Technological Feasibility Analysis

8 November 2019

Sponsors

Dr. Kiona Ogle

Dr. Michael Fell

Mentor

Isaac Shaffer

TreeViz

Riley McWilliams
Qi Han
Haitian Tang
Daniel Rustrum
Alex Bentley

Contents

Introduction **3**
Background 3
Problem 3
Solution 3
Document Purpose 3

Technological Challenges **4**
Client Requirements 4

Technology Analysis **5**
Design Decision 1: ReST Framework **5**
CherryPy 6
Falcon 7
Flask 7
Final Decision 8
Design Decision 2: Tree Visualization **9**
Unity 10
CreateJS 10
Threejs 11
Final Decision 12
Design Decision 3: Web Development Service **13**
Angular 14
React 15
Vue.js 16
Final Decision 17
Design Decision 4: Backend Database **18**
MongoDB 18
MySQL 19
Redis 19
Final Decision 20
Design Decision 5: Host Environment **21**
Local NAU Desktop 21
Heroku 22
Firebase 22
Final Decision 23

Technology Integration **24**

Conclusion **26**

References **27**

Introduction

Background

To better understand how certain factors affect trees, Dr. Kiona Ogle of Ogle Labs developed a simulation that shows a tree's growth over time. The simulation is called the Allometrically Constrained Growth and Carbon Allocation (**ACGCA**) model.

Problem

The problem is that the model is currently built in a way that severely limits who can use it. The main user demographic that Dr. Ogle built the model for is biologists and requires the user to know some programming because it runs using command line prompts. The model outputs raw data as text and does not visually represent the tree's life cycle. Dr. Ogle wants the model to be utilized by more than biologists and not just for lab research. For example, students in a classroom should be able to use it and learn how certain factors may determine how a tree grows.

Solution

Team TreeViz, consisting of the members: Riley McWilliams, Alex Bentley, Daniel Rustrum, Haitian Tang, Qi Han, and mentor: Isaac Shaffer, is tasked with making the model more user-friendly. This will be completed in two major parts: input and output. First, the input process will have to be more streamlined. We will create a web application with an easy-to-use graphical interface. Instead of entering input through a command line, users will enter all of the necessary data in text boxes and drop-down menus. Second, the output will be converted into a graphical representation of a tree. With our user-friendly web application and visually appealing representation of tree growth, Dr. Ogle's model can reach a much broader audience.

Document Purpose

The purpose of this document is to figure out which technologies we will be using to build our final product. We broke the project down into five design decisions, each needing a certain type of technology to solve its specific problem. Our goal for this document is to consolidate our research and organize it for comparison. Then we will decide the best technology according to important factors that satisfy client requirements.

Technological Challenges

In this section, we list out all of the technological needs that our final product requires. The purpose is to clearly show which problems need to be solved. We must make our decisions based on certain requirements that the Dr. Ogle has specified.

Client Requirements

Dr. Ogle has requested a user-friendly platform for tree visualization. She wants a web application that makes it easier for users to enter parameters to her simulation. Then, the results should be visualized as a graphical representation of a tree. She also wants a way to keep track of user information such as name and occupation. Finally, there needs to be an environment to host the web application.

To deliver the product that our client requests, we separated her requirements into five design decisions:

- **Design Decision 1: ReST Framework**
  The ReST API acts as a bridge from the user interface to the **ACGCA** model. The framework ensures that we implement the ReST API correctly and promptly. We need a modular framework so that we can isolate complex functional parts according to the requirements and processes established with Dr. Ogle.

- **Design Decision 2: Tree Visualization**
  A key component of this project is the visualization of trees based on the **ACGCA** model's output parameters. While the client does not specify nor prefer a specific-looking output, we must decide on a visualization tool. Variables like trunk radius, tree height, and leaf density must be taken into account when choosing a software.

- **Design Decision 3: Web Development Service**
  A framework or library for the website is needed to develop the web-facing application. It also must contain all of the necessary code and dependencies for the visualization software to run on the internet. The type of service is not specified by the client but needs to run quickly and efficiently to create a suitable application.

- **Design Decision 4: Backend Database**
  A database is needed to store information about the users, including name, affiliation, location, and purpose. It also needs to contain the necessary information to determine whether the user is new or not. To satisfy Dr. Ogle's requirement for storing user information, we must find a suitable database.

- **Design Decision 5: Host Environment**
  Dr. Ogle envisions this application to be easily accessible by hosting it on the internet. Finding a stable and fast host environment is necessary to reach a broader audience.

Technology Analysis

In this section, we look at each of the five design decisions and showcase a handful of possible solutions that address each problem. Careful consideration is put into each option, and research and analysis are done in order to pick the best fit for our final product.

