Technological Feasibility

November 03, 2021

Fossilized Containers

Team Members:

Jadon Fowler

Jeremy Klein

Emily Ramirez

Mumbi Macheho-Mbuthia

Sponsor:

Dr. Nicholas McKay

Mentor:

Melissa D. Rose

1 Introduction

Climate change is a loaded word. It is fantastical that the world may no longer be a home, but yet it remains, constantly worsening, constantly present. From the 19th century to now, the global average temperature has risen 1.18 degrees celsius [1]. While it appears to be negligible, this small change in temperature contributes to increased droughts, heatwaves, wildfires, and more extreme weather conditions. When the public discusses climate change, the focus stays on the present and the looming future. **Paleoclimatology**, the study of past climates, takes a different approach. By understanding how Earth's climate has changed over the past several thousand years, scientists can predict and prepare for changes in the future.

Creating climate reconstructions is an important facet of studying climate change. Reconstructions are maps that show different metrics, one being temperature, of a region across different points in history. Dr. Nicholas McKay is a researcher in the Paleoclimate Dynamics Laboratory (PDL) at Northern Arizona University working to keep reconstructions updated with new datasets and techniques. PDL, along with collaborators at the University of Southern California, have worked to create a system that gathers reconstructions and provides constant updates. This is the Paleoclimate Reconstruction Storehouse, otherwise known as **PReSto**.

With PReSto, Dr. McKay and collaborators can accept different datasets and models from different researchers and submit them to their system. With the thousands of types of datasets, however, it is difficult for researchers to submit their code to PReSto without an existing standardized way to review them. Scientists use a multitude of programming languages, libraries, dependencies, and operating systems that are not guaranteed to work on other systems. **Containerization** is a way to package software so that it will be compatible across different host operating systems. It also allows users to test a model without having to install different libraries or dependencies. They simply need to build a container and run it to view the model.

A current problem within the research community is that not every climate scientist has the skillset to create their own containers since it is newer technology. Having scientists who are trying to contribute to PReSto learn the intricacies of containerization is inefficient. PReSto aims to create a streamlined process so that scientists can contribute their models in a more efficient manner. This is what the team, Fossilized Containers, is going to solve. The team will create a command line interface, **CLI**, for scientists interacting with PReSto so that they can submit their models in an efficient manner that is easily accessible and language agnostic. The CLI is bundled within the **Fossilized Controller**, the overarching tool that helps scientists build and communicate with their containers. The tool will guide the user through the container creation process with prompts for clarification. From there, they can run the models or send files with the help of **adapter libraries**. The libraries are added to the model code so that the Fossilized Controller can adequately communicate with the containers.

The rest of the document will be covering the challenges that arose when designing the structure and flow of the Fossilized Controller. First, an outline of the four major challenges that will be faced for the different components of the project. After a short introduction and description, the challenges are discussed individually and much more in depth, covering the possible solutions and their benefits and downfalls. Once the possible solutions have been covered, the chosen solution will be introduced as well as the reasons behind that choice.The final part of this section of the document will explain how the different options will be tested and why it is crucial in order to ensure that there are no major flaws in the structure and plan for the project. It also allows us to give our client to review what programming languages, data structures, or connections we plan to use and the thought process behind each of them.

2 Technological Challenges

Part of the PReSto project is defining standards for climate reconstructions to use. Dr. McKay has standardized the Linked Paleo Data (**LiPD)** file format containing annotated paleoclimate data collected from real world samples. This format is used as the input to

climate reconstructions. Another file format, the Network Common Data Form (**NetCDF**), is used as the output for climate reconstructions. These climate reconstructions can also take in arbitrary parameters that change the underlying model. Together, these form the standard input and output for climate models.

In order to standardize the containerized climate models, the team will build a few components that will facilitate the interactions with **PReSto Containers**, which are climate reconstruction programs that have been containerized. These components will define input LiPD files, climate model parameters, and output NetCDF files to be passed around using an HTTP connection between the Fossilized Controller and the climate reconstruction program. These components include the following:

1. The **Fossilized Controller** will maintain a connection to the PReSto containers to send & receive files.
2. The Controller will also contain a **Command Line Interface (CLI)** for managing the running of the PReSto containers.
3. Python and R versions of an **Adapter Library** which the climate model program will use for standardized communication with the Controller from within a Docker container

The **Fossilized Controller** (herein referred to as the “Controller”) is a standalone program that includes a CLI for creating and managing PReSto Containers. PReSto Containers are **Docker Containers** that utilize our Adapter Libraries and contain a climate model. The Controller is responsible for creating this Docker Container by generating the necessary files, and communicates with the Docker Engine running to manage the activities of the climate reconstruction within the container. This process is defined by the technological components we analyze in section 3 and their integrated whole in section 4.

