done, it is unlikely that users who are unfamiliar with performing such installations will be able to succeed. The diverse combinations of hardware, operating system, and preexisting system software make creating documentation that is applicable to every user very difficult and, even if it were to be done, the user would have to look through a substantial amount of said documentation, find what is directly applicable to them, understand the complexity and intricacy of Arachne's deployment process, and then perform it without making any mistakes.

Improved documentation is a vital tool for developers, system administrators, and technically experienced members of the community; however, the installation process will remain challenging for less technically experienced users and will take several hours of active involvement to complete. In addition, the large number of steps that need to be taken means a non-trivial likelihood of mistakes being made, which prospective users likely wouldn't be able to understand or remedy without assistance. The installation process of Arachne, largely due to the installation processes of Chapel and Arkouda, offers nearly a hundred environment variables and configuration options to set, and while most of these aren't used in the majority of installations, their presence can intimidate users and their intricate nature may confuse developers. While some of these problems are specific to Arachne, other complex open-source packages often face the same, or similar, deployment challenges.

A. Challenges for Developers

Arachne's development workflow usually includes the initial prototyping done on a laptop or workstation, preliminary testing done on a small cluster, and the final deployment being done on a supercomputer or in the cloud. This means that to engage in Arachne development (in our lab, as a collaborator, or as a member of the community), it would have to be installed on each developer's computer. The time and difficulty it takes for the initial installation of Arachne can, at best, increase the amount of time before productive work can start or, at worst, deter individuals or groups from working with Arachne altogether. In addition, Arachne developers have to install the software repeatedly with experimental configurations to verify its compatibility with new releases of Arkouda, Chapel, and other dependencies. Currently, the time-consuming process of building Arachne with custom configurations for testing prevents developers from being able to simultaneously do other work, reducing their efficiency and increasing the 'cost' of performing thorough testing. In addition, the current installation process of Arachne routinely necessitates its developers to stop other work to aid users in getting the software working on their system successfully.

B. Challenges for Users

Installing Arachne necessitates the installation and/or compilation of many packages and dependencies, the configuration of multiple environments, and the building of Ark-

[1]Which is currently in the process of being rewritten.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on April 10,2025 at 10:06:23 UTC from IEEE Xplore. Restrictions apply.

ouda/Arachne itself. Not only can this be daunting to less technically experienced users, but even developers and system administrators experienced with this so-called 'dependency hell' [7] often face issues with version conflicts, missing dependencies, and out-of-date documentation, among other issues. These issues are often difficult to trace to their root causes, even by professional developers. For example, the order in which the steps of configuring Arachne's Python environment are completed can create a critical failure, with an error message regarding the C compiler, stating that it 'is likely not a problem with pip' even though that is the root cause of the error. These kinds of issues can be very difficult for users not working in software development to resolve. The installation and configuration process will likely be the first interaction with the project that a prospective user will have and, therefore, it is imperative that it be as simple and easy as possible. Arachne is currently being used primarily in the analysis of: cyber, financial, and national security; biological and social sciences; and information systems. Furthermore, Arachne is becoming the foremost graph analysis tool in neuroscience connectome research; to succeed at this, it must handle an influx of users experienced in neuroscience, rather than systems administration, who need to install the software.

III. Design and Development of the Tool

The design of this deployment tool should ensure that it can provide value to both developers and users, regardless of their level of experience, which can be measured by an increase in productivity and efficiency. There are several mature build/deployment systems that offer different functionalities, such as GNU Make [15], Meson[2], and Jenkins [23], however, these general-purpose tools lack the flexibility to exactly fit Arachne and the challenges faced by its developers and users. A specialized approach, on the other hand, can be customized to include the functionality requested (e.g. the ability to easily install different releases without version conflicts), while ensuring that it is straightforward for users and configurable for developers. While this tool is designed to address the specific needs of Arachne, its design paradigms are applicable to other large or complicated software packages.

