SOFTWARE TESTING PLAN

Version 1.1

April 1, 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

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, specifically climate reconstructions or **CRs**. CRs produce climate models or climate maps [2]. With **PReSto**, a Paleoclimate Reconstruction Storehouse, Dr. McKay and collaborators can accept different reconstructions from researchers and submit them to their system. Dr. McKay has spearheaded PReSto as a centralized warehouse for CRs that allows for accessibility and standardization.

To create CRs, scientists use a variety of programming languages, libraries, dependencies, and operating systems that are not guaranteed to work on other systems. It hinders accessibility and standardization.

It is also a big problem. Scientists create CRs to create their models and allow other scientists to configure their own. Thankfully, there is a method that can tackle this issue.

Containerization packages software to be compatible across different operating systems. It can allow users to use a CR without installing libraries or configuring dependencies. The user only needs to build a container and run it to produce a model. However, containerization is not a straightforward process. It is complicated and still does not provide the accessibility and standardization Dr. McKay requires.

Here is where our team steps in by creating the **Fossilized Controller**. This tool allows for the painless creation of standardized containers and provides an online guide that walks through containerization. The project consists of three major components: a command-line interface or **CLI**, a Container Manager, and adapter libraries. This document will detail the testing of the Fossilized Controller.

An explanation of the three main modules is needed to understand how effective the proposed testing is. The CLI is how the user interacts with the Fossilized Controller. Specifically, the CLI connects the user to the Container Manager and allows the user to manage containers. The Container Manager directs and keeps track of all containers and directly interacts with **Docker's API**. The adapter libraries are inside each container alongside a CR. They are the link between a CR and the Container Manager and communicate with the Container Manager via an HTTP connection.

According to IBM, software testing ensures a software end-product works as intended [3]. Software testing is needed because of how complicated software can become. Testing can bring to light areas of the software that are lacking such as poor functionality and input handling. How software testing works overall is by conducting a variety of testing strategies to verify that most workflows function correctly.

For this project, the plan is to use unit testing, integration testing, and usability testing.

The Fossilized Controller uses unit testing to test each critical, individual component that facilitates communication between modules. Unit testing is adequate and appropriate for this tool because it creates a strong foundation of parts in a complex architecture. Integration for the recent Alpha Demo was easier because individual components were tested thoroughly, so the time assigned for integration solely focused on integration.

Integration testing is integral because most of the Fossilized Controller's use cases require each module to work as a cohesive whole. Each module relies on the others to simplify containerization and can not stand alone.

Usability testing ensures that this tool adequately satisfies the accessibility requirement given by Dr. McKay. This type of testing obtains valuable information to make the Fossilized Controller

more understandable for its user base, scientists with CRs they wish to containerize. Because the user pool is highly specialized, it is small. The Fossilized Controller must reach a high standard of accessibility to break into and stay relevant in this population.

The remainder of this document will thoroughly discuss each testing strategy: what it is, how the testing strategy relates to the Fossilized Controller, and a detailed plan to employ the tests regarding the Fossilized Controller.

2. Unit Testing

2.1 What Is Unit Testing

The next section of this document goes into detail about a singular kind of testing used in software development. Below is a quick description of what unit testing is and why it is so important in the software development process.

To start off, unit testing is the process of testing the smallest part of the software possibly to ensure that the unit is working as intended. These tests aim to isolate small blocks of code that perform a very specific function to ensure that it performs correctly when not being used with other units.

There are multiple reasons why unit testing is so important for the software development process. The first is to ensure that the methods and practices researched before implementation will work as the developer intended. The second is to ensure that individual components will behave correctly during integration. This allows developers to focus on integrating the units together and not fixing broken units. The overall goal of unit testing is to test individual code blocks, or units, by themselves.

2.2 Unit Testing Plans in Detail

Next is what will and will not be unit tested for the Fossilized Controller. This section will be divided based on the three modules of the Fossilized Controller: the command-line interface, the Container Manager, and the adapter libraries. To run the unit tests on these modules a library called `unittest` will be used, allowing for the tests to be run and not break on erroneous outputs.

