SOFTWARE DESIGN DOCUMENT

Version 2.0

March 2, 2022

Fossilized Containers

Sponsor: Dr. Nicholas McKay

Mentor: Melissa D. Rose

Team Members: Jadon Fowler, Jeremy Klein,

Emily Ramirez, Mumbi Macheho-Mbuthia

Table of Contents

1. Introduction……………………………………………. 2

2. Implementation Overview…………………………… 3

3. Architectural Overview……………………………….4

**4. Module and Interface Descriptions……………………
6**

4.1 Container Manager Module…………………………..6
4.2 Command Line Interface Module…………………… 7
4.3 Adapter Module………………………………………..9

**5. Implementation Plan…………………………………
13**

6. Conclusion…………………………………………….14

Glossary……………………………………………………..15

References... 16

---

1. Introduction

Climate change is a term that immediately catches attention. The Earth is becoming increasingly inhabitable, 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 temperature change 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. However, **paleoclimatology**, the study of past climates, takes a different approach. By understanding how Earth's weather has changed over the past several thousand years, scientists can predict and prepare for changes in the future.

Paleoclimatologists create code. This code reconstructs models of the past called paleoclimate reconstructions or **PRs** [2]. PRs are usually written in the programming languages R or Python. With **PReSto**, a Paleoclimate Reconstruction Storehouse, Dr. McKay and collaborators can accept different datasets and models from researchers and submit them to their system. With thousands of datasets, researchers have difficulty submitting their code to PReSto without an existing standard.

Scientists use a variety of programming languages, libraries, dependencies, and operating systems that are not guaranteed to work on other systems. **Containerization** packages software to be compatible across different host operating systems. It also allows users to test a model without installing libraries or dependencies. They only need to build a container and run it to view the model. The **Fossilized Controller** is a boon since containerization is not easy for these scientists. The tool allows for the easy creation of containers and an online guide that walks through this creation. The project consists of three major components: a command line interface, a container manager class, and an adapter library; the rest of this paper will break down these components.

2. Implementation Overview

To understand the envisioned solution of this project, the technology that the solution relies on needs some explanation. **Docker** is a container service that allows you to create, destroy, and manage containers, specifically Docker Containers. A **Docker Container** is Docker's implementation of a container. **Docker Hub** is a platform to host Docker Containers and is an easy way to share containers. The **Docker Daemon** is the service that handles the management of Docker Containers.

The first part of the team's solution vision is a **command line tool**, or **CLI**, that will guide scientists through the process of creating Docker Containers that contain their PRs. The container manager and the CLI interact with the Docker Daemon to perform any action involving Docker. The CLI uploads the containers to Docker Hub on a basic account created solely for this product. The Docker Hub account will have a public repository with reconstructions tagged to differentiate between them. Users can access containers by interacting with the CLI or viewing the project's Docker Hub account.

The second part of the program runs containers on an external service such as an external remote server. For reconstructions that require more computing power, a remote server can provide the resources to run intensive computations. Scientists who want to run a PR can send files and parameters to the program and receive back results. On the backend, the CLI downloads the Docker Containers from the public Docker Hub repository on the external service hosting it. The program will then run the necessary container with the files and parameters given by a scientist and send back the results. The solution is developing with a general external service in mind, so an exact service is not provided.

Next, this document will examine the program's architecture from a top-level perspective. This examination will cover the main modules and their responsibilities, how they communicate with one another, and the main workflow of the program concerning its modules.