Design Decision 1: ReST Framework

We need to create a Representational State Transfer (ReST) application program interface (API) that is able to act as an interface. It should allow and support our business logic to work independently and mimic the structure of our architectural diagram, as shown in Figure 2.

Using this structure we can send/receive data to and from instances of the model without needing to orchestrate them, worry about how they are being run, or where they are being run from. The model simply needs to send a request to the ReST API whenever it can process another input or when it needs to send the data that it has processed.

Performance isn't a factor that is important to consider for a couple of reasons, our expected load on the ReST API is expected to be around 30-50 people at once which can be handled by any standard web framework. As the API scales and starts to receive more users than expected bottlenecks will appear, but with load testing these problems can be addressed when they start to happen[1]. It doesn't matter what framework we choose because they are guaranteed to happen anyway.

To guide our research of ReST APIs, we focused on a handful of important factors:

I. **Supported**

Software being supported is whether or not the latest release is within the last three months and has had at least two frequent releases before that. It is important to use a framework that will continue to receive updates. Otherwise, it might become depreciated. This will be measured with either a yes (it has recent releases) or no (it does not have recent releases).

II. **Modular**

Modularity is determined by two things: whether or not the framework interferes with testing and debugging without using their provided method, and whether or not you are getting any optional features that aren't required for a ReST API. Both of these criteria need to be met for it to be considered modular. This is important because it allows us to ensure that the application we are creating has a minimal amount of bugs. This will be measured with either a yes (it is modular) or no (it is not modular).

III. Hard-Coded

A software option being hard-coded is determined by whether or not the framework requires you to hardcode data without giving a viable alternative. Hard-coded variables can provide problems when extracting information into a configuration file/object. This will be measured with either a yes (it does require hard-coding) or no (it does not require hard-coding).

IV. Has Dependencies

Dependency is determined by whether or not the framework requires an external resource in order to use it. Fewer dependencies mean there is less for our client to maintain. We will measure this with either a yes (it has dependencies) or no (it does not have dependencies).

To determine the best technology to use for our project, we researched three possible software for a ReST API. To test each option according to the factors, we made a simple demo. We also referenced pypi[2] for details about the framework and the documentation that the framework provides.

1) CherryPy

CherryPy is a barebones web framework written in python. **CherryPy's** philosophy is that everything that is optional is turned off by default and that extending the framework should be as easy as writing a function.[3]

Here is how **CherryPy** holds up against the factors:

I. Supported

Yes, **CherryPy** is supported. It has frequent releases, the most recent being released on November 3rd, and two more before that on October 2nd and September 3rd.

II. Modular

Yes, **CherryPy's** endpoints are made up of plain Python objects which means that they can be tested individually apart from the framework. **CherryPy** also has isolated objects that either contain functionality or information. The information is freely available to use and the functionality can be overridden or inherited for a customized experience.

III. Hard-Coded

No, **CherryPy** is not hard-coded. While it does use decorators, an architectural pattern that Python has implemented allowing you to wrap functionality with functionality, it does the wrapping during load time and not during runtime. **CherryPy** provides an alternative way of using them.

IV. Has Dependencies

No, **CherryPy** has no dependencies. This means it is fully capable of handling HTTP requests and sending HTTP responses on its own. It can also handle opening ports up to the internet and running a web server.

2) Falcon

Falcon, written in Python, is a ReST specific framework, meaning it is not able to deliver files, or stream data, etc. However, due to its limitation, it constrains you to only having a ReST API.[4]

Here is how **Falcon** holds up against the factors:

I. Supported

No, **Falcon** is not supported. The last three releases that **Falcon** has had were on April 29th, 2019, January 16th, 2018, and September 6th, 2017.

II. Modular

Yes, **Falcon** is modular. Unlike **CherryPy**, it is able to achieve this by having the user pass the function they want to be called to the framework. It provides a separation between the framework and its implementation, allowing for the functionality to be tested individually while also allowing the user to “hook” into processes that may otherwise be hidden. **Falcon** also uses generic Python objects to create and define the endpoint from a given path.

III. Hard-Coded

Yes, **Falcon** is hard-coded. Due to how the hooks in **Falcon** are implemented, the parameters that need to be passed to the function must be hard-coded. This is because data is passed to the function during load time instead of during run time when variables can be used.

IV. Has Dependencies

Yes, **Falcon** does have dependencies. It relies on an external package to start a web server. However, it can handle HTTP requests and HTTP responses on its own.

3) Flask

Flask is the least lightweight framework out of the three options. It is also one of the most popular HTTP frameworks to use. Due to that reason, it has evolved into something that is able to handle multiple use cases, one of those being a ReST framework.[5]

Here is how **Flask** holds up against the factors:

I. **Supported**

No, **Flask** is not supported. The latest was on July 8th, 2019, more than 3 months ago. The two releases before that were on July 4th, 2019 and May 17th, 2019.