3 Technological Analysis

Each of these components contains several technology choices that can greatly affect the speed and efficiency of our development process and resulting code. For each of these choices, we present an in-depth analysis of available options, weighing the pros and cons for our use case, and argue why our choice is a good fit.

3.1 Connection between Fossilized Controller and PReSto Containers

For PReSto Containers to be used by external users and services, the Fossilized Controller will facilitate connections to the containers and control what they are running. Since PReSto Containers are actually Docker Containers, we can use existing mechanisms to communicate with the climate model program inside, but a standardized protocol must be created for the Controller to properly manipulate the containers.

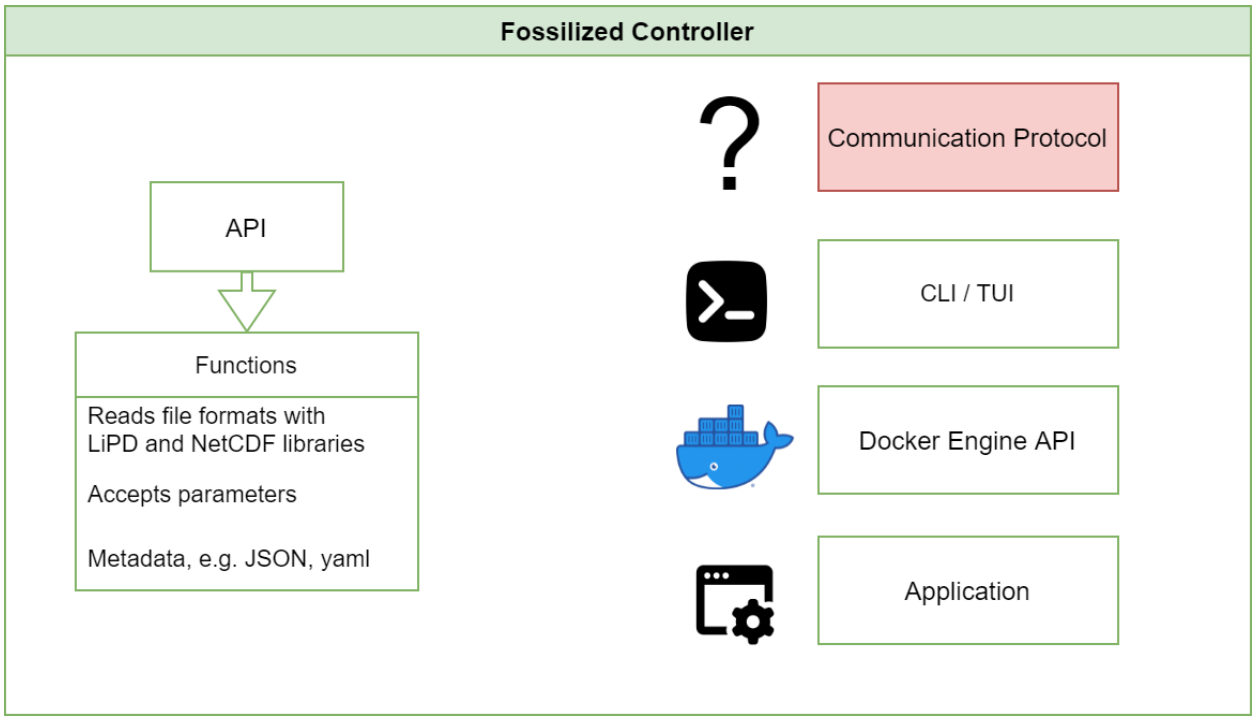


Figure: High level overview of the controller showing the different components within it and the missing communication protocol.

We need this protocol to send and receive files and metadata. This can be done by Docker Containers in a few different ways. The first approach is using a **Docker Volume** which is useful for persisting data generated by the container. Volumes are stored in a part of the host filesystem that is managed by Docker. It acts as a “filesystem in a filesystem” for the container to use for storing data between runs. It is not recommended for other processes to manipulate the files within a Volume. Using Volumes, climate model programs could store their resulting NetCDF files, but the Controller wouldn’t (or really, shouldn’t) be able to write the input LiPD files to a PReSto Container’s Volume.

