Creating Reproducible Environments with Nix for Scientific Computing

Morphy Kuffour
morphy.kuffour@uconn.edu
Department of Computer Science and Engingeering, University of Connecticut
Thesis Advisor: Prof. Clay Tabor
Honors Advisor: Prof. Caiwen Ding

Abstract

The Nix package manager provides a way to manage dependencies between multiple software packages and configurations, enabling users to access various versions of a package and its dependencies. This allows users to maintain reproducible environments for the Community Earth System Model Version 2.0 (CESM 2.0) code across different platforms, specifically for its computational components, enabling developers to have a consistent development environment for scientific computing. Managed environments also provide a way for users to easily access and install new packages specific to the CESM project, as well as quickly update its existing packages. Using the nix package manager for reproducible development environments helps ensure accurate and consistent CESM project results, and makes it easier for developers to collaborate on the same project.

1 Introduction

Scientific computing has become increasingly important in many fields of research, including physics, chemistry, biology, and Earth science. These fields rely heavily on complex software and tools to create, analyze, and visualize data. However, managing software dependencies and configurations across different platforms and systems can be challenging, and inconsistencies in these environments can lead to irreproducible results. This is where the nix package manager comes in to provide a solution.

The nix package manager provides a way to manage dependencies between multiple software packages and configurations, enabling users to access various versions of a package and its dependencies. This allows users to maintain reproducible environments for scientific computing across different platforms, enabling developers to have a consistent development environment for their projects. In particular, nix has proven to be a valuable tool for the CESM project, which is a complex modeling framework used to simulate the Earth's climate system.

In this paper, we will explore how the nix package manager can be used to create reproducible environments for scientific computing, with a focus on the CESM project. I will discuss the benefits of using nix for managing dependencies, how to create and manage nix environments, and share our experiences using nix for porting CESM project to the UConn High Performance Computing (HPC) system. I will also discuss the community support and resources available for nix users, and provide examples of other scientific computing projects that have successfully implemented nix for reproducibility. This paper aims to provide a comprehensive understanding of how nix facilitates reproducibility in scientific computing and offers guidance on incorporating nix into individual projects.

1.1 Need for software reproducibility in scientific computing

The need for software reproducibility in scientific computing cannot be overstated. Scientific research relies heavily on computer simulations and models to test hypotheses and make predictions. These simulations and models are typically implemented using various software packages and tools, and the results obtained from them are highly dependent on the specific versions of the software used, as well as the configurations and dependencies of the underlying systems.

In their paper titled "Reproducibility in Scientific Computing" (Ivie and Thain 2018) provide a comprehensive analysis of the current state of reproducibility in computational research and highlight the urgent need for improving the reproducibility of scientific results. They argue that software is a critical component of scientific research and that the inability to reproduce scientific results due to software issues undermines the integrity and reliability of scientific research.

Furthermore, the lack of reproducibility can have significant consequences, both for the scientific community and for society at large. Incorrect or unverifiable scientific results can lead to misguided policies and decisions, wasting valuable resources and potentially endangering public health and safety. For instance, public distrust of climate science has resulted in a failure to act on the overwhelming evidence of anthropogenic climate change, leading to catastrophic consequences. Therefore, it is essential that scientific software be designed and developed in a way that ensures reproducibility and facilitates the sharing and reuse of scientific code.

To address the need for software reproducibility in scientific computing, various approaches and tools have been proposed, including containerization technologies like Docker and virtualization platforms like VirtualBox, as well as package managers like Nix, which enable the management of software dependencies

and configurations. These tools and approaches help ensure that the software used in scientific research is fully documented and can be easily reproduced, validated, and reused by other researchers.

2 Nix Fundamentals

2.1 Introduction to Nix

Nix is a powerful package manager that provides a unique approach to dependency management and enables the creation of reproducible software environments. Unlike traditional package managers that rely on global installation and modification of system-level libraries, Nix uses a purely functional approach that ensures each package and its dependencies are installed in a self-contained and isolated environment. This allows for multiple versions of the same package to coexist on a system without interfering with each other. Moreover, Nix provides a declarative language for defining package dependencies and their configurations, which can be versioned and shared across different systems. By using Nix, developers can easily reproduce the exact same environment and dependencies needed to run an application, regardless of the host operating system and hardware. As a result, Nix has become a popular tool for creating reproducible environments in scientific computing, enabling researchers to easily share their work and reproduce their experiments on different platforms (Dolstra et al. 2004).

