**Design Portfolio 3**

A State Machine Viewer Using Web Technology

Ed F., Julien F., Shizhao W., Devon E., Maurice M., Michael C., and Leslie O.

## Introduction

## **Abstract**

We are presenting a containerized web application that displays a finite state machine(FSM) for run-time execution testing and graphical status monitoring. The system will use Docker so that it can be installed and run on a wide variety of platforms. State machines represent an event-driven system. Those events and the possible guard settings determine FSM behavior. The goal is to expand upon a C++ state machine autocoder that has text-only display and control capabilities. The interactive graphic interface of our web app will enable the developer to experiment and to see how a system will perform under all possible conditions.

**Executive summary**

The project consists of multiple stages starting from a UML Statechart Modeling Tool that receives a UML statechart design and outputs an XML file containing graphical information. This XML file then gets converted into a C++ executable by a Statechart Autocoder before being sent to the State-Machine Target Application for running.

The web server is Flask which will contain most of the data processed by the state machine program. The server will allow us to communicate with all the modules of the project. This includes the UML Statechart modeling tool, the Statechart Autocoder, and the state machine program. Python will be used for the backend and it will manage requests, store and manipulate data, and provide a socket for API purposes. CSS and HTML are likely candidates for the frontend side of our project. State machine programs are valuable in most domains where verifying the validity of a program is of crucial importance. Studies have shown that programs that are evaluated and verified by state machines are generally more reliable, less buggy, and easier to understand. Statecharts are easier to read than code. Statecharts are also useful because they allow developers to understand the how from the why due to the decoupling of behavior from the rest of the program.

As program complexity grows, so must the mechanism that verifies it. Statecharts are known to also scale well which makes them great candidates for the ever-growing complexity that is occurring in today’s industry. These facts are enough to justify the usage of state machine programs as well as provide a reason for having them become the standard for verifying programs.

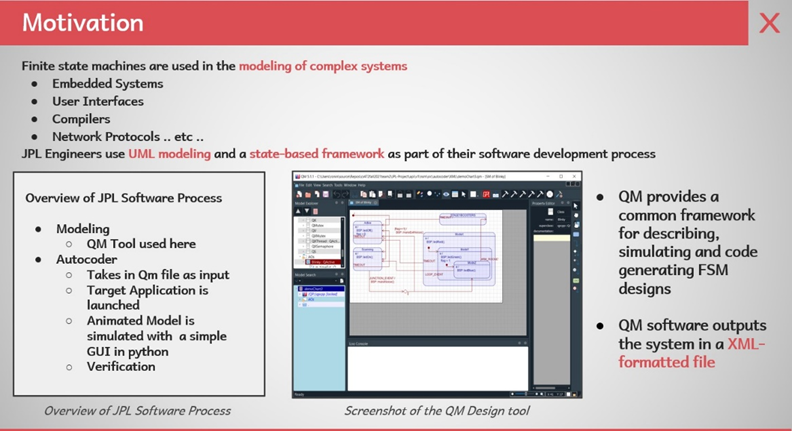
## Motivation

Needs Assessment

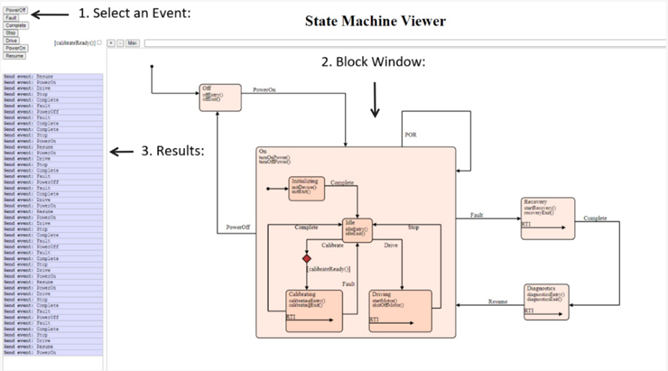
In the past decade, there have been several state machine programs offered to the general public. Programs such as XState, ETAS, and Matlab StateFlow are among the most popular and have provided multiple versions of application software that delineates the sum total of all the states as well as the state-to-state transitions of a program. Although these variations have proven to be useful especially in the software development industry, they do lack certain features that would make such a program complete. The motivation that our team has is to provide JPL with a state machine program that contains some of these features along with an easy-to-use interface that is simple, thorough, and responsive.

JPL currently has a Python implementation of a state machine diagram visualization program, but since it is not open source, we cannot access it. It is generated alongside the C program for the state machine from an XML file using their autocoder and has a zooming function that can focus and follow active states. During a meetup with Leonard and Garth (advising engineers), they also mentioned that a previous team of students had created a plugin or addon for another UML diagram designing tool which allowed a real-time visualization of the resulting state machine.