II. **Modular**

No, **Flask** is not Modular. It wraps the framework on its own using decorators. This means that we would need to either pull away that abstraction ourselves or use the testing functionality that **Flask** provides.

III. **Hard-Coded**

Yes, **Flask** is hard-coded. It requires the paths and methods to be hard-coded into the script. This means we will not be able to manipulate that information with variables or be able to extract that information with configurations.

IV. **Has Dependencies**

No, **Flask** does not have any dependencies. Like **CherryPy**, it is able to handle the HTTP requests and HTTPS responses required for the ReST API. It can also handle exposing the API to the internet by managing the ports and running the webserver.

Final Decision

Table 1 compares the three options based on our analysis according to the factors.

	CherryPy	**Falcon**	**Flask**
Supported	Yes	No	No
Modular	Yes	Yes	No
Hard-Coded	No	Yes	Yes
Has Dependencies	No	Yes	No

Table 1: DD1 Results

We chose **CherryPy** because, overall, it fits best for what we are looking for. It is supported, modular, not hard-coded, and does not require any dependencies to run. Its modularity and maintainability will help move towards our goal for a ReST Framework.

Design Decision 2: Tree Visualization

We need a way to create graphical trees based on the input that a user enters. These trees should be dynamically created from the output of the **ACGCA** simulation and be similar to the real-life tree they represent.

To guide our research in tree visualization, we focused on a handful of important factors:

I. **File Size**
The file size is the amount of storage that will be taken up on the server. We will have a limited amount of server storage space; therefore, we do not want the build to take up too much of it. This will be measured in kilobytes.

II. **Boot-Time**
The boot time is the amount of time from when the webpage is accessed to when the user can interact with it. We do not want users to have to wait long periods of time to use the application. This will be measured in seconds.

III. **Performance**
Performance is the amount of activity the webpage has. This is important because a lower-end computer may not be able to handle the amount of activity that the software requires. To measure this, we will run each demo for 60,000 milliseconds (ms) and record how much of that time was spent performing actions.

IV. **Familiarity**
Familiarity is how many team members have some experience with the software. The team can work more effectively if there is a higher familiarity. This will be measured in the number of members who are familiar with the software out of five.

V. **Built-in Compatibility with the ACGCA Model**
Built-in compatibility is whether or not the software has built-in tree creation functionality. If not, then we would have to create the trees with our own algorithm. We will measure this with either a yes (it has compatibility) or no (it does not have compatibility).

To determine the best technology to use for our project, we tested three possible software with tree visualization capabilities:

1) Unity

Unity is a popular game development engine with a built-in 3-dimensional tree creator. It uses a rendered scene to place objects and UI elements, and C# scripts for logic.[6]

To test **Unity's** capabilities in tree visualization, we created a demo using its built-in tree maker. The demo consists of a text file is that is created based on user input. Then a tree is formed based on the data in the text file. It was very easy to get **Unity** to run on a web browser. There are settings to build the project in WebGL, and strip unnecessary imports such as the physics engine, allowing for smaller builds. It took approximately two hours to make this demo.
Demo: https://dana.ucc.nau.edu/~rcm264/TreeViz/Unity-demo/index.html

Here is how **Unity** holds up against the factors:

I. File Size

Unity has a build size of 7,400 kilobytes.

II. Boot-Time

It takes about 2-4 seconds for the webpage to become intractable by the user. This is due to **Unity** having a splash screen that lasts 2 seconds.

III. Performance

When ran for 60,000ms, the **Unity** demo had 34,545ms of activity.

IV. Familiarity

3 members of the group are familiar with **Unity**.

V. Built-in Compatibility with the ACGCA Model

Yes, **Unity** has built-in functions that can be directly related to the output of the **ACGCA** model. This includes a built-in tree creator with variables such as trunk and branch radius.

2) CreateJS

CreateJS is a suite of JavaScript plug-ins that make creating a simple web application easy. One of the plug-ins, EaselJS, has a canvas where objects are drawn and rendered.[7]

To test **CreateJS's** capabilities in tree visualization, we created a simple demo. The demo displays a very simple 2D tree made of a rectangle for the trunk and a triangle for the leaves. The basic shapes were easy to make and it was easy to place them where we wanted. It took approximately an hour to make this demo.

Demo: https://dana.ucc.nau.edu/~rcm264/TreeViz/CreateJS-demo/index.html

Here is how **CreateJS** holds up against the factors:

I. **File Size**

CreateJS has a build size of 8,100 kilobytes when only using EaselJS. If we want to use the other three libraries, it will total 34,000 kilobytes.

II. **Boot-Time**

It takes less than a second for the webpage to become intractable by the user when running **CreateJS**.

III. **Performance**

When ran for 60,000ms, the **CreateJS** demo had 108ms of activity.