2.2 Nix's approach to package management

Nix provides a unique approach to package management compared to other traditional package managers. Instead of relying on a global installation location and modifying the system environment, Nix uses a purely functional approach to package management (Dolstra et al. 2004). Each package is installed into its own isolated environment, ensuring that packages do not interfere with each other and that dependencies are always satisfied. Additionally, Nix enables users to install multiple versions of the same package side-by-side, which can be useful for developers who need to test their software with different versions of a dependency.

Nix relies on a few essential components to implement its functional approach to package management. First, Nix uses a lazy evaluation model, meaning that packages are not built until they are needed. This allows Nix to build only the packages that are actually required, reducing the amount of unnecessary builds and saving disk space. Second, Nix uses a content-addressed store, where each package is stored with a unique content-based identifier. This means that if two packages have the same contents, they will share the same identifier and only be stored once, further reducing disk space usage. Finally, Nix uses the Nix expression language, which is a declarative language for describing packages and their dependencies.

Devresse et al. (2015) note that Nix's approach to package management is distinct in that every package is immutable and the outcome of a stateless function, rendering it deterministic and idempotent for a particular set of inputs. This is accomplished through the application of SHA-256 binary hashes for each package, which are deposited and extracted from a centralized store implementing a Key-Value store model. As stated in the paper, "In Nix, each package is immutable and the result of a stateless function, pure in the functional sense that is guaranteed to be deterministic and idempotent for a given set of inputs" (Devresse, Delalondre, and Schürmann 2015).