The first module to examine is the CLI. Before going into the commands that will be tested, it is important to cover those that will not be unit tested and why. Most of the commands inside of the CLI simply call other files and scripts that perform specific actions. For example, most commands use Python's `docker` library and the Container Manager's *get_container* method. Because of this, only two of the CLI's commands are being unit tested. The two commands that

will be unit tested are the *create* function, which allows users to create a **Dockerfile** for their model, and the *clean* function, which allows users to clear up the space being used by running or cached containers.

The second module to expand on is the Container Manager. This module holds the most important unit test for the Fossilized Controller tool: the *get_container* function. This unit is the most important because it is how the CLI sets and gets information on containers. Since only one part of the Container Manager is being unit tested, it is important to discuss why the other parts were not unit tested. The main reason is the *get_container* function is the only unit in this module that performs a vital action. The rest of the module consists of helper functions, e.g. creating and assigning values to variables.

The third and final module is the adapter libraries. The units to be tested are: starting the HTTP server and handling the POST requests the HTTP server receives from the CLI. The reason only two functions are being tested is because they are the majority of what the library will be doing. The rest of the functions inside are simply getters, setters, and helper functions that help the HTTP server send and receive appropriate files.

2.3 Examining Each Code Unit

Now that the three different sections of unit testing are covered, a deeper dive into what these unit tests will look like is necessary.

The first unit test is the *create* function in the command-line interface. The test for this command will use a function that is a part of the command-line interface that the package is already using. Using the library, the unit test will call the *create* command with an input for the prompt asking for a command to run the main file, the input being used to insert into the file is “python unitTest.py”. Then, to test that the Dockerfile was created correctly, the file contents are saved to a string and compared to the expected file contents. For this unit test there are no erroneous inputs since the function is just writing a string to a file and creating the file if it doesn’t exist already.

The second unit test is for the second part of the CLI, the *clean* function. This test will ensure that the cache of Docker containers is properly deleted and clears up space on the users machine. In order to do this, the unit test library is used to call the command and retrieve the

result from it. Once the result is received, the test checks that there are no containers currently cached using Docker's library. If there are no containers then the test will return true. For this unit test there is no input and therefore no erroneous inputs.

The third usit test is for the Container Manager's *get_container* functionality. In order to ensure the unit is working correctly, the function should create a container information object and cache it so that it can be accessed again. To check this, the function is called five times with set names to be tested later. Once the objects are created, the controller is deleted and the cache file is checked to make sure that the objects were cached correctly. The names of the container information objects are "unitTest - get_container:" and then one through five. There are no erroneous inputs for this unit test since the input is just the name of the image.

The fourth unit test is for the adapter library. More specifically for the *start_server* function, which is responsible for opening a server that can receive files. In order to check this, the function will be called to start the server and then will create a client that sends a get request to the server. Once the client gets a response from the server, the test checks that the status code returned was equal to the open literal. Since this unit test is starting a server there are no potential erroneous inputs or boundaries to check, just that the server is up and running.

The fifth and final unit test is for the *handle_post* function inside of the adapter library. The test that needs to be performed will check that the server received and returned a value. To execute this function, another client is created that sends an HTTP POST request to the server and ensures that the function returns an empty zip file. This is because the test only checks that the server can properly receive the files and return them over the HTTP connection. For this test there are no erroneous inputs even if no files are sent. There are also several boundaries that will be tested regarding how long the client will wait for a response from the server. The low boundary will wait a second before sending a response and the high boundary will wait roughly ten minutes for the server to run. Now that unit tests have been extensively described, it is time to go into integration testing.

3. Integration Testing

3.1 What Is Integration Testing

While unit testing is the process of testing individual pieces of code that make up the components, integration testing is the process of testing the functionality of components and the interactions between them. Instead of looking at individual functions, focus is put on the totality of the components as one product. Integration testing is important for testing end-to-end use cases, where unit testing is too fine-grained. Unit tests check individual aspects of code, and integration tests check the combined functionality. This section will go over the test harness for the controller and adapter libraries, and how these components are tested in the cloud.