IV. **Familiarity**

2 members of the group are familiar with **CreateJS**.

V. **Built-in Compatibility with the ACGCA Model**

No, **CreateJS** does not have built-in functions that can be directly related to any outputs of the **ACGCA** model.

3) Threejs

ThreeJS is a JavaScript plug-in that is similar to CreateJS. However, it focuses more on three-dimensional rendering. It also uses a canvas to draw and render objects.[8]

To test **ThreeJS'** capabilities in tree visualization, we created a simple demo. The demo displays a very simple 3D tree made of a cylinder for the trunk and a cone for the leaves. The syntax for creating these shapes was somewhat difficult to figure out at first, but we eventually got it working. It took approximately an hour to make this demo.

Demo: https://dana.ucc.nau.edu/~rcm264/TreeViz/ThreeJS-demo/index.html
Here is how **ThreeJS** holds up against the factors:

I. **File Size**

ThreeJS has a build size of 1,100 kilobytes.

II. **Boot-Time**

It takes less than a second for the webpage to become intractable by the user when running **ThreeJS**.

III. **Performance**

When ran for 60,000ms, the **ThreeJS** demo had 10,747ms of activity.

IV. **Familiarity**

1 member of the group is familiar with **ThreeJS**.

V. Built-in Compatibility with the ACGCA Model

No, **ThreeJS** does not have built-in functions that can be directly related to an output of the **ACGCA** model.

Final Decision

Table 2 compares the three options based on our analysis according to the factors.

	Unity	CreateJS	ThreeJS
File Size	7,430 kb	33,956 kb (whole suite) 8,114 kb (EaselJS only)	1,188 kb
Boot-Time	~ 3 s	< 1 s	< 1 s
Performance	34,545 ms	108 ms	10,747 ms
Familiarity	3 members	2 members	1 member
Built-in Compatibility	yes	no	no

Table 2: DD2 Results

We chose **Unity** as our technology for tree visualization. Our choice is based on the fact that it has more built-in compatibility with the **ACGCA** model than the other options. More group members are also familiar with the software.

We will test **Unity** further by creating trees based on an increasing amount of the **ACGCA** model's outputs. This will allow us to incrementally build-up to a tree that is fully representative of the model's output.

Design Decision 3: Web Development Service

A large part of this project is the interaction between the user and our service. A web framework or library needs to exist within the program to be able to mediate the information between the client-side and the server/algorithmic side.

To guide our research in front-end web development, we focused on a handful of important factors:

I. **Time to Load**

The time to load is the initial script bootup time (see Figure 1) it takes for the webpage to load. Speed is a crucial part of the internet, and to keep a user's attention we will need a service that takes the least amount of time to load the webpage. This will be measured in milliseconds (ms).

II. **Familiarity**

Familiarity is how many team members have some experience with the software. The team can work more effectively if there is a higher familiarity. This will be measured in the number of members who are familiar with the software out of five.

III. **Type**

Type is whether or not the technology chosen is a library or a framework. We will need to know if the extra complexity of a library is necessary for our product. The factor is measured on whether the type of service is a 'library' or a 'framework'.

IV. **File Size**

File size is the size of the framework and all of its dependencies. The size is an important factor to consider because it will determine how much data is being transferred over a network. This will be measured in bytes.

Figure 1 is a graphic of the results from a js-framework-benchmark developed by Stefan Krause and was used to determine the results of some factors.

Name	vanillajs-non-keyed	svelte-v1.41.2-non-keyed	inferno-v3.10.1-non-keyed	angular-v5.0.0-non-keyed	react-v16.1.0-non-keyed	vue-v2.5.3-non-keyed
consistently interactive a pessimistic TTI - when the CPU and network are both definitely very idle. (no more CPU tasks over 50ms)	40.2 ± 10.4 (1.0)	38.9 ± 1.1 (1.0)	45.1 ± 2.9 (1.2)	97.3 ± 1.7 (2.5)	58.3 ± 1.1 (1.5)	54.9 ± 1.6 (1.4)
script bootup time the total ms required to parse/compile/evaluate all the page's scripts	3.8 ± 0.1 (1.0)	4.6 ± 0.2 (1.0)	9.8 ± 0.4 (1.0)	46.1 ± 0.9 (2.9)	22.0 ± 0.6 (1.4)	19.6 ± 0.9 (1.2)
main thread work cost total amount of time spent doing work on the main thread. includes style/layout/etc.	157.9 ± 3.0 (1.0)	160.4 ± 2.9 (1.0)	165.4 ± 2.1 (1.0)	214.0 ± 1.4 (1.4)	177.1 ± 1.8 (1.1)	175.0 ± 2.2 (1.1)
total byte weight network transfer cost (post-compression) of all the resources loaded into the page.	164,019.0 ± 0.0 (1.0)	163,741.0 ± 0.0 (1.0)	191,779.0 ± 0.0 (1.2)	304,134.0 ± 0.0 (1.9)	263,087.0 ± 0.0 (1.6)	221,610.0 ± 0.0 (1.4)