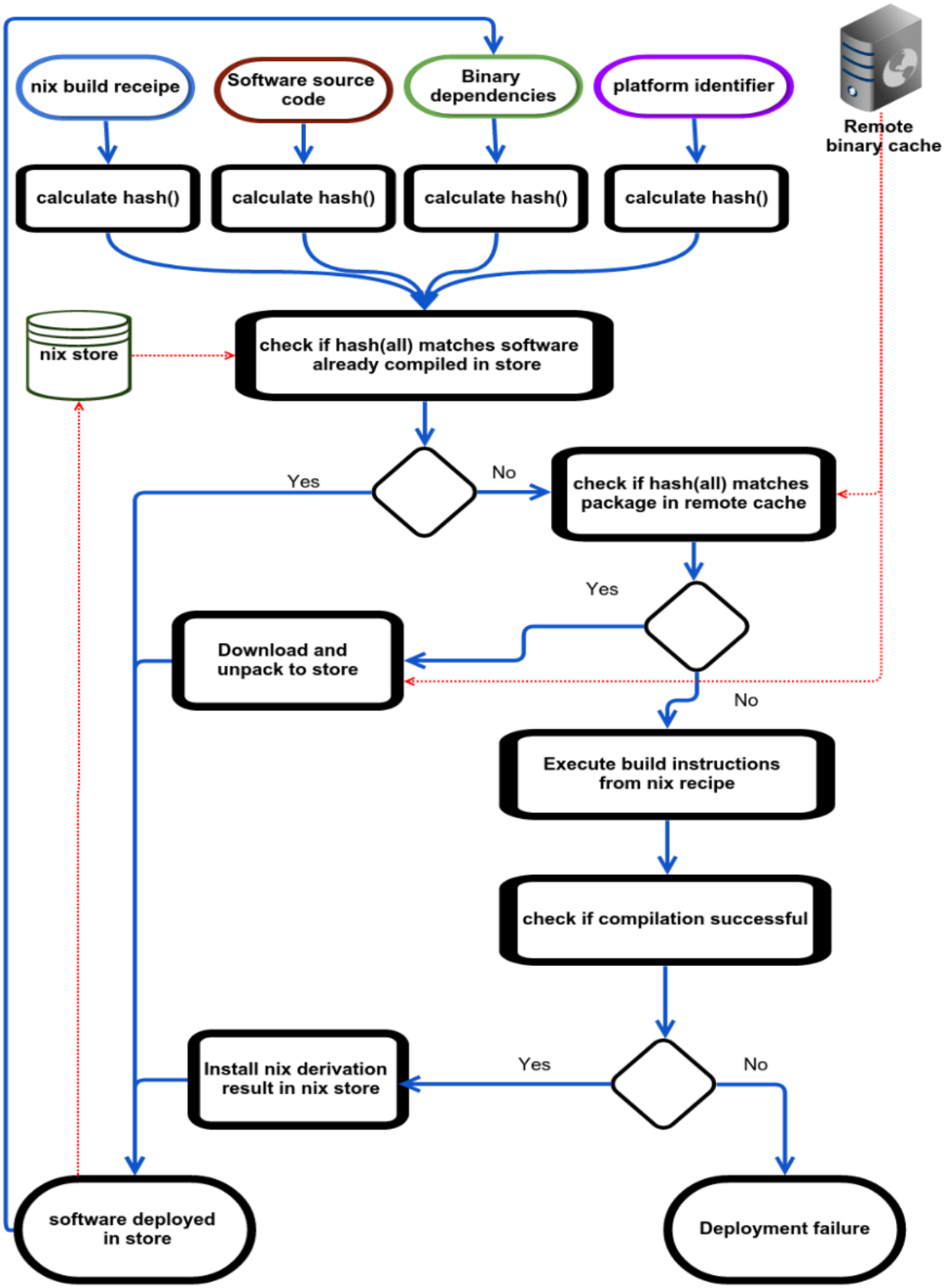

Figure 1: A diagrammatic view of the process by which nix packages software and its dependencies

3 Related Work: Nix for Scientific Computing Management

Bzeznik et al. (2017) discuss the challenges of using traditional package management systems on HPC clusters and how Nix can be used to overcome these challenges. It provides an overview of Nix and explains how it can be used to manage packages on HPC clusters (Bzeznik et al. 2017).

The authors also present a case study of using Nix on a large HPC cluster at the University of Oslo. They demonstrate how Nix was used to manage packages and dependencies, resulting in a more streamlined and efficient package management system.

Overall, the paper provides valuable insights into the use of Nix on HPC clusters and highlights the advantages of using a functional package management system in this context. It can serve as a useful reference for researchers and system administrators looking to improve package management on HPC clusters.

3.1 Deploying CESM on an HPC System Using Nix Package Manager

CESM 2.1 is a complex and powerful software tool that allows researchers to simulate and study the Earth's climate. Deploying this tool to a high-performance computing (HPC) environment like the UConn HPC can be a daunting task, as it requires managing many dependencies and ensuring that the software is compiled and configured correctly.

During my work with Assistant Professor Clay Tabor, we utilized the Nix package manager to help facilitate the deployment of CESM to the UConn HPC. Nix allowed us to define a reproducible environment that included all the necessary dependencies for CESM, ensuring that the software was compiled and configured consistently across different runs and different systems.

Furthermore, Nix provided us with the ability to easily manage the many dependencies required by CESM, including complex scientific libraries like NetCDF and MPI. With Nix, we could easily define and manage the versions of these dependencies, ensuring that the correct versions were used for each run of the CESM model.

3.2 Building and managing scientific packages with Nix

Access to high-performance computing (HPC) systems is often limited, and users may have only limited sudo access, which can make installing and managing software a challenge. In our work deploying CESM to the UConn HPC, we encountered these limitations, but we were able to overcome them with the help of community software, such as nix-user-chroot.

Nix-user-chroot is a tool that allows users to create a Nix environment inside a chrooted directory, without requiring root access. This is particularly useful for HPC systems where users may not have sudo access. By using nix-user-chroot, we were able to create a self-contained environment for our CESM installation, with all of the required dependencies and configurations.

When working on deploying CESM to the UConn HPC, we encountered a challenge with limited sudo access to the HPC, which meant that we could not install new software or access software outside of the pre-installed HPC module.

With limited sudo access on an HPC, it was challenging to manage software dependencies and versions. The HPC may have specific software modules installed, but they may not be the required version or have the necessary dependencies. Additionally, installing software locally without sudo access can result in conflicting dependencies and installations, leading to a broken environment.

However, with the help of the community software nix-user-chroot, we were able to create a separate namespace environment with its own software dependencies and configurations. This allowed us to install and use the necessary software for CESM without interfering with the HPC module. The namespace separation provided by nix-user-chroot created an isolated environment where the necessary software packages could be installed and configured independently of the HPC module.