3. Architectural Overview

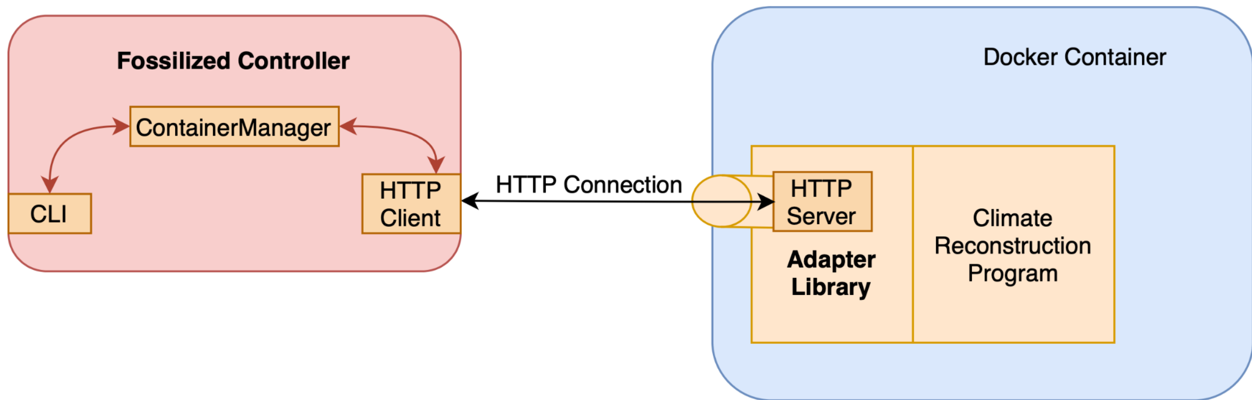

Figure 1: Architecture Diagram of the Fossilized Controller

The architecture contains two primary components: the *Fossilized Controller* (aka "the controller") and the *Adapter Libraries* (aka "adapters"). The controller is a standalone Python package that users can interact with to connect to the climate reconstruction programs they want to run. By interacting with the Command Line Interface (CLI), users can bundle their code into a Docker Container which is internally managed by the controller. The controller uses the Docker SDK to spawn, run, and destroy these containers. These containers are expected to have an adapter that can communicate with the controller. Initially, adapters for Python and R will be provided, but these can be expanded to any programming language. These adapters reveal a standard interface so, no matter how a climate reconstruction program is structured, the controller can send LiPD files and receive NetCDF files. These files are the standard input and output for these climate reconstruction programs: LiPD files containing environmental data are transformed into NetCDF files containing the result of the climate model. The climate models within the Docker Containers can be configured by passing model-specific parameters to the controller through the CLI, which are then passed to the adapter library within the specified container. This is the main workflow users will go through to run their climate models.

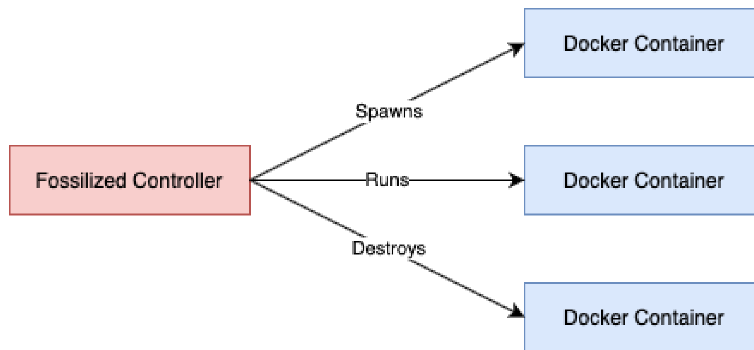

Figure 2: Fossilized Controller Managing Docker Containers.

The controller's main job is the overall management of Docker Containers. By connecting with the Docker Daemon, the controller can spawn, run, and destroy Docker Containers at the will of the user. The controller does this by communicating with the Docker Daemon through its standard API. Since both of these pieces are written in Python, this communication is simple. Users can do these actions by using the controller's CLI, which will explain the available commands. The CLI is built using a Python package called *Click* which makes it easy to implement usable commands. Once the connection to a container is determined, an HTTP Client is created to communicate with the HTTP Server within the adapters. LiPD and NetCDF files are transferred over this connection.

The adapters provide a standardized way to communicate with climate reconstruction programs within Docker Containers. These are standardized so, no matter what a climate scientist's code may look like, the program can run their model. When the Docker Container is spawned, an HTTP Server within the adapter is started. This server waits for the controller to contact it. When a connection is established, model parameters and LiPD files are sent to the adapter and passed to the reconstruction program. NetCDF files are returned through the same connection.