Figure 1: js-framework-benchmark [13]

To determine the best technology to use for our project, we tested three possible software for front-end web development:

1) Angular

Angular is a popular web development framework maintained by Google for designing dynamic web applications and is the oldest of the three services that were tested.[9]

A simple server was created using **Angular** to test the difficulty of the setup process. Setting up **Angular** is an easy thing to do, with downloading the service and creating an environment taking about 15 minutes. **Angular** handles all of the necessary dependencies and routing when creating a new project. Creating an **Angular** environment makes routing easy and webpages easier to work with because pages update every time the HTML file is saved. The amount of files that are installed in a basic setup of **Angular** is large and can be difficult to navigate, which can add unnecessary complexity to our program.

Here's how **Angular** holds up against the factors:

I. Time to Load

The time it takes to boot the initial scripts for **Angular** is the slowest based on the results from Figure 1, **Angular** takes about 45.9 ± 1.5 ms.

II. Familiarity

3 members of the group are familiar with **Angular.**

III. Type

Angular is a framework.[12]

IV. File Size

The file size of the **Angular** packet to be transmitted over a webpage is around 304,139 bytes according to Figure 1, which is relatively large compared to the other services available.

2) React

React is a library for building user interfaces and was designed and currently maintained by Facebook for updating web pages rapidly when changing data objects within a webpage.[10]

A simple server was created using **React** to test the difficulty of the setup process. While testing, **React** proved to be very simple to set up as well as modify data. The whole process took approximately 15 minutes. The environment was very similar to setting up the environments of **Vue.js** and **Angular**. The difficulty of **React** was that because it was a library it was much harder to navigate files as well as change the environment. It needed more hard-coding because it needed to be told exactly what to do.

Here's how **React** holds up against the factors:

I. Time to Load

React has a much faster script bootup time than **Angular** according to Figure 1. **React** has an initial boot time of 22.0 ± 0.6 ms, making this a very fast library for websites.

II. Familiarity

2 members of the group are familiar with **React.**

III. Type

React is a JavaScript library.[12]

IV. File Size

The file size of the **React** packet to be transferred over a network is approximately 263,076 bytes according to Figure 1.

3) Vue.js

Vue.js is an open-source JavaScript framework that was developed by Evan You, who was an old **Angular** team member. The goal of **Vue.js** was to create a new framework that combined the best approaches to front-end web development.[11]

A simple server was created using **Vue.js** to test the difficulty of the setup process. **Vue.js** was slightly harder to set up compared to the other two services. It required more information for the initial setup, such as determining a unit testing program, the stylesheet formats, etc. The amount of options gives us more control over the initial set-up for **Vue.js** compared to **Angular** and **React**. The environment is still easy to use and was able to be set up in approximately 15 minutes.

Here's how **Vue.js** holds up against the factors:

I. Time to Load

Based on figure 1, **Vue.js** is faster to load than the previous two options with an initial boot time of 19.6 ± 0.9 ms.

II. Familiarity

2 members of the group are familiar with **Vue.js.**

III. Type

Vue.js is a JavaScript Framework.[12]

IV. File Size

The file size of the **Vue.js** packet is approximately 221,610 bytes according to Figure 1, making **Vue.js** the smallest transfer size.

Final Decision

Table 3 compares the three options based on our analysis according to the factors.

	Angular	React	Vue.js
Time to Load	45.9 ± 1.6 (ms)	22.1 ± 0.6 (ms)	20.8 ± 0.7 (ms)
Familiarity	3 members	2 members	2 members
Type	Framework	Library	Framework
File Size	304,134 bytes	263,076 bytes	221,610 bytes

Table 3: DD3 Results

Vue.js is our choice because it does not require many features for us to be able to set up the **ACGCA** model. The file size and time to load benchmark results prove **Vue.js** to be the fastest out of all the options and the most lightweight. Although more team members are familiar with **Angular,** TypeScript and JavaScript are languages the team knows and will provide a small barrier of entry when working with **Vue.js.**

Design Decision 4: Backend Database

We need a backend database that allows us to store user data. The database needs to be accessible by administrators so they can view the data. Dr. Ogle does not want a large variety of user data, so we only need a simple database.

To guide our research on databases, we focused on a handful of important factors:

I. **Time to Set Up**

The time to set up is the speed of creating the database. This is important the database may crash, and we need to be able to rebuild it quickly. This will be measured by the amount of time spent making a demo from the beginning to the end, including the time spent downloading the software.