Finite state machines are used to model complex systems that normally represent programs that have some functional utility to everyday life. Such state machine programs are useful in many contexts and allow the user to understand and improve upon their system as well as represent it in an easily absorbable manner. The screenshot below describes the usage of UML modeling as well as state-based frameworks used by JPL as their software development. They assigned us the task of improving upon previous iterations of their state machine program by having us provide a web-based site that visually represents the states of a given program.



This is the current visualization of our state machine viewer. It is made up of three sections; Select an Event, Block Window, Results.

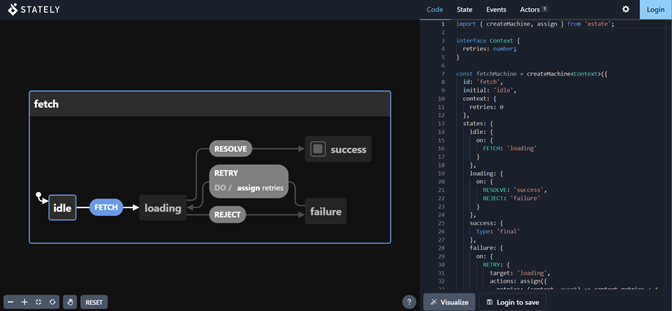


**Competitive Analysis**

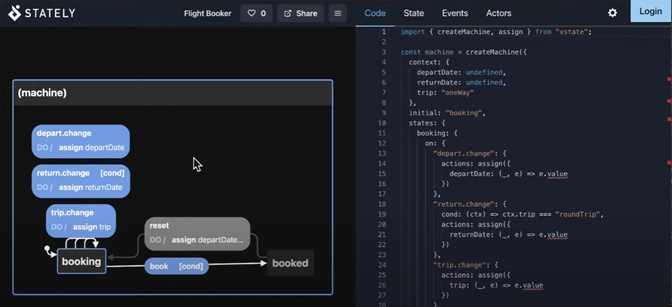
1. XState

Other State Machine visualization tools include <https://xstate.js.org/viz/Xstate>, which takes JSON data describing a state machine, and allows the user to send events to it. XState is a library for creating, interpreting, and executing finite state machines as well as state charts.

When comparing it to our program, we can clearly see that both contain block diagrams representing the states. The main difference is the thematic coloring as well as the fact that our program takes in an XML file that already represents the entire series of state changes as opposed to an on-the-go coding platform that allows the user to code in the state diagram.

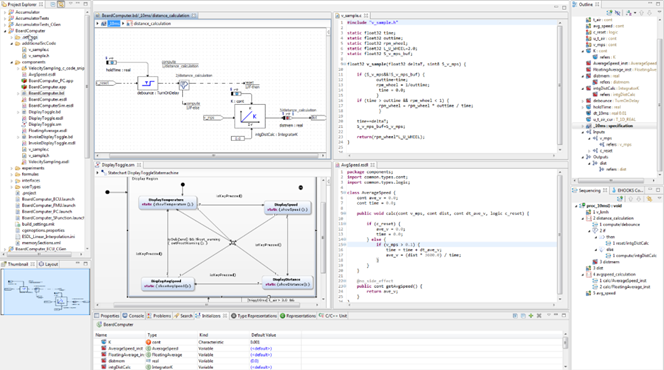


This makes our program a lot simpler to use provided you have the autocoder since the user doesn’t have to code anything from scratch. Although, if a developer is looking to represent a complex application via a state machine program, perhaps XState might be the better fit.

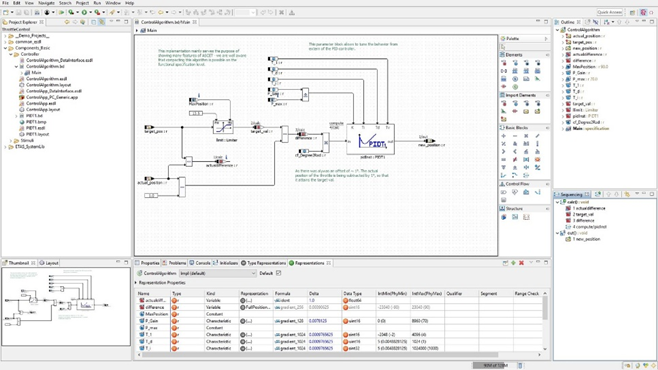


2. ETAS

Another state machine program that has similar functionality is ETAS; which essentially is a software tool for modeling complex systems. Features include the ability to create manageable, maintainable, and easily reviewable models for state machines. Their website is <https://lp.etas.com/>.



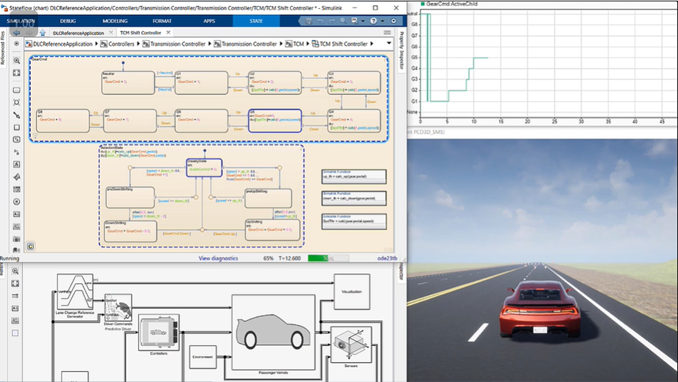
This is perhaps the second most intensive state machine program out of the compared products that allow the developer/modeler to create very detailed state diagrams for precise modeling. When comparing this program with ours, we can attest to the fact that our state machine program is much easier to use and that the learning curve is less steep.