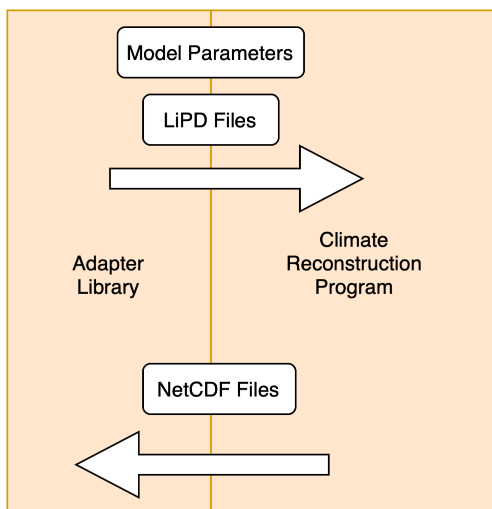

Figure 3: Interactions Between the Adapter Library and Climate Reconstruction Program

4. Module and Interface Descriptions

This section describes the different modules within the architecture. It includes diagrams and relevant functions.

4.1 Container Manager Module

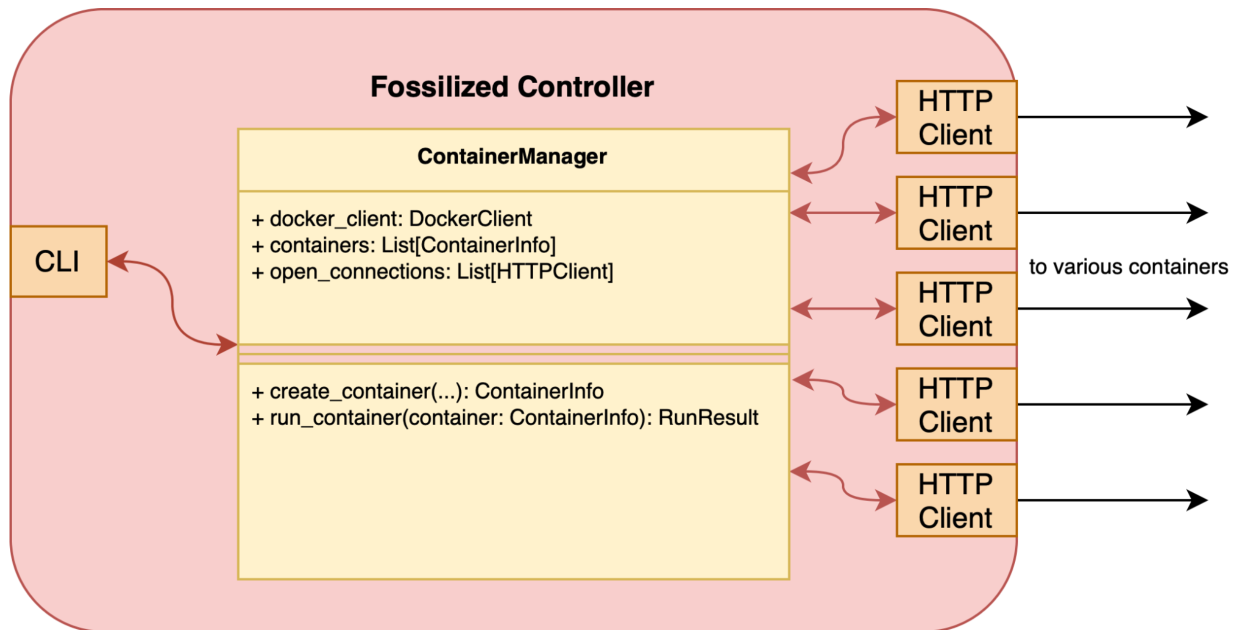

Figure 4: Container Manager Diagram

The brain of the controller is the *ContainerManager*. It contains the state of the containers and controls the associated HTTP Clients that connect to them. Because this component and the CLI are in the same Python package, they can communicate seamlessly. The interface provided allows for creating and running containers.

- *Function*: **create_container(image: String) -> ContainerInfo**
    - *Parameters*: The name of the image the functions uses to create a container
    - *Outputs*: A reference to the container it has created in memory
    - *Description*: Calls the Docker SDK to create a container from the specified image. This reference it returns contains information about the container.
- *Function*: **run_container(container: ContainerInfo)**

- *Parameters*: A reference to the container to run
- *Outputs*: Data created by the climate reconstruction program in the container
- *Description*: This function uses the Docker SDK to run the climate reconstruction program inside the specified container and outputs the resulting data.