II. **Familiarity**

Familiarity is how many team members have some experience with the software. The team can work more effectively if there is a higher familiarity. This will be measured in the number of members who are familiar with the software out of five.

III. **Type**

The type of database is whether the database is dynamic or static. This matters because it could determine whether or not we need more operations to achieve the same purpose. If the database is dynamic, it means that the structure is flexible and that we can easily make changes to it. A static database needs more operations to make the same changes.

To determine the best technology to use for our project, we tested three possible software for a backend database:

1) MongoDB

MongoDB is a document database that is based on the web. This allows users to more efficiently access and build the database. Because **MongoDB** is NoSQL, when we use it, we do not need to consider complex tables and relationships between them.[14][15]

To test whether **MongoDB** is the best choice for us, we built a demo for storing data from the web.

Here's how **MongoDB** holds up against the factors:

I. **Time to Set Up**

It took about 2 minutes to build the demo. **MongoDB** does not require a download.

II. **Familiarity**

3 members of the group are familiar with **MongoDB.**

III. Type

MongoDB is the dynamic database that allows the users to flexibly add or delete the document contains data.

2) MySQL

MySQL is a relational database management system based on SQL – Structured Query Language. It focuses on the relationship between tables and when you make any changes, you need to also change all the tables related.[14]

Here is how **MySQL** holds up against the factors:

I. Time to Set Up

It took about 5 minutes to build the demo for **MySQL** including download cost. It is simple to create a MySQL database with the MySQL language especially when the size of data is not very large.

II. Familiarity

4 members of the group are familiar with **MySQL**.

III. Type

MySQL is a static database because the information stored in it is structured as tables connected with others.

3) Redis

Redis is a key-value database and supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperlogs, geospatial indexes with radius and streams.[14]

Here is how **Redis** holds up against the factors:

I. Time to Set Up

It took 2 minutes to download it and 10 minutes to build a demo. The total cost time is 12 minutes.

II. Familiarity

1 member of the group is familiar with **Redis**.

III. Type

Reids is a dynamic database and it is a NoSQL language.

Final Decision

Table 4 compares the three options based on our analysis according to the factors.

	MongoDB	**Mysql**	**Redis**
Time to Set Up	2 minutes	3 minutes	12 minutes
Familiarity	3 members	4 members	1 member
Type	Dynamic	Static	Dynamic

Table 4: DD4 Results

We chose **MySQL** because there isn't much difference between these three candidates. We preferred **MongoDB** for its access to the web, but Dr. Ogle doesn’t need more than 5 variables to be stored in the database. In this case, **MySQL** is the best choice. It is easy to use and won't have any complex relationships.

Design Decision 5: Host Environment

Although we can do the development on our computers, we can not run our personal computers all the time to use them as the server of our project. So, we need to find a stable host environment that can run our program and communicate with the end user's browser.

To guide our research on host environments, we focused on a handful of important factors:

I. Storage

Storage is the amount of space within the environment to store the user information and our project. This is important because it helps us ensure our web application is running with no exceptions and receives the user data at the same time. Dr. Ogle wants us to control the budget to be as low as possible, we will measure this with the storage space based on the lowest plan tier of each service.

II. Price

Price is the amount of money that each web service will cost per month. This is important because our budget is limited and Dr. Ogle wants us to keep the budget as low as possible. We will measure this with the price per gigabyte of storage.

III. Reliability

Reliability is whether or not the chosen server will crash frequently. This is important because we do not want our web application to crash while being used. Reliability will be measured based on the number of recorded incidents in the last 60 days.

To determine the best technology to use for our project, we researched three possibilities for a hosting environment.

1) Local NAU Desktop

Local NAU Desktop is a computer located in the School of Informatics, Computing & Cyber Systems building. Dr. Ogle will provide a basic machine for us to run as a server, which we can set up any way we like. It is a basic choice for a server that can run all the time and provide services to our end users.

Here is how **Local NAU Desktop** holds up against the factors:

I. Storage

The storage of the local desktop is fixed and it can not be upgraded. Because Dr. Ogle is still preparing the local desktop, the storage is unknown for us.

II. Price

It is free if we choose to use **Local NAU Desktop**, but it can not be upgraded if we want more storage or faster process.

III. Reliability

As Dr. Ogle is still preparing the **Local NAU Desktop**, we don't know if it is reliable or not right now.

2) Heroku

Heroku is a cloud platform supporting several programming languages including Java, Node.js, Scala, Clojure, Python, PHP, and Go. It is a polyglot platform as it has features for us to build, run and similarly scale applications across most languages. It doesn't offer web storage hosting. It only provides the service to run our web program.[16]

Here is how the **Heroku** server holds up against the factors:

I. Storage

Heroku doesn't offer hosting storage services. We need to use Github to store our project and use **Heroku** to run our project. The lowest tier for GitHub storage is 1GB which is also the free tier.