3.3 Disadvantages of using Nix for Scientific Computing

While Nix has many advantages for scientific computing, there are also some disadvantages to consider. One potential drawback is the learning curve associated with using Nix. It can take some time to learn how to use the tool effectively, especially for those who are not familiar with functional programming concepts. Additionally, because Nix is a relatively new tool, there may be a lack of community support and resources compared to more established tools.

Another issue to consider is that many HPCs use a module system to manage software dependencies, which may not be fully compatible with the Nix package manager. While Nix can be used in conjunction with the module system, there may be some challenges when it comes to managing conflicting dependencies and ensuring that the correct software versions are being used.

Additionally, some HPCs may have specific configurations and requirements that are not fully supported by Nix. This can lead to compatibility issues and may require additional work to ensure that the software can be properly deployed and managed on the HPC.

Finally, there may be some software packages that are not yet available in the Nix ecosystem. While the Nix community has made great strides in recent years to add support for a wide range of packages, there may still be some gaps in coverage, particularly for more niche scientific software. Despite these potential drawbacks, the benefits of using Nix for scientific computing, such as reproducibility, dependency management, and ease of deployment, make it a powerful tool that is worth considering for scientific computing workflows.

4 Creating a Reproducible Development Environment with Nix

Currently, I am collaborating with Professor Clay Tabor on the development of a website that enables users to choose and display climate model outputs. To create this website, we are using Flask, a popular Python library for building web applications. However, building and managing the various dependencies required for the Flask framework has been a challenge. It has been difficult to ensure that the website is consistently built without having to patch and reinstall packages repeatedly, which can be a tedious and time-consuming process. The use of the Nix package manager has been instrumental in managing the dependencies and ensuring that our development environment is reproducible. With Nix, we can easily define and manage the various packages required for the Flask framework and other libraries, allowing us to focus on developing the website's functionality and features rather than worrying about package management.

4.1 Creating a reproducible python environment with Nix

In the context of building a website with the Flask framework, managing dependencies can be a daunting task. One of the dependencies required for our website is the MetPy package, which provides meteorological

analysis functionality for Python. However, installing and managing MetPy's dependencies was challenging, especially when trying to ensure a reproducible development environment.

```
metpy = with pkgs.python310Packages;
  buildPythonPackage rec {
    pname = "MetPy";
    version = "1.4.1";
    src = fetchPypi {
      inherit pname version;
      sha256 = "sha256-oT3S2jYOv9hWJw5BdG5P1Uutyp3NvYASKegDgs4x27k=";
    };
    buildInputs = with inputs.nixpkgs; [
      matplotlib
      numpy
      pandas
      pint
      pooch
      pyproj
      scipy
      traitlets
      xarray
      importlib-resources
      importlib-metadata
    ];
    doCheck = false;
  };
```

In the example code above, we can see how using Nix's `buildPythonPackage` function and `fetchPypi` defined in python standard library of nix allowed us to easily install and manage `MetPy`, a package with several dependencies that can be difficult to install and manage without a package manager. By defining the package using `buildPythonPackage`, we were able to specify the package name, version, source, and required dependencies in a declarative way. This made it easy to ensure that the correct version of each dependency was installed and that the package was built consistently across different systems. Additionally, by setting `doCheck` to false, we were able to skip running the package's tests during the build process, which can save time when building larger packages. With Nix and `buildPythonPackage`, we were able to manage `MetPy` and its dependencies in a reproducible and efficient way.

On a more technical note, the code snippet for MetPy as shown above is part of a Nix flake. In Nix, a flake is a declarative, reproducible way to describe a build and development environment. It allows you to specify all the dependencies required for your project, including system packages, libraries, and programming languages, in a single file. By defining a Nix flake, you can create a completely reproducible build of your website, ensuring that anyone who uses your code gets the same development environment and can reproduce the exact same build.

In our case, the Nix flake describes the entire build of the Flask website we are creating, including all

the Python packages required, such as MetPy. This means that anyone who clones our repository and runs the Nix flake will have the exact same development environment and can build and run the website with the same dependencies we used. This eliminates the "works on my machine" problem and ensures that the website is reproducible across different systems and environments. Overall, using Nix flakes has been essential in creating a consistent and reliable development environment for our Flask website project.