A. Ease of Use

From a development perspective, 'ease of use' means that creating and updating of the installer script should be largely automated, and integrated closely with the existing development workflow. To support this, the deployment tool's workflow should leverage GitHub actions to, whenever a pull request is accepted or a commit is pushed, create both the installation script itself and a companion file. These two files should be created from a configuration file updated by the repository maintainer. The configuration file should use a standard config file format (YAML [6], TOML[3], JSON [18], etc.) to make implementing and extending the tool's functionality easier. This file should contain both declarative instructions (used to generate the companion file) as well as imperative logic structures (used to generate the installation script). The companion file should outline the specific environment variables that need to be set, the versions of dependencies to use, and other information regarding the installation process. This would enable automated testing of experimental configurations across various environments, integrating with most major CI/CD pipelines [25], as well as the ability for branches (and forked repositories) to have different companion files (and for the developers/users to tell the installer where to look for its companion file).

From a user perspective, 'ease of use' means that the installer should be highly portable (i.e. allowing the same script to be used regardless of the computing environment), simple to use, and able to provide easily understandable information to the user. The installer should also have easy to set options for common configurations that a user might want, such as the desired release type (stable or rolling release) and environment type (user, developer, or just the Python client). The installer should use the presence of the companion file in Arachne's GitHub repository to ensure it has the most up-to-date information for a rolling release installation or be able to go back and get the appropriate dependency versions for previous stable releases of Arachne.

B. Variable Levels of Interactivity

One way for both users and developers to be more productive would be for the installation of Arachne to be handled in an automatic, or 'unattended', manner. By not requiring human interaction during the installation process, the time to get new team members set up to start working would be reduced to the time it takes their system to compile dependencies, developers could run multiple tests simultaneously while doing other work, and users could get the software installed on their system easily and quickly. However, a lack of interaction risks compromising the tool's value for developers and system administrators, so it should be designed to function across different levels of interactivity as follows:

1) Unattended: When running the installer in 'unattended' mode, a user should only have to copy a single line of code from the documentation into their terminal and run the command to get a usable Arachne environment on their system when it finishes running. The user should be able to pass arguments to the installer to set common installation options, such as whether to install a user or developer environment.

2) Attended: When running the installer in 'attended' mode, the installer should still be approachable and provide human-readable messages that are understandable without experience or background in handling software installation in a *nix-like environment. Whenever prompting the user for input, it should provide a high-level summary of the changes that would be made to their computer and give the recommended response for their system.

3) Developer: When running the installer in 'developer' mode, the installer should allow the user a great deal of control over the installation process, providing more detailed

[2]https://mesonbuild.com/

[3]https://toml.io/en/

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on April 10,2025 at 10:06:23 UTC from IEEE Xplore. Restrictions apply.

information, increased logging, the ability to stop and resume (or restart) the installation, or the option to modify the next step in the installation while the script is running. It should allow a user to override any part of the installation process by passing the installer a configuration file (e.g., replace or change the version of a package/dependency, change how or when environments are configured, etc.).

The deployment tool should always perform error checking and adjust its response depending on what mode the script is being run in. By performing checks after each step of the installation process, the installer should be able to determine if, where, and what error occurred. The installer should collect information by leveraging a list of common, known issues, where the installer ran into an issue, and search the output stream for phrases or patterns associated with these issues. Once it has collected information, it should tailor its response depending on run mode by: 1. attempting to handle the errors (unattended), 2. providing a human-readable list of likely/potential errors and creating a log file (attended), or 3. pausing where it encountered the error, printing detailed information to the screen, and waiting for the user to resolve the error (developer).

C. Portability

To ensure that the installer is simple and straightforward for the user, it should be able to adapt to different computing environments and allow the same command to universally perform the deployment. It should be able to run on nearly all major operating systems, including most GNU/Linux distributions [5], [8], BSD derivatives [14], [19] (including MacOS [12]), and Windows Subsystem for Linux (WSL2) [1]. To this end, it should be POSIX compliant [11], shell-agnostic, and otherwise as portable as possible. Even on systems that are incompatible with Arachne or its dependencies, the installer script should run and print a message regarding the incompatibility and direct the user to a list of compatible systems.

For all systems supported by Arachne and its dependencies, the script should be able to identify the essential system information to adjust itself to integrate with the system's package manager, file system, method of running privileged commands, etc. For example, if the system already has a method of handling Python environments installed (e.g. conda, pipenv, venv, poetry, etc.), the installer should either configure the environments using the currently installed solution or, depending on the mode the installer is running in, ask the user which method they prefer to use and provide a note that Anaconda is the recommended one.