3.2 How Modules Communicate Within the System

GitHub Actions is the Continuous Integration, or **CI**, service used for running these integration tests in the cloud. Workflows describe the actions taken when new code is created, one of which runs the tests. These tests are run in a Virtual Machine that hosts the latest version of Ubuntu. This Ubuntu server runs the controller and Docker containers like normal processes. The workflow looks very similar to a list of commands a user can run on their local machine.

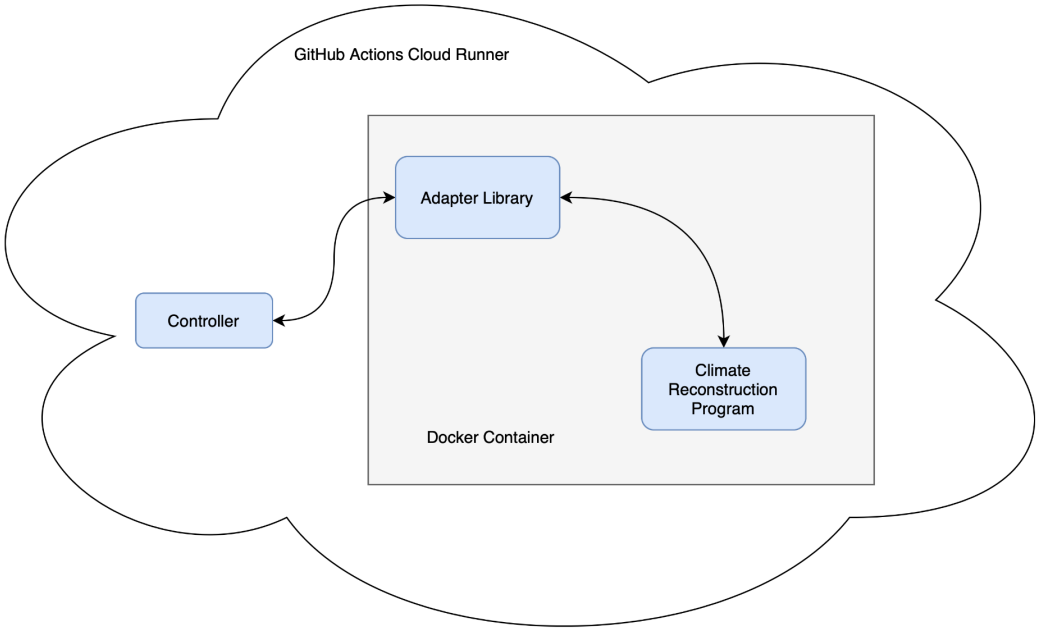

Figure 1: The components running in GitHub Actions in the "cloud", which is one of GitHub's servers. This cloud "runner" runs the described workflow to run the integration tests.

Every time a commit is pushed to the GitHub repository, the CI Workflow is triggered and the integration tests are run. The results of the workflow can be seen in GitHub's interface. The main components tested are the controller, the adapter libraries, and the climate model. The controller and Docker container communicate over HTTP, while an adapter library and the climate model are bundled into the Docker container. The CLI functionality is tested by running the commands in a shell on the CI server. The functionality of the contents of the Docker container is tested by running the container and invoking the model.

3.3 Examining Variations of the Integration Tests

The workflow sets up the environment for the test harness, which runs the controller with a test climate model. Various climate models can be used, and a test climate model is stored in the same repository so testing can be done by any holder of the source code. The workflow instructs the controller to build a Docker container from the source code of a climate model, using the CLI in the process. Finally, the Docker container is run, test data is sent in by the controller, and the results are checked. Checking the results ensures that the entire pipeline worked correctly, and the resulting climate reconstruction is what is expected. This process

makes up the basic test harness. This can be expanded to many different reconstruction programs that test different functionality of the controller and adapter libraries.