We find the store path containing the source code of the flake using the following

```
nix flake metadata | grep "Path" | awk '{print $2}'
```

We input the following shell command. During the execution of this command, the `sbomnix` github flake `nixgraph` attribute creates a dependency graph of the nix derivation,

`/nix/store/a2lnf0wrxx2k9zqj4x4pzghp2pv55sbw-python3-3.9.16-env.drv`.

```
nix run github:tiiuae/sbomnix#nixgraph \
/nix/store/a2lnf0wrxx2k9zqj4x4pzghp2pv55sbw-python3-3.9.16-env.drv -- --depth=2
```

This dependency graph is simply a `png` file and shows a diagrammatic view of the Flask dependencies and libraries. See figure 2 for reference.

5 Nix Ecosystem and Community

5.1 Overview of the Nix ecosystem

The Nix ecosystem is a collection of tools and utilities that provide a complete solution for creating and managing reproducible software environments. At its core, Nix is a package manager that is designed to provide a reliable, deterministic way of installing and managing software dependencies. Unlike traditional package managers, Nix is designed to work across multiple platforms and with multiple versions of the same software, which makes it particularly well-suited for scientific computing and other research applications.

In addition to the package manager, the Nix ecosystem includes a number of related tools and utilities that can be used to build, deploy, and manage software environments. These tools include:

- Nixpkgs: a collection of over 50,000 pre-built packages that can be easily installed using the Nix package manager.
- NixOS: a Linux distribution based on Nix that provides a complete operating system environment that is fully reproducible and easy to deploy.
- NixOps: a tool for deploying and managing Nix-based software environments in the cloud.
- Lorri: a tool for quickly iterating on Nix-based development environments.
- Hydra: a continuous integration and deployment tool that can be used to build and test Nix packages.

Taken together, these tools provide a powerful and flexible set of tools for creating and managing reproducible software environments. Whether you are working on a small research project or a large-scale scientific computing application, the Nix ecosystem provides the tools you need to ensure that your software environment is always reliable, reproducible, and up-to-date.