Another approach would be to use a **Bind Mount** which mounts a directory in the host machine to the Docker Container. This makes the files in the directory usable by both the container and the Controller. Bind Mounts might work well for our usage, but giving containers the ability to arbitrarily manipulate files in the host system is insecure and could impact other programs running on the host system. This risk shouldn’t be taken since we are uploading arbitrary climate model programs that may have bugs or malicious code inside them.

The chosen approach for communicating between the Controller and PReSto Containers is to use an **HTTP Server & Client** for sending LiPD and NetCDF files and related metadata. The **Adapter Libraries** will contain an HTTP Server that accepts LiPD files and returns NetCDF files, and the opposite end will be built into the Controller using an HTTP Client. This will be how the Controller interacts with the PReSto Containers. An HTTP Connection also provides much better security than Volumes or Bind Mounts and doesn’t require the Controller to be running on the same filesystem as PReSto Containers.

	Obtain NetCDF Files	Obtain Parameters	Security	Final Score
Volumes	✓	✗	★★★☆☆	★★★☆☆
Bind Monuts	✓	✓	★☆☆☆☆	★★☆☆☆
HTTP	✓	✓	★★★★★	★★★★★

Figure: Comparison of Volumes, Bind Mounts, and HTTP Connections in relevant areas of interest.

To test the feasibility of sending data over HTTP, we will be building an HTTP Server in the Adapter Libraries that supports accepting LiPD files and returns NetCDF files. We will use fake data until all of the components are completed and we can run a real climate model program. Additionally, an HTTP Client will be created in our Controller program that is capable of sending the proper files to the HTTP Server in PReSto Containers.

3.2 Command Line Interface for Fossilized Controller

For users and external services to communicate with the Fossilized Controller and the PReSto containers it owns, the Controller will come bundled with a Command Line Interface (CLI). The CLI is a multifaceted tool that will utilize the Docker Engine API to interact with the Docker daemon. It is meant to encompass the different commands of Docker itself without requiring the user to have extensive knowledge of containerization. The primary purpose of the CLI is to guide a scientist through the containerization process through multiple prompts asking about their program. There are also standard Docker commands integrated into the tool so that scientists do not have to switch between different tools.

```
# Using an example model
~/projects $ cd ./Temp12k/


# guide the user through the creation process for the Dockerfile & other
metadata, creating prompts like "Are you using R? [Y/n]: "
~/projects/Temp12k $ presto create --maybe --some --flags --here
Are you using R? [Y/n]: Y
Creating PReSto Project...


# on the user's computer, they can run the PReSto (Docker) container with
~/projects/Temp12k $ presto run --some --other --optional --flags
Running PReSto Project Temp12k...
```

Figure: Example usage for the CLI

A programming language is therefore needed that is compatible with the Docker API and allows for a deeply customizable tool. The first candidate is **Go**, which is the language Docker itself is built off of. Go has a strong presence in cloud computing and is often the language of choice for CLIs. It is usually preferred by companies because it was initially built with speed and Google's infrastructure size in mind. Docker has official software development toolkits (SDKs) for two languages, with Go being one of them. This allows the CLI to interact with the Docker daemon. While Go is a new language for most of the team, it is not difficult to pick up and has simple syntax. Despite it being easy to pick up, there would still be a longer learning curve for the team to get comfortable with the language. This would result in longer delays when creating the CLI tool.

Another language of interest is **Java**, an object oriented language. Its speed is comparable to Go because they are both compiled languages. The issue with Java that makes it the least attractive candidate is that it does not have an official Docker SDK or library. There are many community libraries for Java, however, they are either incomplete or have no active maintenance. Setting it up properly is also more difficult as

you have to add community Docker libraries as dependencies or do extra work as per the specific client's requirements. An officially supported and maintained SDK or library is therefore more preferable.

The chosen language for creating the CLI that users interact with is **Python**. It also has an official SDK for Docker. Unlike Go and Java, which are compiled languages, Python is interpreted. This means that its speed is significantly hindered compared to the other languages. Despite it's lowered performance, Python is a language the entire team is familiar with and would not have a large period for self-training. When comparing the SDKs of Python and Go, implementation on Python is more efficient for the team. Starting a container in Python is only 3 lines, compared to the 50+ required for a Go implementation. Both languages are equal in complexity when it comes to creating a simple CLI tool, so the team ultimately decided on Python. It is a language the entire team is familiar with and integrating the Docker API is not difficult.