D. Modularity

The deployment tool should be designed to be modular in order to promote the easy extension of its functionality to ensure that it maintains relevancy and usability as the software and its project develop and change. The deployment tool is made up of four parts: a configuration file, a GitHub Action, a Companion File, and an Installer Script. All four components should be modular and use common formats to be easier to work with. For example, the GitHub action should be comprised of several modules, such as: a parser, an interpreter, and generators. Different parser modules could be written to support ingesting different configuration file formats. The interpreter module could be extended to support custom logic. New generator modules could be written to allow for the generation of the installer script or companion file in different languages (such as Python or Pkl) or even Docker images.

IV. Implementation

The deployment tool's implementation is such that a single command can be copied into the terminal of the vast majority of GNU/Linux systems and, as quickly as their machine can finish the compilation, they will have a working Arachne installation. The installer script, 724 lines of POSIX shell in seven discrete modules, runs in both the 'attended' and 'unattended' modes without any issues. Several user-experience improvements are planned, such as a continuous progress status (in addition to one for each step) and more human-readable messages throughout the installation. To achieve this, the installer has very robust information collection mechanisms and the ability to integrate with the system package managers of GNU/Linux distributions and MacOS (with other BSD-derivatives being planned to be supported in the future). The installer also integrates with the two most popular tools for handling Python environment management (Anaconda & venv/pip), while support for other tools (pipenv and poet) have been outlined but are not currently under development. The installer's error checking is thorough, although its ability to handle errors is currently limited to only a few common issues with Anaconda environments, Chapel configurations, and linking dependencies to Arkouda (`Makefile.paths` remediation through symlinks). Further development into improving the error handling mechanisms, such as supporting other Python environment managers, is underway. Once it is completed, the new Arachne documentation will these single commands to run the installer for different use-cases, along with how to use the installation script. The same information will be shown if a user runs it with the `-h|--help` flag or without passing it any flags.

The preliminary implementation of the deployment tool has focused on getting a fully-functional installer script for users to begin taking advantage of, and thus much of the integration with GitHub actions is still under development. A configuration file (implemented in YAML) has been written that is able to represent the installer, which is comprised of 491 lines split across three sub-documents. The Python script used to generate the companion file and installer script from the configuration file has been written with 605 lines across four modules (a parser, an interpreter, and two generation modules – one for the companion file and one for the installation script). Further development to make the configuration file easier to use and the integration of the Python script with GitHub actions is underway. The companion file, currently implemented in TOML (due to the ease of writing a parser for that format in POSIX shell), supports the overriding

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on April 10,2025 at 10:06:23 UTC from IEEE Xplore. Restrictions apply.

of: packages and dependencies (their versions, if they are installed or compiled, etc.), the configuration of the Chapel environment, and the Anaconda environment (entirely or with specific modifications). The ability for this file to contain overrides for any function/step of the installer (through direct code replacement or another mechanism) has been partially implemented but is still in early testing and is not feature complete. Similarly, some of the more advanced features of the 'developer' run mode are still under development, such as the ability to stop and resume (or restart) the installer, or to modify its next step before it is run. These features have been prototyped by having the installer create a temporary file that tracks initial flags/values passed, files and variables created/set, steps completed, and the actual commands for the next step to be taken; by writing its next step to a file and pausing, the user can edit the next step without having to stop and restart the installer.

V. Results

The deployment tool has been tested in both virtual and real-world environments. Virtual testing was primarily conducted on virtual machines (VMs) running a set of common distributions of GNU/Linux[4], while additional testing was done on less-used operating systems[5]. These VMs were hosted[6] on a high-performance server with 2 x Intel Xeon E5-2630v3 @ 2.4 Ghz CPUs with 16 cores per CPU and 226 GB of DDR4 RAM, running ProxMox VE 8.1.4, and were provided with 20 virtual CPU cores (x86-64-v2-AES), 32 GB of memory, and a 64 GB boot disk. Results of both virtual and real-world tests are measured in three ways: the duration of the installation process, the duration and number of necessary human-interactions, and the number and severity of errors encountered during the installation process.