4.2 Command Line Interface Module

The most important part of the PReSto system is the command line interface. The CLI is how the user will interact with containers and create containers easily on their own machine. Inside the CLI, several functions create, run, and manage containers. These functions are possible due to the Docker Software Development Kit, a Python library to access core Docker functionalities.

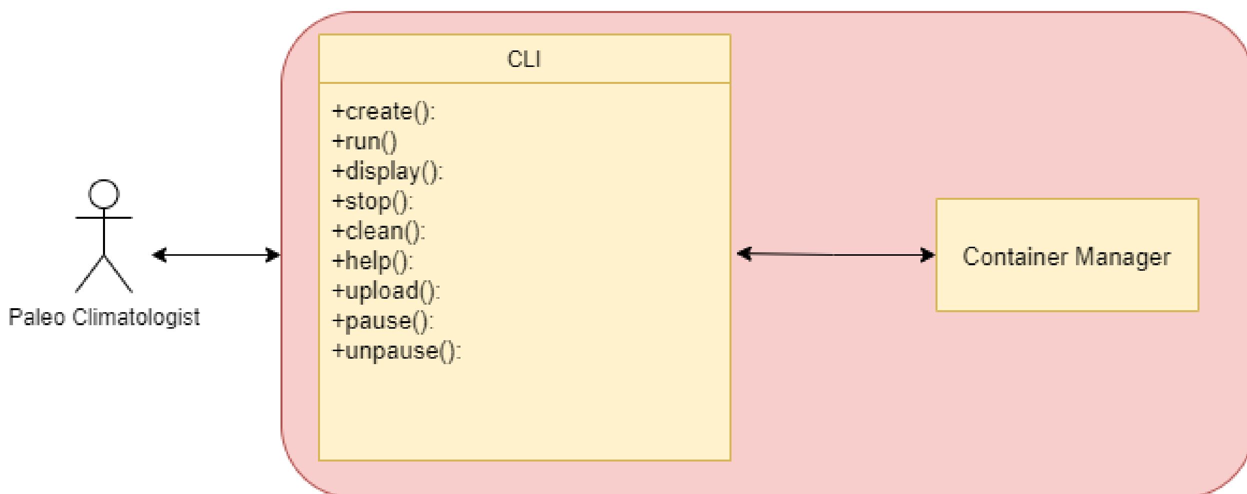

Figure 5: UML diagram of the Command Line Interface

The command line interface consists of nine functions depicted in Figure 5. *create* is the primary function since it produces a Dockerfile, a file Docker uses to create containers. The *run* function is second in importance because it will run containers that the scientist has created or downloaded. The remaining methods are features that are not required. However, they make the tool more accessible and allow users to perform functions on containers that would typically not be easy to implement or understand.

- Function: **create()**
    - User Input: Various user prompts
    - Outputs: A Dockerfile

    - Description: A step by step process that guides users through the containerization process. Users are met with multiple prompts asking for specifics of their project to create the appropriate Dockerfile. An appropriately named Dockerfile is saved in the users local directory.
- Function: **run()**
    - User Input: Container name
    - Outputs: Result of attempt to run a container
    - Description: Takes in the name of a container that the user wants to run and starts the container.
- Function: **display()**
    - User Input: N/A
    - Outputs: Formatted list of containers and their status
    - Description: Displays a list of all of the containers and various information about them, such as their name, status, and uptime. The list is formatted so that it is easily readable.
- Function: **stop()**
    - User Input: Name of the container
    - Outputs: Result of the operation
    - Description: Takes in the name of a container and attempts to stop the container from running, if the container exists.
- Function: **clean()**
    - User Input: N/A
    - Outputs: Result of the operation
    - Description: This command will clear out the cache of containers currently on the machine. This function takes no parameters and does not return anything, it simply prints the result of deleting the cache.
- Function: **help()**
    - User Input: N/A
    - Outputs: Link to the documentation
    - Description: This function prints the url to the program's documentation so users gain access to more information on how to use the CLI tool.
- Function: **upload()**
    - User Input: The container name and repository (Docker Hub by default)
    - Outputs: Result of the operation

    - Description: Takes in the container that they want to upload. Users can specify the repository.
- Function: **pause()**
    - User Input: Name of the container
    - Outputs: N/A
    - Description: Takes in the container name and pauses the container if it is currently running.
- Function: **unpause()**
    - User Input: Name of the container
    - Outputs: N/A
    - Description: Takes in the container name and unpauses the container if it is currently paused.