II. Price

The price of using **Heroku** has two parts. One part is the price for **Heroku**, the other part might come from GitHub storage if we use more than 1 GB storage in the future. **Heroku** does offer the free tier but the server will sleep after 30 minutes of inactivity. The higher tier which needs $7/month for **Heroku** offers 10 process types instead of 2 process types for the free tier. The higher tier also keeps the system always active.[17] Table 5 shows the price options for **GitHub**.

	Amount
Basic Storage Data	1 GB
Extra Storage Data	50 GB for $5/month

Table 5: Github Price[18]

III. Reliability

Heroku reports that the service has had about 12 recorded incidents in the last 60 days.[19]

3) Firebase

Firebase is a mobile and web app development platform that provides a variety of developing tools and web server service which can provide us a stable online server. [21] Here is how **Firebase** web server holds up against the factors:

I. Storage

The storage space for the lowest tier which is also the free tier is 1GB.

II. **Price**

Firebase web server offers a free level when the amount of the transferred data is under 10GB per month and the amount of storage data is under 1GB per month. Our project might mainly transfer some statistical data for input and some picture data for the output, so it might be enough for our project. For upgrading the amount of storage, it only needs 0.026 dollars per GB per month.[22] Table 6 shows the price options for **Firebase**.

	Free Plan	**Pay As You Go**
Storage Data	1 GB	$0.026/GB
Transferred Data	10GB /month	$0.15/GB

Table 6: Firebase Price

III. **Reliability**

Firebase has reported having had 4 recorded incidents in the last 60 days.[20]

Final Decision

Table 7 compares the three options based on our analysis according to the factors.

	Local NAU Desktop	**Heroku**	**Firebase**
Price for the Lowest Tier	Free	Free for the lowest tier	Free for the lowest tier
Basic Storage	Unknown	1 GB	1 GB
Price for Upgrading Extra 1GB Storage	Can not upgrade	5 dollars/month	0.026 dollars/month
Accessibility to the User Data	Anytime	Need to request and wait	Need to request and wait
Reliability	Unknown	12 incidents	4 incidents

Table 7: DD5 Results

We chose a **Local NAU Desktop** because we want the budget to be as low as possible in the early stages of development. Its storage will not be a big problem as our project might not have a high requirement for it. In the future, as we need a more secure, stable, and accessible server, we might consider using other options.

We will test this option further by operating our project directly on the **Local NAU Desktop**.

Technology Integration

The challenge we face is separated into three parts: whether the algorithm can be run quickly or in parallel, how fast we can get users to and from the algorithm process while also validating and gating undesirable inputs, and whether we are able to gather the algorithm inputs and display the outputs.