A. Virtual Testing

When testing for compatibility and portability between GNU/Linux environments, the deployment tool was run in Fedora, Debian, five different versions of Ubuntu, and two different editions of RHEL. All but one environment was able to run and install Arachne without issue, while the final environment (RHEL 9.4 Workstation) had an issue with the configuration of its Chapel environment. In this instance, the environment was configured (defining a set of variables in a plaintext file) and recompiled (with `make clobber`) manually. When the installer was re-run, it recognized Chapel was already on the system and continued with the rest of the deployment without issue. Further compatibility testing was completed on two different versions of MacOS, FreeBSD, and WSL2 through Windows Server. When running on FreeBSD, a handful of issues were encountered, which went unresolved due to time constraints and the relatively low priority of supporting that operating system at this time. Otherwise, the same script was run on eight different distributions/versions of GNU/Linux, two versions of MacOS, and several distributions in WSL2 on a Windows Server 2022 VM, all without issue.

After testing for portability, the deployment tool was tested for its resiliency against variable initial system states while being simultaneously benchmarked against the three metrics mentioned above – these tests were all conducted on VMs running Ubuntu 22.04 Server LTS. These VMs all had different simulated software configurations installed onto them (using configurations found on GitHub and Reddit as well as common GNU/Linux utilities). Looking at the first 14 tests (run in pairs, with 2 VMs installing Arachne simultaneous and the installers in 'unattended' mode), the user only had to interact with their machine twice (once to paste the command / run it, and once to enter their password), and in every test the user was finished interacting with the installation process within 30 seconds. The total time until the first start of Arachne (by the user, not the automated testing) was, on average across all 14 tests, 2 hours 21 minutes and 57 seconds. All fourteen tests completed without encountering any issues, producing fully functional Arachne installations, although certain configurations seemed to take an unusually long time when solving their Python environments (despite using Anaconda).

Next, to simulate a system administration workflow, four systems were connected via SSH and the installer was run on them in 'unattended' mode, but this time each system had different flags passed to the installer (i.e. release: stable/rolling and environment: user/developer). This test was run three times, and the average total time of human interaction (connecting to each machine, writing out the command, and entering a superuser password) was just 4 minutes and 35 seconds. In all of the tests, all of the Arachne installations were successful, creating four different environments with less than 5 minutes of active work. While having 4 simultaneous installations slowed the time until the first start of Arachne to nearly 7 hours rather than 2, this is almost certainly a limitation of the virtual environment host rather than something that needs to be corrected in the deployment tool.

When running the installer in 'attended' mode, there was a higher number of human-interaction events (as it is designed to do) and, while the time spent performing those actions were low (usually only a few seconds to a minute), they were spread throughout the installation process and, therefore, required the user to be monitoring their system. Averaging the timings across 6 runs (3 sets of 2 parallel tests), it took approximately 45 seconds when the script was first run to select the 2 options (release and environment), enter the root password, and accept the list packages being installed by their system's package manager. It took approximately 2 hours for the prerequisite dependencies to be installed, Chapel to be configured and built, Arkouda and Anaconda installed, and their environments configured. If the user selected a developer environment, they were prompted to enter the URL of the fork of Arachne they would like to use or to simply hit enter to use the upstream repository. After approximately 20 more minutes (while the

[4]Ubuntu 20.04 (Desktop & server), 22.04 (Desktop & server), & 23.10 Desktop, Debian 12.5 Desktop, Fedora 39 Workstation, and Red Hat Enterprise Linux (REHL) 9.4 (Workstation & Server).

[5]Windows Server 2022 Standard, MacOS 13 & 14, and FreeBSD 13.3.

[6]Except the MacOS VMs, as per Apple EULA.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on April 10,2025 at 10:06:23 UTC from IEEE Xplore. Restrictions apply.

Arkouda/Arachne server was built and tested), the Arachne environment was fully ready to be used.

B. Real-world results

Because it is still so new, the tool has only been run by several users on their machines so far, however, it has already demonstrated that it offers significant improvements over the original deployment process of Arachne and provided substantial benefits to the people who used it. Before the installer was created, a PhD student, who, despite working in our lab for a full year and trying multiple times, was still unable to get Arachne working on their system (running MacOS 14). After beginning working on a new project where they needed Arachne, a meeting was setup with them, and within 3 hours of running the deployment tool on their system, they had Arachne installed successfully. This was not an entirely unattended installation because it required the cleaning up of previous attempts to install the software, the setting of several environment variables before the script was run, and the use of a Chapel utility after the script finished. However, the reaction to the successful completion of this installation was significant and overwhelmingly positive, not only from the PhD student, but also from several other members of the lab, especially by those who had previously helped with or tried to install Arachne on the individual's computer.