4.3 Adapter Module

An adaptor's purpose is to handle communication between the Controller and the Docker Container that houses the adapter. The adapter receives files and parameters from the Container Manager, which gets these files and parameters from the CLI, which obtains those files and parameters from the user.

The adapter communicates with the Container Manager via an HTTP connection. The adapter serves as the HTTP server; the Container Manager serves as the HTTP client. The adapter communicates with the Climate Reconstruction program with a call in the Climate Reconstruction's programming language. Because of this, the adapter is in the primary languages for climate reconstruction programs, Python and R.

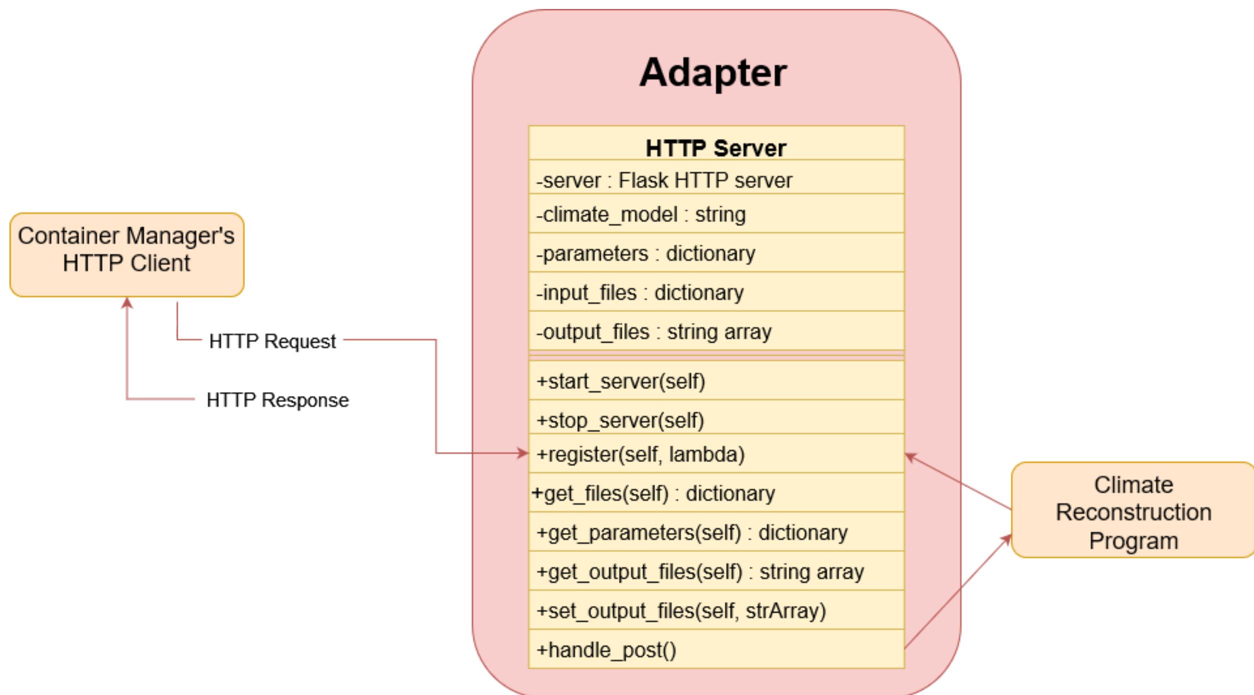

Figure 6: Adapter Library Diagram

The diagram above displays the parts that make up the adapter and the adapter's relation to other modules in the program. The adapter directly connects to the Container Manager's HTTP Client and the Climate Reconstruction program. Below are descriptions of the public interface methods.

- Function: **start_server(self)**
  - User Input: N/A
  - Outputs: N/A
  - Description: Starts the HTTP server in the adapter object.
- Function: **stop_server(self)**
  - User Input: N/A
  - Outputs: N/A
  - Description: Stops the HTTP server in the adapter object.
- Function: **register(self, callback)**
  - User Input: String
  - Outputs: N/A
  - Description: Saves "callback" in the adapter's "reconstruction" field. "callback" is a string that represents a Python code block which runs the climate

reconstruction if executed. If “callback” is more than one line, then “callback” needs to be a triple-quoted string.