	Docker API Compatibility	Speed	Team Experience	Overall Score
Go	✓	★★★★☆	★★☆☆☆	★★★☆☆
Java	✗	★★★☆☆	★★★★★	★★★☆☆
Python	✓	★★☆☆☆	★★★★★	★★★☆☆

Figure: Comparison of different programming languages for the CLI across relevant fields of interest.

To test the feasibility of using Python to create the CLI, the team will create sample tools that use basic Docker functionalities and use a containerized PReSto model. The tool will start by doing basic commands, such as pulling and starting a container. Doing so

ensures that Python will meet all of the requirements needed for the CLI in an efficient manner.

3.3 Python Adapter HTTP Server Choice

Climate model programs within PReSto Containers that are written in Python will be using our provided Python Adapter Library to communicate with the Fossilized Controller. Communication will be handled by a Python HTTP Server included in the Adapter Library that is running within every PReSto Container, and an HTTP Client embedded into the Controller.

There are a number of libraries that would provide the kind of connection that is required. The first option is the HTTP server and client included in the Python standard library. The `http.server` and `http.client` modules require a large amount of syntactic overhead (the amount of code needed to do simple tasks is large). Using the standard library does not increase container size, but the resulting code's explicitness makes the Adapter Library difficult to read and extend. This makes it a poor choice for our use case.

Another option is **Django**, a web framework designed for full scale website development. It provides superb extensibility, offering first and third party modules for adding database connectivity, templating, and several other useful features for website development. Django is very large and would increase the container size by a substantial amount, as well as produce a moderate amount of syntactic overhead when building the HTTP server. Most of the features it provides are unnecessary for our use case, making this another poor choice.

The final option and our selected approach is to use **Flask**, a microframework for building HTTP servers quickly. Since it is a microframework, its compactness won't inflate container sizes and makes it very simple to build an HTTP server that sends and receives files. Flask doesn't provide as many features as Django, however the Adaptor Library's use of the server is very limited. This makes Flask ideal for us.

	Library Size	Extensibility	Syntax Overhead	Overall Score
Standard HTTP Libraries	✓	★☆☆☆☆	✗	★★☆☆☆
Flask	✓	★★★☆☆	✓	★★★★★
Django	✗	★★★★★	✓	★★★★☆

Figure: Comparison of different Python libraries for the HTTP server across relevant fields of interest.

In order to test the functionality of the different HTTP server libraries, we are going to implement them to mimic what the adaptor library will be doing. The client will send a POST request to the already running server inside of the container to run the climate model program. This POST request will contain the LiPD files that will be used in the computation. Once the computation is complete the server will send a response message containing the resulting NetCDF file.

3.4 R Adapter HTTP Server Choice

The issue at hand is how a PReSto container will communicate with the controller. We decided that each PReSto container would run an application that the controller can communicate with to get and request information. So, the issue has evolved to which R package to use for R-based PReSto containers to create and maintain this communication application.

The characteristics needed for this feature rely on the ability to receive and react to HTTP POST requests. The initial search for R packages that handle HTTP led to a web page maintained by R that lists all packages related to web and server frameworks.

Keeping the goal in mind, we picked three candidates: `httpuv`, `opencpu`, and `shiny`. From a glance, they all seemed to provide the functionality needed to accept and handle HTTP requests.

	Creates Server	Ease of Use	Security	Overall Score
httpuv	✓	★★★★★	★★☆☆☆	★★★★☆
OpenCPU	✓	★★★☆☆	★★☆☆☆	★☆☆☆☆
Shiny	✓	★★★★☆	★★★☆☆	★★★☆☆

Figure: Comparing `httpuv`, `opencpu`, and `shiny` R packages in regards to characteristics needed to create an HTTP server in R

To test and compare the packages, a server prototype was created for each one to send a simple HTTP package to an R client. Each category is ordered top to bottom in order of importance, with the top category being the most important.

Whether the packages created its own server was a binary choice. Ease of use was much more subjective: while coding the aforementioned prototypes, we documented the experience and ranked each package from 1 to 5, with 1 being the worst. Security was a much harder category to measure. Because of our lack of knowledge with Internet security, we used documentation provided about the packages to rate each package.