When our lab began working with a group of researchers at Harvard, a graduate student and research fellow working in Computer Engineering and Software Development needed to deploy Arachne on their system. This process took over one hundred back-and-forth messages with several of Arachne's core developers and lasted nearly a week before they had a successfully running Arachne installation. The process required several reconfigurations of environments as well as the reinstallation or recompilation of several dependencies. Despite having nearly unlimited access to the Arachne development team, sending them over a dozen screenshots and copy-and-pasting the outputs of several terminal commands, the installation still had to be partially repeated multiple times before the Arachne installation was usable.

In contrast, when an undergraduate intern had issues installing Arachne manually (up until now, the only way to install it), they were directed to reach out to us for the deployment tool. They sent three screenshots of the errors they had been encountering. By analyzing those errors, it was revealed that several non-trivial environment-configuration errors had been made several steps previously in the installation process and that they had been trying to install incompatible versions of Arkouda and Arachne. They were directed on how to clean up their previous installation attempt and provided with the deployment tool. After that, they had a single issue with Chapel not showing up on their PATH, which they were quickly able to resolve on their own to get Arachne working. This issue was a result of the installer checking which shell process it was being run in, and because it was piped to bash, it added the shell hook for Chapel to the individual's `.bashrc`, even though their default shell was ZSH. The deployment tool has already been modified to recognize both the current and default shell; when running as an attended installation, it will prompt the user about which shell to use if they are different; otherwise, it will use the default shell. Altogether, fewer than 15 messages were exchanged over the course of four days, with less than one full day having elapsed between them receiving the script and having Arachne running successfully.

At this point, the installer has been tested on several machines that people use as their daily computer, all with existing software, environments, and other configurations on them from regular use prior to the installer being run. The installer, so far, has worked on every GNU/Linux and MacOS system it has been used on in the real world. When two other undergraduate interns, both on Windows 11 systems, tried to run the installer in WSL2 (using Ubuntu 24.04), the first individual encountered pathing issues which required a series of symlinks to be added manually, while the second ran the installer without any issues. These pathing issues appeared when using existing LLVM installations; however, the installer is now able to automatically correct these issues, if encountered, by building LLVM with Chapel. Future testing and work could be done to automate the generation of symlinks to fix this issue without rebuilding LLVM.

VI. Conclusion and Future Work

The Arachne deployment tool has enabled multiple users to easily install and configure the software environment on their systems as an unattended installation, it is able to automatically address the majority of the known complexities and challenges associated with installing such an intricate piece of software across diverse computing environments. This approach allows users to deploy Arachne with a single command, vastly streamlining the process, and greatly reducing the need for technical support. The experimental results show that, despite long compilation times, it significantly reduced the amount of time and effort required for the user to perform the installation, often to less than one minute.

Future work will focus on further refining the deployment tool, expanding compatibility, creating more robust error handling, and promoting its ability to be a largely automatically generated installation solution. In the future, it would be possible to increase the generality of the deployment tool, such as extending it to support the installation of other popular modules for the Arkouda framework. The Arkouda framework offers a variety of modules with functionality that could be beneficial for Arachne as well, such as: an improved multi-user experience through the use of multiplexed and bidirectional communications (arkouda_proxy_server), a containerized multi-locale deployment of the server on Kubernetes (arkouda-helm-charts), and a more user-friendly monitoring experience through an integration with Prometheus (arkouda_metrics_exporter). This work is a tool for Arachne, but its design idea, development experience, and some of its code/framework can be shared by other open source packages, and is publicly available on GitHub (https://github.com/GarrettGR/Arachne_Deployment_Tool).

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on April 10,2025 at 10:06:23 UTC from IEEE Xplore. Restrictions apply.

ACKNOWLEDGMENT

We thank the Arkouda and Chapel communities for their continued support, as well as our interns for helping test the deployment tool and providing feedback. This research was funded in part by NSF grant number CCF-2109988.

REFERENCES

[1] Hayden Barnes. *Pro Windows Subsystem for Linux (WSL)*. Springer, 2021.