- Function: **get_files(self)**
  - User Input: N/A
  - Outputs: Dictionary
  - Description: Returns the dictionary held in the adapter’s “input_files” field.
- Function: **get_parameters(self)**
  - User Input: N/A
  - Outputs: Dictionary
  - Description: Returns the dictionary held in the adapter’s “parameters” field.
- Function: **get_output_files(self)**
  - User Input: N/A
  - Outputs: String array
  - Description: Returns the string array held in the adapter’s “output_files” field.
- Function: **set_output_files(self, str_array)**
  - User Input: String array
  - Outputs: N/A
  - Description: Appends the values of “str_array” to the adapter’s “output_files” parameter. Repeat values will be ignored.
- Function: **handle_post(self)**
  - User Input: N/A
  - Outputs: N/A
  - Description: This function is called when a HTTP POST request hits the adapter’s HTTP server.
    1. Parses the metadata.JSON file from the request’s payload. The metadata.JSON file has two parent string keys: “parameters” and “input_files”. The “parameters” key holds the parameters for the climate reconstruction. The “input_files” key holds the name and the location of each file the controller sends to the adapter’s server. An example of the JSON file is shown below in Figure 7.
    2. Saves the key-values from the “parameters” key into the adapter’s “parameters” field as a dictionary. Saves the key-values from the “input_files” key into the adapter’s “input_files” field.

3. The other files in the HTTP Request POST payload are stored in the adapter's input_files field with the file name as the key and the file data as the value.
4. Runs the reconstruction by evaluating the expression held in the adapter's "reconstruction" field. This is done by using the Python *exec* method.
5. Compresses the files in the adapter's "output_fields" into a zip file and sends the zip file back to the controller in an HTTP Response message.

```
{
    "parameters": {
        "spread": true,
        "duration": 3000,
        "binvec": [1, 2, 3, 4, 5, 6],
        "localization_radius": "None",
        "data_dir": "/projects/pd_lab/data/data_assimilation/",
        "exp_name": "default",
        "models_for_prior": ["trace_regrid"],
        "assimilate_selected_seasons": ["annual","summerOnly","winterOnly"]
    },
    "input_files": {
        "lipd_file": "lipd-files/GeoB9307_3.Weldeab.2014.lpd",
        "netcdf-file": "nc-files/WMI_Lear.nc"
    }
}
```