Variations of this integration test harness are used to test different configurations the software may be used in. To deeply test every component, the test harness is filled with either the Python or R adapter, along with a climate model. The climate model is run and results are checked for each variation.

Integration Test	Adapter Used	Climate Model
Variation #1	Python Adapter	Test model in Python
Variation #2	R Adapter	Test model in R or Python
Variation #3	Python Adapter	*Holocene*
Variation #4	R Adapter	*Temp12K*

Figure 2: Table of variations of the integration testing harness. Holocene *and* Temp12K *are private reconstruction projects worked on by Dr. Erb and Dr. McKay respectively. They will be used in the GitHub Actions workflow if they are made publicly available.*

Test data is used with all of the models, and is bundled with a copy of the output data to verify the results were correct. With all of these variations running with every commit made, regressions in the code can be checked immediately. Integration tests increase the stability of the code by testing the many variations of complete functionality of the components. Unit testing and integration testing and great for analyzing the correctness of code, but equally important is how users feel about the software. The next section discusses the process for communicating with users.

4. Usability Testing

4.1 What Is Usability Testing

The final form of testing utilized is usability testing. Usability testing focuses on the end users, unlike the other sections that make sure the products code and modules are tested properly. The goal of usability testing is to make sure that the product is something that end users can use to solve their problems with ease. This is done by gaining feedback from a group of end users concerning the usability of the tool. Users will test the tool and determine what works well and what does not. In this case, users will interact with the CLI to create and access PReSto containers.

4.2 Tailoring Usability Testing to the Fossilized Controller

Before usability testing can start, it is necessary to define the type of end users that will test the Fossilized Controller. The tool requires some prior knowledge from the user and they must fit certain criteria for usability testing. The Fossilized Controller is developed with a specific group of users, climate scientists, in mind and is not meant to be used by the general public.

The Fossilized Controller is specifically designed with climate models in mind. As such, the users should also be familiar with climate models because otherwise they have increased chances of failure when containerizing programs. The testing plan was created with this in mind. The controller is not for general containerization purposes. Alongside that, users need to have programming and command-line experience.

The following prerequisites are required for a user to participate:

1. Have knowledge of climate models
2. Have knowledge of either Python or R
3. Have experience with using command-line interfaces

After the above prerequisites have been met, the other main criteria for testers is that they have no knowledge of the Fossilized Controller. Accurate and detailed feedback is best provided from users who do not have prior experience using the tool. It creates a neutral environment and

allows for a fair evaluation. Parts that were obvious to the team but not to the final end users are identified.

4.3 Usability Testing Plans in Detail

For testing sessions, users are provided documentation on how to install and use the Fossilized Controller across multiple sections. The sessions are largely independent and users will have the team's contact information if any issues come up during testing. The sections involve the three core components of the tool: installing, creating, and accessing. As such, the three components are tested across four separate sections. The sections start simple and gradually grow more complex so that users have time to get familiar with the tool.

Users are chosen based on their previous experience with climate models. They are split into two separate categories for usability testing based on their academic level. The first category of users are scientists who have created their own climate models. They understand the models intimately which might create a smoother process for containerization. Examples are lab leaders or experienced scientists. The second category of users are scientists who are using models not created by themselves. An example could be students of a laboratory. By choosing participants based on the two categories, more insightful feedback is gained.

To gather the results, each section has an associated feedback table with their individual tasks and multiple prompts. Each section has individual tests to gather deeper insight on what parts of each process are difficult or easy. The first column explains the test and the expected result. The first prompt asks the user to rate the difficulty in completing the task on a scale of 1 to 5, with 1 being no issues and 5 being extremely difficult. The second prompt asks if they needed support from the team to complete the test. The third prompt is the additional feedback section that allows users to provide further comments about how their experience went. The three prompts will determine which parts of the product are difficult for end users and can improve them. Any tests with high difficulties and/or low success rates will provide insight on what modules need improvement.