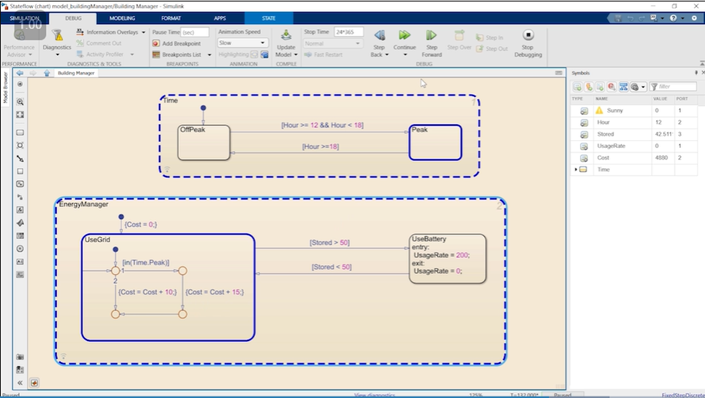
Our program also is open source and free to the public (under the permission of JPL and the program developers) which benefits users since they don’t have to pay to use such a program. It must be said that if the user requires complex modeling and intricate state representation, ETAS might be the better option since it allows for highly customizable parameters. We strive to add such features contained in ETAS in future iterations as our program improves throughout time.

3. Matlab StateFlow

<https://www.mathworks.com/help/stateflow/gs/finite-state-machines.html> MatLab StateFlow allows the creation and simulation of state machines. It provides the ability to model, simulate, test, and build a state diagram from scratch. Features include an interface that connects to a graphical language (as shown in the following screenshot) that includes state transition diagrams, flow charts, state transition tables, and truth tables. It also allows the simulation of state machine programs via a video interface that depicts what is currently happening in a program.



Matlab Stateflow is the most comprehensive product regarding state machine programs on the market as of now that is widely available to the public. However, such a program contains everything our program features plus more, it comes at a premium price, and the learning curve to understanding and utilizing said the program is steep. Our state machine program aspires to obtain more of the features included with Matlab Stateflow over time.



As the screenshot above shows, Matlab Stateflow contains just about every feature a state machine program should have. It should be stated though that if the modeler is looking for an easy-to-use program that interfaces quickly as it depicts states, our state machine program would probably be the better option. Our program in its current state is beneficial for those with simpler needs who are looking for a cost-effective and highly responsive modeling solution for a given system.

**Journal Articles**

[Article 1](https://ieeexplore.ieee.org/document/8600324)

*Automatic Code Generation From UML State Chart Diagrams*

S. E. V. and P. Samuel, "Automatic Code Generation From UML State Chart Diagrams," in *IEEE Access*, vol. 7, pp. 8591-8608, 2019, DOI: 10.1109/ACCESS.2018.2890791.

The article explains that a major downside to using UML statechart diagrams to generate code is that currently, there are two components of a state diagram: hierarchy and concurrency, that cannot be effectively implemented into the generated code. Then, a method is proposed to better implement said components by using classes to represent composite states to abstract multiple states that are in hierarchy or concurrence.

While we likely will not be implementing the proposed design patterns and creating an autocoder is out of our scope, we will still be working with hierarchical state machines, and exploring the autocoder made by the previous JPL team. The proposal and explanation in the article give insight into how hierarchy is implemented and how we may be able to use abstract classes/objects while designing the visual part of the project.

[Article 2](https://ieeexplore-ieee-org.ezproxy.library.unlv.edu/document/5551018)

*UML Everywhere*

D. Spinellis, "UML Everywhere," in *IEEE Software*, vol. 27, no. 5, pp. 90-91, Sept.-Oct. 2010, doi: 10.1109/MS.2010.131.

This article points out the benefits of standardizing UML diagrams. The main benefit is that a standardized notation would allow everyone working in the field to get better at using and understanding it, instead of trying to come up with new notations all the time. The author acknowledges UML’s potential shortcomings but also points out that what’s important for a standard is the user base, and UML has a larger user base than the alternatives.

The mention of the user base is important. During one of our meetings, JPL suggested that we should attempt to generalize the input for our project. Since many people are using UML diagrams to create state machines, our tool could potentially be beneficial for a much wider audience. By generalizing the input specifications, we may be able to allow other people to make use of our tool without compatibility issues, regardless of their circumstances.