Figure 7: Example metadata.JSON file

5. Implementation Plan

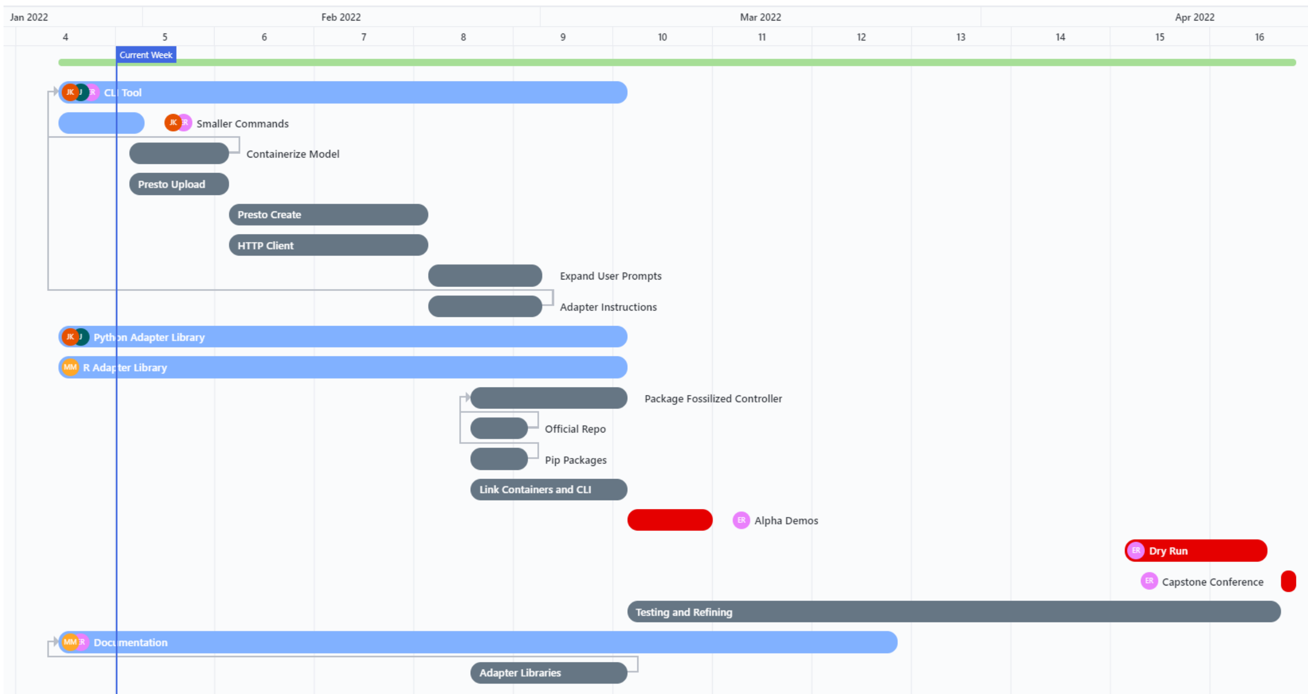

Figure 8: Gantt Chart

The entirety of the Fossilized Controller is split up into the development of the CLI tool, the Python/R Adapter libraries, and the documentation. The CLI tool is focused on developing the commands that users can implement to create and manage the containers of their reconstruction models. Development is planned for the beginning of January until the beginning of March. The commands have been grouped by sections based on the complexity of the implementation. More time is given to commands with higher complexity. The Python and R libraries are developed at the same time as the CLI and have the same development dates. The adapter libraries have the same functionality but are created for their respective languages. They are in parallel to the CLI because they are needed for scientists to communicate with their created containers. The entire Fossilized Controller enters the testing and refining stage at the beginning of March and will continue until the middle of April. The last major development phase is documentation. It will run from the beginning of January until the end of March. The documentation is important so that the scientists can understand how to use the tool and have a reference to the different commands. As it stands right now, the team has finished developing

the core functionality of the CLI for guiding scientists through the containerization process. What needs to be done is to further expand on the user prompts that ask the scientist about their model as the CLI only has baseline questions

Multiple members are working on each phase throughout the entire development process. The phases are developed in parallel so members will work on multiple parts at the same time. Generally, each phase has two people working on it at once. The figure below is a table showcasing which members are working on the different parts.

	Fossilized Controller CLI	**Python Adapter**	**R Adapter**	**Documentation**
Members	Emily Ramirez, Jeremy Klein	Jeremy Klein, Jadon Fowler	Mumbi Macheho-Mbuthia	Emily Ramirez, Mumbi Macheho-Mbuthia

Figure 9: Milestone Member Assignments

---

6. Conclusion

Paleoclimatologists build models as their main output which helps to better understand and combat climate change. These models, code to create climate representations of the past, are not built with different systems in mind. Containers can help solve the issues created by using models on different systems. In order to create these containers, the tool will consist of four major parts that all help in containerizing paleoclimatologists' code. These components consist of a command line interface, which allows the user to create, run, delete, as well as several other features straight from the command line. There is also a Python and R adapter library that reads LiPD files over an HTTP server to give to the climate reconstruction model as well as receive output files from the container. The last part of the project is the container manager, this is what the command line interface will interact with in order to perform functions on the created container objects. Altogether the four modules should work together well to create a simple process for climate scientists. Overall, this paper helped plan the implementation of different components and ensure they all perform the required functionality of the final project.

Glossary

CLI: Common Line Interface

Containerization: Packaging software so that it will be compatible across different host operating systems.

Containers: A way to package software so that it is lightweight and compatible across multiple operating systems.

Paleoclimatology: The study of climates of the past

CR: A climate reconstruction program or code the creates models of climates

PReSto: Paleoclimate Reconstruction Storehouse, a system that allows for the storage of Paleoclimate Reconstructions or PRs

LiPD: A way to store and exchange paleoclimate data in a standardized format

References

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

[2] PReSto A paleoclimate reconstruction storehouse. *USC Climate Dynamics*. [Online]. Available: https://climdyn.usc.edu/projects/5_presto/. (Accessed:16-Nov-2021).