[2] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[3] Bradford L Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson, Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, et al. Chapel comes of age: Making scalable programming productive. *Cray User Group*, 2018.

[4] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[5] Francesco Di Cerbo, Marco Scotto, Alberto Sillitti, Giancarlo Succi, and Tullio Vernazza. Toward a gnu/linux distribution for corporate environments. In *Emerging Free and Open Source Software Practices*, pages 215–236. IGI Global, 2007.

[6] Malin Eriksson and Victor Hallberg. Comparison between json and yaml for data serialization. *The School of Computer Science and Engineering Royal Institute of Technology*, pages 1–25, 2011.

[7] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 463–474, 2020.

[8] Jesús M González-Barahona, Gregorio Robles, Miguel Ortuño-Pérez, Luis Rodero-Merino, José Centeno-González, Vicente Matellan-Olivera, Eva Castro-Barbero, and Pedro de-las Heras-Quirós. Analyzing the anatomy of gnu/linux distributions: methodology and case studies (red hat and debian). In *Free/Open Source Software Development*, pages 27–58. IGI Global, 2005.

[9] David Caster Hoaglin, Frederick Mosteller, and John Wilder Tukey. *Understanding robust and exploratory data analysis*, volume 3. Wiley New York, 1983.

[10] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks–a publishing format for reproducible computational workflows. In *Positioning and power in academic publishing: Players, agents and agendas*, pages 87–90. IOS press, 2016.

[11] Donald Lewine. *POSIX programmers guide*. " O'Reilly Media, Inc.", 1991.

[12] Modhuparna Manna, Andrew Case, Aisha Ali-Gombe, and Golden G Richard III. Modern macos userland runtime analysis. *Forensic Science International: Digital Investigation*, 38:301221, 2021.

[13] B. McDonald, M. Strout, S. Coghlan, and O. A. Rodriguez. Data science beyond the laptop: Handling data of any size with arkouda. Presented at ChapelCon, Jun. 5 2024. [Online]. Available: https://chapel-lang.org/ChapelCon/2024/arkouda-tutorial.pdf, 2024.

[14] Marshall Kirk McKusick, Keith Bostic, Michael J Karels, and John S Quarterman. *The design and implementation of the 4.4 BSD operating system*, volume 2. Addison-Wesley Reading, MA, 1996.

[15] Robert Mecklenburg. *Managing Projects with GNU Make: The Power of GNU Make for Building Anything*. " O'Reilly Media, Inc.", 2004.

[16] M. Merrill. Arkodua: Interactive supercomputing for data analytics made possible by chapel. In *Achieving Productivity at Scale with Chapel in User Applications, SIAM-P22*, 2022.

[17] Michael Merrill, William Reus, and Timothy Neumann. Arkouda: interactive data exploration backed by Chapel. In *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, pages 28–28, 2019.

[18] Ioannis Papagiannopoulos. *JSON application programming interface for discrete event simulation data exchange*. PhD thesis, University of Limerick, 2015.

[19] John S Quarterman, Abraham Silberschatz, and James L Peterson. 4.2 bsd and 4.3 bsd as examples of the unix system. *ACM Computing Surveys (CSUR)*, 17(4):379–418, 1985.

[20] William Reus. CHIUW 2020 Keynote Arkouda: Chapel-Powered, Interactive Supercomputing for Data Science. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 650–650. IEEE, 2020.

[21] Oliver Alvardo Rodriguez, Zhihui Du, Joseph T. Patchett, Fuhuan Li, and David A. Bader. Arachne: An Arkouda package for large-scale graph analytics. In *The 26th Annual IEEE High Performance Extreme Computing Conference (HPEC), Virtual, September 19-23, 2022*, 2022.

[22] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[23] Nikita Seth and Rishi Khare. Aci (automated continuous integration) using jenkins: Key for successful embedded software development. In *2015 2nd International Conference on Recent Advances in Engineering & Computational Sciences (RAECS)*, pages 1–6. IEEE, 2015.

[24] John W Tukey. *Exploratory data analysis*, volume 2. Reading, MA, 1977.

[25] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 471–482. IEEE, 2021.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on April 10,2025 at 10:06:23 UTC from IEEE Xplore. Restrictions apply.