[Article 3](https://www-proquest-com.ezproxy.library.unlv.edu/docview/2028724052)

*Comprehensibility of system models during test design: a controlled experiment comparing UML activity diagrams and state machines*

Felderer, M., & Herrmann, A. (2019). Comprehensibility of system models during test design: A controlled experiment comparing UML activity diagrams and state machines. *Software Quality Journal, 27*(1), 125-147. doi:http://dx.doi.org/10.1007/s11219-018-9407-9

A test was conducted with a group of participants using state machine transition tables and a visual diagram of the same machine to evaluate the comprehensibility of state machine representations. The result was that visual diagrams resulted in consistently fewer errors when participants were asked to derive test cases for the state machine compared to transition tables, although both describe the same machine.

Our project, being focused on visually representing a running state machine’s active states, is motivated by the findings in this article. Being able to represent a state machine as a visual diagram allows humans to understand it easier, and we will be exploring other potential elements that may further help with comprehension.

## Requirements

**High-Level Description**

The architecture of our project consists of a Python Flask Backend that communicates with a Web Application Front end. A web client user will be able to monitor and graphically display runtime testing of a hierarchical statechart GUI that can be run-time annotated.

**Functional Requirements Table**

| **ID** | **Title** | **Requirement Description** | **Date** | **Done** |
| --- | --- | --- | --- | --- |
| F1 | Timeline of Events | Graphical display widget showing the history of events, state changes, and guard settings for the current model loaded into the browser application. It should be resettable with zoom and scroll features | 9/17/21 |  |
| F2 | Block diagram  GUI | A UML state chart representation of the state machine, highlighting current guards setting events and current state | 12/1/21 | X |
| F3 | Event window | List of buttons to send Events, list of radio buttons to indicate guards, and drag and drop interface to load a model in XML format | 12/1/21 | X |
| F4 | State Machine | A C++ executable program compiled by the backend to simulate a user-defined state machine. | 9/17/21 |  |
| F5 | Host Machine | The Host machine will set up/deploy the webserver. | 9/17/21 |  |
| F6 | Python Backend | RESTful API that allows client access to available resources and data for the rendering of state machine transition events. | 9/17/21 |  |

**Functional Requirement descriptions**

*F1 Timeline of Events*

Graphical display widget that is loaded automatically at run time of the state machine or when a new model is loaded through the web browser. This timeline shows a history of all events and will be designed to increase the readability of state machine behavior. This will appear as a section inside of the single browser page.

*F2 Block diagram GUI*

A runtime block diagram representation of the state machine that highlights active state or triggered events and has zoom capabilities. This is the main graphical feature of the application.

*F3 Event Window*

This feature allows for interaction within the graphical user interface to send events or trigger events in the current state. These events are simple C++ ENUM-type events for this project.

*F4 State Machine*

This is a C++ program run by the backend server which emulates the state machine operation. It parses the XML/C++ state machine description. Then receives event triggers from F3 and generates the information needed for the front end components(F2 and F3) to render their graphical representations of the state machine.

Inputs:

* XML/C++ source code descriptions of a State Machine following the QP/C++ Real-Time Embedded Framework (<https://www.state-machine.com/products/qp/>)
* HTTP POST commands from front end interface indicating Event signals from the user

Outputs:

* JSON REST API signals to the front-end components to give them the necessary details to render their graphics or text.
* Text status and dialog signals for the user to verify that events have been properly triggered and models correctly loaded.

*F5 Host Machine*

The Host machine will set up/deploy the webserver. The current strategy is for an Oracle Virtual Box Virtual Machine to serve as an endpoint within the local network so that events can be triggered by other machines on the network.

Inputs:

* HTTP POST request

Outputs:

* HTTP Status Codes

Technical Requirements:

OS: Linux

Flavor: Debian 11 (Bullseye)

Memory: Dynamic, minimum 30Gb

Network: Bridged Mode

Packages:

Python 3.8+

Python-pip

Docker CE, docker-ce-clie, containerd.io

*F6 Python Backend*

Inputs:

* HTTP requests forwarded from open ports on the host machine
* HTTP POST commands from front end interface indicating Event signals from the user
* JSON/log outputs from State Machine AutoEncoder

Outputs:

* Graphical data for frontend rendering
* Error/Event log

**Non-functional requirements table**

| **ID** | **Title** | **Requirement Description** |
| --- | --- | --- |
| NF1 | Operating System | The system will run on the Linux server Debian 11 |
| NF2 | Programming Language | The C++ Programming language will be used for the application. The backend server will be written in Python 3.8. |

**Non-functional requirements description**

Our project builds on last year’s team autocoder to expand its functionality and add a web application that runs alongside it. Since our application is browser-based it should work for most operating systems. The web application will be hosted on a virtual machine to allow for communication between the machines on the same network. The front end will be developed using HTML/CSS and Javascript so that the application is accessible from a wide variety of browsers. The Python Flask Back is lightweight and highly scalable. For more information on why we chose these technologies please refer to the design section of our document.