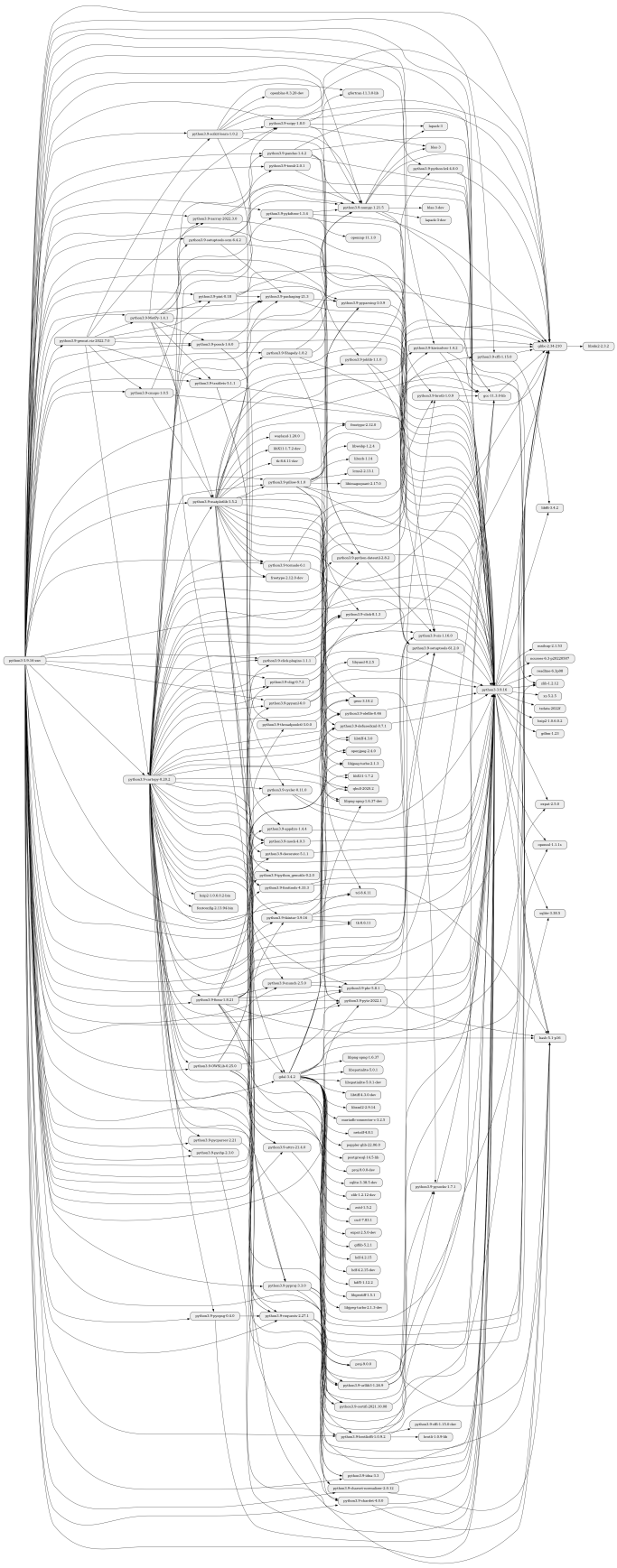

Figure 2: Dependency Graph of Flask Project

5.2 Community support and resources

One of the main advantages when using the nix ecosystem to deploy software is the community support. The nix community website, which provides a comprehensive documentation for Nix, including tutorials, guides, and reference manuals. The community also maintains a package repository, called nixpkgs, which contains a large number of pre-built packages that can be easily installed using the nix package manager.

The NixOS community provides various resources for users to get help and support, including the official forum, Discourse. In my experience, Discourse has been a great resource for troubleshooting and finding answers to specific questions related to Nix and its ecosystem.

For example, I recently asked a question on Discourse regarding the installation of several Python packages, including `cartopy`, `cmaps`, `geocat.viz`, and `metpy`, for a scientific computing project. Within a few hours, I received a response from a community member who fixed a build issue in the cartopy package by fixing the build issue in a pull request. Now the package installs perfectly using the nix package manager.

This kind of community support can be invaluable, especially for users who are new to the Nix ecosystem and may need guidance on how to navigate the various tools and features available. It also shows how active and helpful the NixOS community is, which can be reassuring for users who are considering adopting Nix for their own projects. Overall, the community support and resources available for Nix make it a powerful tool for reproducible scientific computing.

6 Conclusion

Nix provides a powerful package management system for scientific computing that can help solve many of the challenges associated with managing software dependencies and versions. Its reproducible builds and declarative package specifications make it ideal for creating consistent and isolated environments for scientific computing. Additionally, the community-driven approach to package maintenance and support ensures that many popular scientific software packages are available in the Nix ecosystem.

While there may be some disadvantages to using Nix, such as limited support for certain niche scientific software packages or the need to work around module systems on HPCs, the benefits of using Nix generally outweigh these challenges. As the scientific computing community continues to adopt and contribute to Nix, we can expect to see even greater support and functionality for scientific software in the future.

7 Acknowledgment

I would like to express my sincere gratitude to Professor Clay Tabor for his invaluable guidance, support, and encouragement throughout my research project. His expertise in scientific computing and his thoughtful insights have been instrumental in shaping this paper. I am also grateful for his patience and willingness to share his time and knowledge with me. Without his guidance, this paper would not have been possible.

I would also like to express my gratitude to the NixOS community and the Nix development team for their invaluable support and assistance during this research project. Their guidance and feedback have been instrumental in shaping my understanding of Nix and its role in creating reproducible environments for scientific computing.

References

Bzeznik, Bruno, Oliver Henriot, Valentin Reis, Olivier Richard, and Laure Tavard. 2017. "Nix as HPC Package Management System." In *Proceedings of the Fourth International Workshop on HPC User Support Tools*, 1–6.

Devresse, Adrien, Fabien Delalondre, and Felix Schürmann. 2015. "Nix Based Fully Automated Workflows and Ecosystem to Guarantee Scientific Result Reproducibility Across Software Environments and Systems." In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, 25–31.

Dolstra, Eelco, Merijn De Jonge, Eelco Visser, et al. 2004. "Nix: A Safe and Policy-Free System for Software Deployment." In *LISA*, 4:79–92.

Ivie, Peter, and Douglas Thain. 2018. "Reproducibility in Scientific Computing." *ACM Computing Surveys (CSUR)* 51 (3): 1–36.