It is noted that a test is highly dependent on the previous one being successful. If one test has not been successful, then the scientist can not move on to further tests. There are two routes the scientist can take if they find an unsuccessful test. The first is submitting the feedback table

as is so the team can determine where users struggle the most. The second is contacting the team for support on how to fix the issues they are coming across.

Example Section			
Tests	Difficulty	Needed Support?	Additional feedback
Description of the individual test. Success: Explanation of a successful test	1 2 3 4 5	Yes No	Any comments about the process can be included here.

Figure 3: Filled out example feedback table

4.3.1 Installing the Fossilized Controller

The first section that every user needs to complete is installing the Fossilized Controller. There are three tests in total in the section that focus on having a working installation of the controller.

Follow documentation to build the Fossilized Controller

Success: The user builds the controller to install

Install the Fossilized Controller

Success: The user used the pip package manager to install the controller

Confirmed Correct Installation

Success: The user will successfully run a command from the controller

4.3.2 Creating a Container by Example

Before a user containerizes their own model, they can go through a guided example that provides all necessary files to create a PReSto container. This is a public climate model already adapted to be PReSto ready. This is so users can have a working example of how to prepare their models

Building a container with provided files

Success: A Docker container ready to run

Running the container

Success: Returned output files

Cleaning up the container

Success: Completely deleted containers

4.3.3 Creating a Container by Instructions

The main difference between creating a container by example versus by instructions is that the user will adapt their models to be PReSto ready in this section. They are provided with instructions of how to modify their model and what files they need to create.

Building a container with created files

Success: A Docker container ready to run

Running the container

Success: Returned output files

Cleaning up the container

Success: Completely deleted containers

4.3.4 Accessing a PReSto Container

The other core feature of the controller is accessing containers that other scientists have created. Users will need to download and run a container created by the team and view the results.

Downloading a container

Success: A Docker container ready to run

Running the container

Success: Returned output files

Cleaning up the container

Success: Completely deleted containers

5. Conclusion

As a reminder, the overall goal of this project is to create a tool that helps paleoclimatologists containerize their code. This tool is known as the Fossilized Controller. It consists of three coding modules: the CLI, the Container Manager, and the adapter libraries.

The project uses three testing strategies: unit testing, integration testing, and usability testing.

During unit testing, the focus is on vital units, code units that communicate with other modules or make up the majority of the module. After identifying the code blocks that belong to this “vital unit” group, their expected and erroneous inputs were detailed.

Integration testing gives the ability to test the controller’s workflows. There are four integration tests. The first two test the controller with fabricated test models in Python and R. The last two test the controller with actual CRs in Python and R. Testing both fabricated and real-world CRs allows for the continued integration with known and unknown CRs.

Section Four tailors usability testing to the Fossilized Controller. There are two main test groups, testers who have created a CR and individuals who use CRs but have not built CRs. Usability testers will accomplish three tasks: installing the controller, creating a container by example, and by instructions. This process will give the team the direction needed to make the controller accessible.

We are excited to use these strategies, outlined and tailored in this document, to improve the Fossilized Controller. As of now, most of its functionality is complete and is ready for testing. After explaining and detailing these testing methods, implementing them will create a sturdier, robust product and launch it into a new development stage. In doing so, the controller is that much closer to helping scientists containerize and share their CRs.

Glossary

API or Application Programming Interface: A defined method for external applications to interact with an application

CLI or Command-Line Interface: A program that operates on the command-line.

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

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

CI or Continuous Integration: A service used for running integration tests.

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

Docker: A service that provides container services, e.g. creation and management. What the Fossilized Controller uses to manage containers.

Dockerfile: A file that instructs Docker on how to create and run a container.

Fossilized Controller: The tool to be tested. A program that can containerize climate reconstructions that is being developed by the Fossilized Containers team.

Paleoclimatology: The study of climates of the past.

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

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).

[3] "What is software testing and how does it work?" *IBM*. [Online]. Available: https://www.ibm.com/topics/software-testing. (Accessed: 1-Apr-2022).