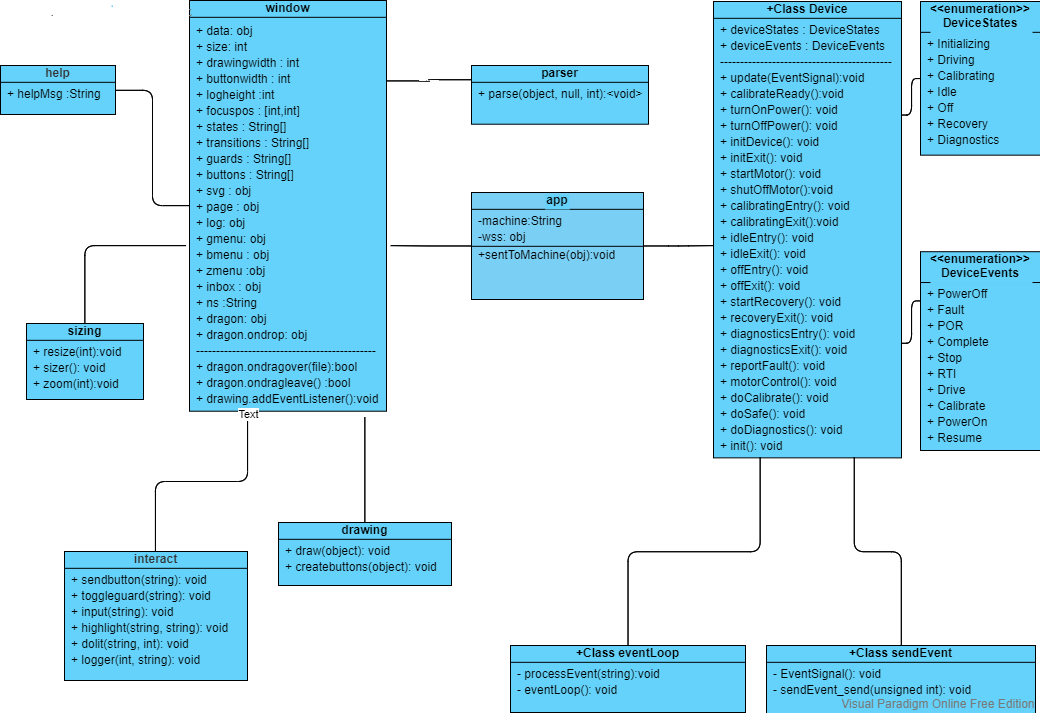
## Design Document

Our system is built around the NSM autoencoder developed in C++ by the UNLV student team of Reynolds B et al.([GitHub - JPL-UNLV-CS-2021/nsm: NSM State Machine](https://github.com/JPL-UNLV-CS-2021/nsm)). Our goal is to use this system as the core of a web-based Python Flask application to serve a simple, intuitive, and flexible graphic interface for loading and interacting with state machines specified in either C++ or the QP framework XML format for representing UML hierarchical state machine(HSM) designs.

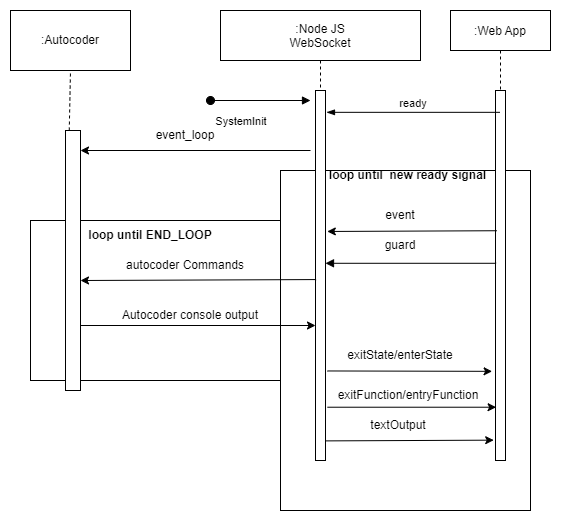
**UML for State Machine Viewer App**

To describe our design choices and how our system operates we've produced a full UML of the classes and how the three main components of our system interact.

1. Web Front End - Contains HTML and React components to render interface controls and output text and diagram.
2. Python Flask Back End - A Flask back end server which will serve the components, save log data, and run the autocoder.



*Figure 1: UML class diagram for Web State Machine Viewer*



*Figure 2: Sequence diagram for Web State Machine Viewer*

**Web Client Front End**

1. XML Drag-n-Drop - An interface area where the user can drag and drop XML files representing the state machine.
2. Event Window - A component that has a list of buttons representing possible FSM events and a set of radio buttons which represent a set of true/false guards that will affect how the state machine performs.
3. Block Diagram - A graphic canvas window that will render a full block diagram highlighting which states and transitions are currently active and any other information about the current operational state of the machine.
4. Timeline Diagram - A graphic canvas window that will render a timeline view of the active states that have been triggered during the current run of the FSM viewer.
5. Results Window - A console-style window showing the text line outputs of the state machine. Can be saved into a text log file for later review.