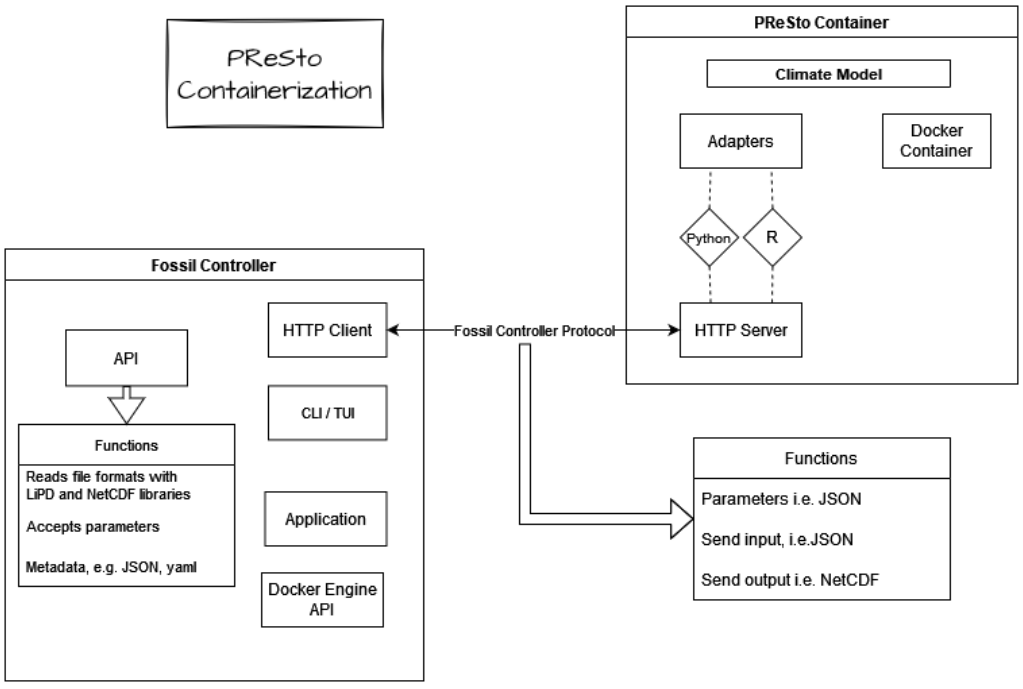
Using these findings, we determined that `httpuv` was the correct package to implement our R adapter. While its security was less than adequate, the ease of use

was the second most important characteristic to us. This project consists of many parts, most of them completely new. The easier it is to pick up new dependencies, such as `httpuv`, the more effective our team will be.

While security is not an issue now, moving forwards the requirements of each feature will be fully defined and the amount of security needed will be addressed.

4 Technological Integration

Together, these components form the building blocks of the PReSto containerization process. This process will be followed by climate scientists in order to properly containerize their code and make their climate models compatible with the rest of the PReSto system.



In this figure, we describe the components in detail. The Fossilized Controller will communicate with PReSto Containers over HTTP using an HTTP Client embedded in the Controller and an HTTP Server in every Adapter Library used by PReSto Containers. This communication will follow a standardized protocol to ensure the climate model programs properly receive the required LiPD files and parameters. These files and parameters are specified by a user or external service and sent to the Controller through its CLI. They are sent to the container and retrieved from the container will be a NetCDF file containing the data emitted from the climate model program.

5 Conclusion

As of now, climatologists can not easily share their models with one another. These models create a historical overview of climate change. As the subject of climate change gains more focus, these models give insight to what is entailed for future climates and what may occur in the present.

Our task is to use containerization to give climatologists the ability to communicate their models with one another. It appears simple, however creating and managing language-agnostic, standardized containers for a client base unfamiliar with containers is an involved project with big challenges.

Using containerization as the foundation, however, we have been able to identify four key problems and their potential solutions.

1. How will PReSto containers communicate with our newly designed Fossilized Controllers?
2. What language do we use to build the command line interface tool?
3. For our Python adapter, what will we use to run a server inside PReSto containers for models written in Python?

4. For our R adapter, what will we use to run a server inside PReSto containers for models written in R?

We have tested these potential solutions against one another using their desired characteristics to determine the best candidate for each problem. Our next step is to define the goals of our product, to ensure our client's vision is understood and to move forward with the project with the correct destination in mind.

References

[1] “Climate change evidence: How do we know?,” *NASA*, 12-Oct-2021. [Online]. Available: https://climate.nasa.gov/evidence/. (Accessed: 18-Oct-2021).