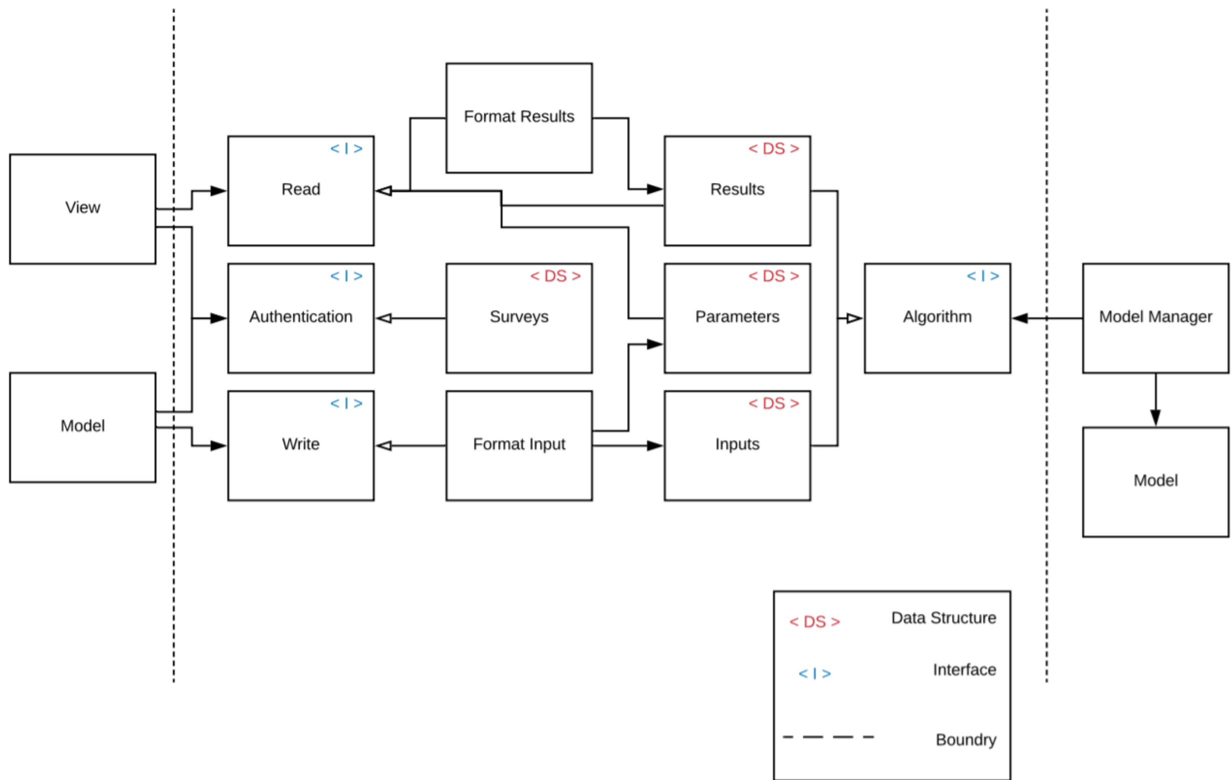

Figure 2: Architecture Diagram

The first solution for the algorithm problem is to cut off the algorithm from running too long and to allow each instance of the wrapped algorithm to contribute to converting users and running their inputs separately. This allows the algorithm to run in parallel allowing multiple instances to chip away at the input queue. The manager that the algorithm is wrapped in manages the inputs and outputs of the algorithm, manages the runtime of the algorithm, and sends and receives data from the user data structures.

The second problem is getting the data provided by users, filtering undesirable inputs, then putting the inputs into a queue for the algorithm to request, then sending the results back to the appropriate user. The solution is an architected process that can handle the user throughput and be able to queue the process in order for the algorithm to request another user.

The third problem, to gather user inputs and display the algorithm results, is solved with the Model-View-Controller design pattern where the controller is the process that the second problem's solution details. This way, the solution doesn't depend on what the user interface looks like, allowing that freedom in design. The user data is sent to the algorithm through the model and the results are retrieved through the view. **Vue.js** will handle the user interface and input. Then **Unity**, which will be built straight into the HTML, will display the output as a 3-dimensional tree.

The Interfaces that are denoted as <I> in blue in Figure 2 are where the Rest API will sit, meaning that operations that happen in the model and the view are completely independent of each other, and never directly interact.

Conclusion

The overall skeleton of the project will be made up of the Pythonic **CherryPy** web framework as well as the front-end web framework, **Vue.js**, for handling the JavaScript and TypeScript of the actual webpage. The **Unity** engine for 3D modeling will be used to create the actual visualization of the tree growth, which will be embedded in **Vue.js**. **MySQL** will be used to store user information in order to track a user's interaction with our web application. We will host this application on a **local NAU desktop** provided by Dr. Ogle and her team.

Table 8 shows our chosen option for each design decision, as well as our confidence in making the decision.

	Chosen Technology	**Confidence Level**
ReST Framework	CherryPy	High
Tree Visualization	Unity	High
Web Development Service	Vue.js	High
Backend Database	MySQL	High
Host Environment	Local NAU Desktop	High

Table 8: Chosen Technology Results

Team TreeViz will be working with Dr. Kiona Ogle to create a user-friendly platform for visualization tree growth. This will take the form of a web application that runs her simulation to create a visual model of a tree's lifespan. Through the research and analysis that the team completed in this document, we were able to choose the technology that fit best for each design decision. We weighed the options and can say with high confidence that we made all the right choices.

References

Design Decision 1: Rest Framework

1. https://medium.com/rpdstartup/rest-api-performance-tuning-getting-started-7a6efe-fa9e20
2. https://pypi.org/
3. https://cherrypy.org/
4. https://falconframework.org/
5. https://palletsprojects.com/p/flask/

Design Decision 2: Tree Visualization

6. https://unity.com/
7. https://createjs.com/
8. https://threejs.org/

Design Decision 3: Web Framework

9. https://www.sitepoint.com/angular-introduction/
10. https://reactjs.org/
11. https://vuejs.org/v2/guide/
12. https://hackernoon.com/angular-vs-react-vs-vue-which-is-the-best-choice-for-2019-16ce0deb3847
13. https://jsreport.io

Design Decision 4: Backend Database

14. https://blog.csdn.net/CatStarXcode/article/details/79513425
15. https://www.mongodb.com

Design Decision 5:Host Environment

16. https://www.heroku.com/about
17. https://www.heroku.com/pricing
18. https://github.community/t5/How-to-use-Git-and-GitHub/What-are-the-pricing-plans-for-Git-LFS-Is-there-something-better/td-p/2622
19. https://status.heroku.com/
20. https://status.firebase.google.com/summary
21. https://firebase.google.com/products
22. https://firebase.google.com/pricing /