**Python Flask Back End**

1. Event Queue - A FIFO queue for holding Events sent from the Event Window component to be delivered to the autocoder to trigger the next state machine transition
2. Router - a Python Flask routing script that routes which pages(main, index or help)need to be served and where the needed source HTML resides.
3. Parser- Python script which parses XML and generates code for building the FSM, Then runs the compiler
4. Autocoder - Built state machine which runs events and takes guards as inputs and generates the information needed for re-rendering the Block Diagram and TimelineDiagram. The Results Window is strictly the text output of the Autocoder as it is operating.

**Host machine**

The front end will be deployed via Docker. Docker is a platform for managing

and running applications. This is a lighter-weight alternative to a virtual machine, especially useful for long-running services. Docker allows for dependencies to be packaged with the source code, such that setup and deployment can be automated and version control for dependencies enforced. This also allows for portability assuming the selected WSGI server is compatible with the host OS. The following steps are an overview of the boot-up process.

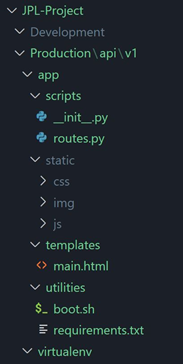
1. Host machine calls docker build/compose to configure the docker-container.

a. Build script initiates a git pull from master to ensure it has the most recent version of the project loaded, constructs the file system within the new container, assigns relevant permissions to processes, and exposes the relevant port for access. The entry point is defined as boot.sh.

b. Boot shell script is called, which loads a virtual environment and installs any dependencies/libraries required by the process i.e. Flask & Unicorn via a recursive pip install command.

1. The WSGI server, in this example Gunicorn, is deployed to a predefined IP address/port local to the docker-container. Note that a predefined port will be exposed such that while active, the docker box will redirect requests to this locally hosted docker container. The docker container will then actively listen via any machine on the network with the appropriate credentials.

**Python Flask Backend**

Flask is a minimalist, python-based, web framework that is highly configurable. For our purposes, it will serve as a host for our backend in conjunction with Gunicorn, a popular WSGI (Web Server Gateway Interface ) that will forward requests from the webserver to the Python/C++ backend. Components will be constructed for different aspects of the API, including the C++ autocoder which will be modularized and interacted with as a separate process, i.e. through the socket. Commands from the autocoder will be forwarded to the front end such that graphical data can be displayed, while event logs will be stored locally to a mounted folder on the Docker host machine to ensure persistence.

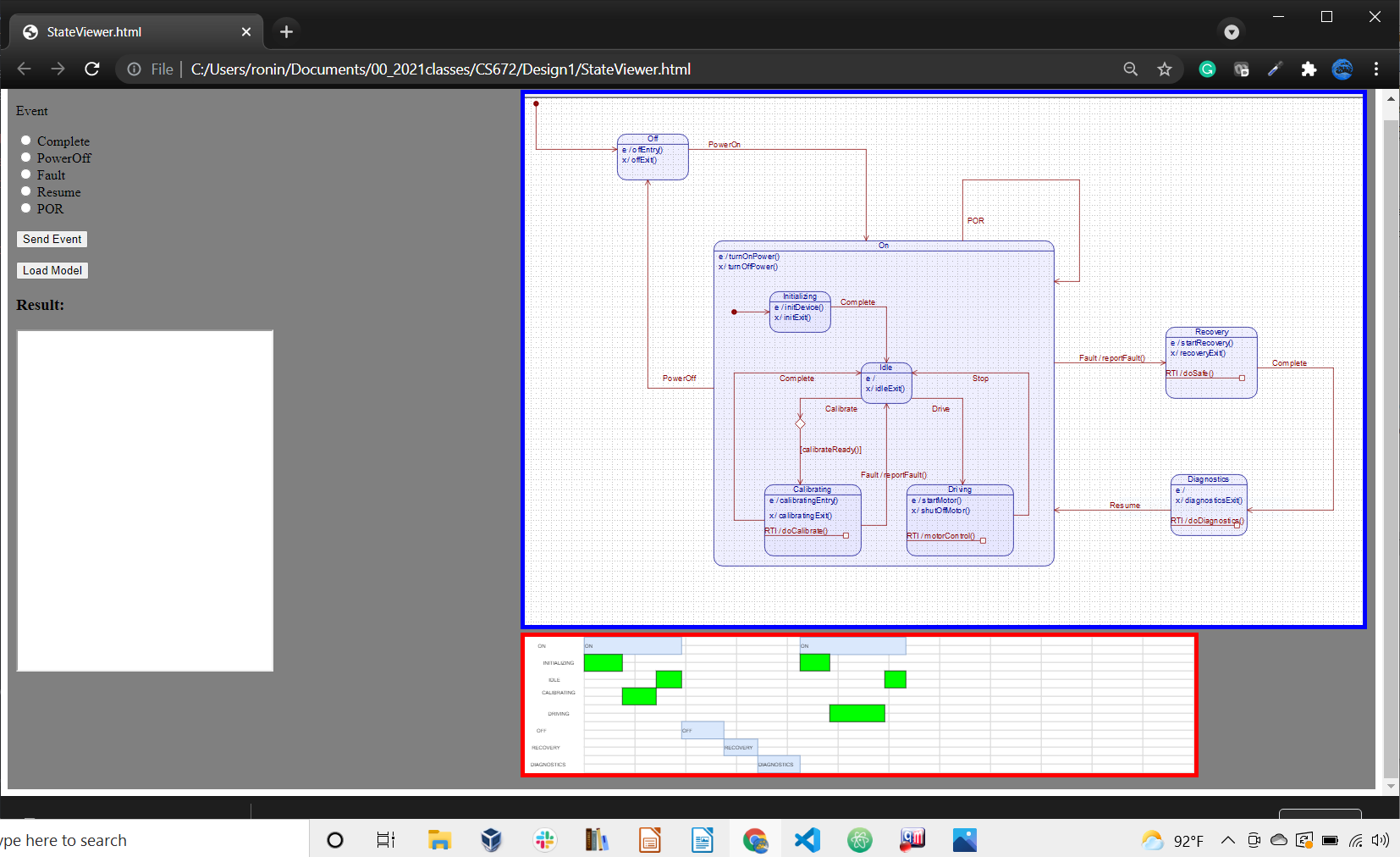
*Figure 3: Sample File structure*

**Organization**

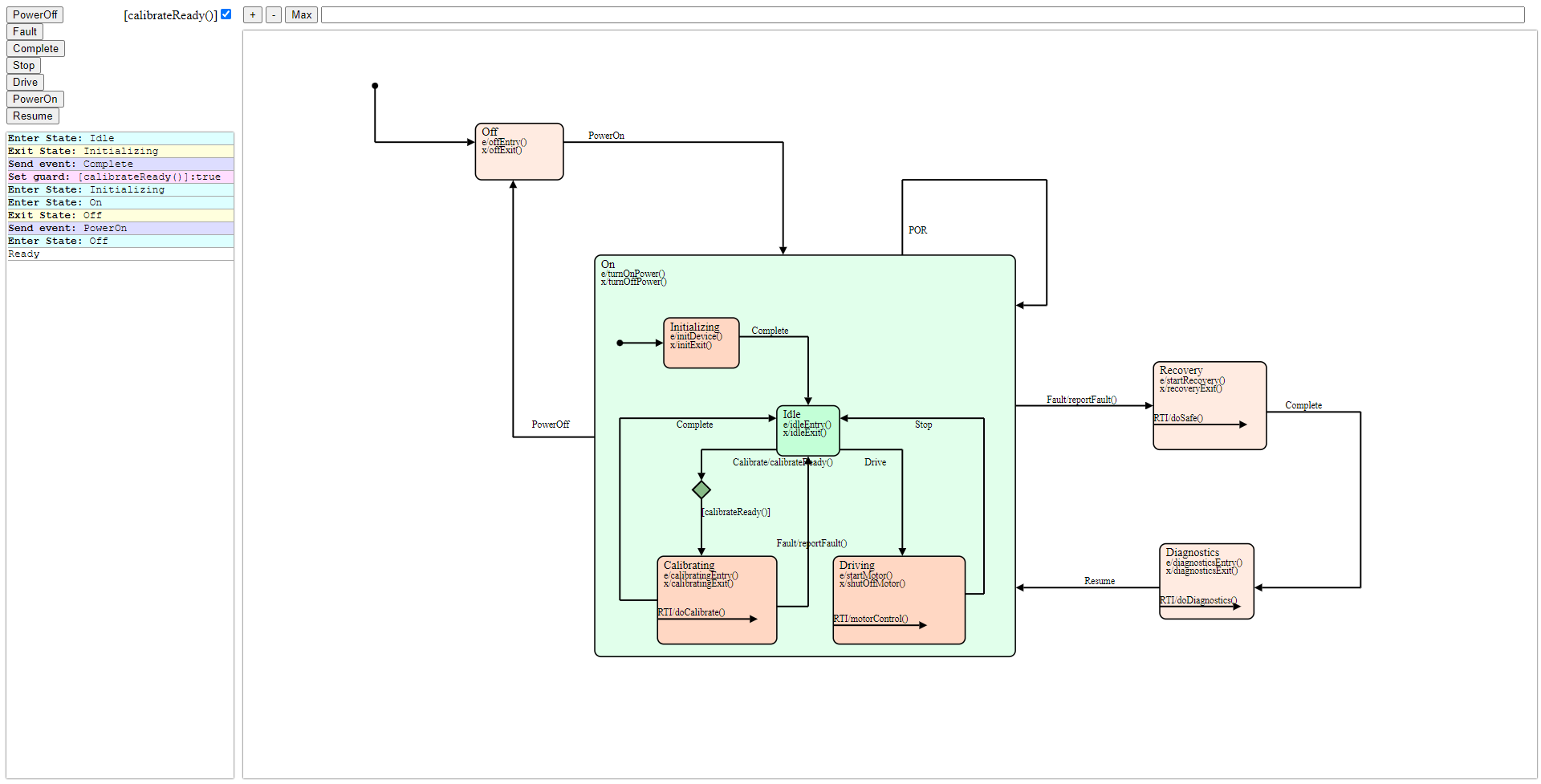
1. The project will be divided into Development | Staging | Production
   1. Code to indicate which code that is being worked on, tested, or is working respectively. Production code will be loaded into the docker container.
2. The entry point of the file structure is named run.py and will be contained in the app folder. It will import an instance of the application created with a basic import from Production/api/v1/app/scripts/\_\_init\_\_.py. This file sets the default locations for the .html/.js/css that will be rendered by Flask’s internal methods.
3. Routes.py determines what actions are performed, such as rendering of content, and calls to external processes i.e. the autocoder when an endpoint is hit.
4. Static contains the styles/javascript/image files associated with each page.
5. Templates will contain all .html files (single page for now)
6. Utilities will contain the DockerFile and Docker-compose.yml file used to generate the Docker container. The boot script for the version of the API as well requirements for the virtual environment are also contained within this folder.

## Prototype

The purpose of our project is to build a web application based on Python Flask which allows the user a graphical interface for interacting with a state machine design that is easy-to-use, visually informative, and flexible. The browser window will show three different areas for displaying and interacting with the state machine.



*Figure 3: First Prototype*



*Figure 4: Final Prototype*

**Viewer windows**

1. Event trigger and response- On the left, it allows the user to send events or load a new model. Text Results of every action will display in the “Results” window
2. State Machine Block Window - On the upper right side, a block diagram representation of the state machine with the ability to highlight the active state or triggered event
3. State Machine Timeline View- On the lower right, a timeline view of state changes in response to event inputs.

## User Study

We decided to model our user study with a task we have the users complete on the current version of our product. Before providing the task we give a brief introduction of what our project is and the purpose behind it. The task involves users taking an XML file and uploading it onto the website. We ask a few general questions before the task and the interview questions at the end of the task. The before task questions are about the users’ demographics and technical background.

The two alternative designs of our prototypes include a light and dark mode with modifications to the arrangement of the sections on the website.

**Interview Questions**

*What do you need in order to use the website?*

We wanted to understand what barriers users would need to overcome in order to effectively navigate the website. Whether it was a general understanding of the purpose of the website or the functionality of individual features on the website itself. We were hoping to highlight areas that could cause confusion or seem out of place.

*What would you like to see changed on the website? (font, buttons, background, etc)?*

While this may be a low-level change we wanted general feedback on what makes a user use or not use a particular UI. Since our application is meant for monitoring we wanted suggestions on how we could improve the look of our website to be visually appealing and effective.

*On a scale of 1-10, how intuitive is the webpage? (10 being the most intuitive)*

Again, our emphasis was on how easy it is to use and understand the purpose of the website and the role of each main section on the page.

*What features on the website did you like/dislike?*

The four main sections on the website each serve a role where some features are more important than others. We wanted our users to let us know how each section was being represented and what we were doing right/wrong.

*How much of the screen should the block diagram take up?*

The block diagram is one of the most important sections on our website. We wanted feedback on how this content should be displayed to better emphasize that.

*Do you have any other suggestions?*

*Do you think there’s a better way to present the content?*

These last two questions were used to receive feedback on any other areas of our product we may have missed or anything else the users felt needed to be brought to our attention.

**Demographics**

Since we are working on a project for JPL Engineers our product (arguably) is intended for customers with a more technical background and use case. However, state machines are highly researched so it would be okay to assume that our product could potentially interest other CS students, people who use the QM modeling tool, or others in relating industries/interests that pertain to monitoring state machines.

Our study included 3 individuals in the age range of 23 - 40.

User 1

* 23 year old
* Senior Computer Science student

User 2

* 33 year old
* Civil engineer

User 3

* 40 year old
* Ph.D. Computer Science student

**Results**

All of our users agreed that they would need some basic understanding of what state machines are or an overview of the purpose of the website. A majority of our users were not familiar with state machines. All users had issues with the overall layout of how the content was being presented.

*Positive Feedback*

A majority if not all of the positive feedback was on the low-level content of our application. The drag and drop file upload functionality, dark mode option, and potential of event buttons.

*Negative Feedback*

The major issues users had been with the overall arrangement of the sections on the website, confusion about the purpose of the event buttons, and not being able to see the block diagram clearly. One of our users pointed out that we should rearrange the website to have the block diagram as the first major component on the left-hand side with the event buttons on the right like a toolbar. Since our current page layout makes the block diagram hard to see which is the most important feature on the page.

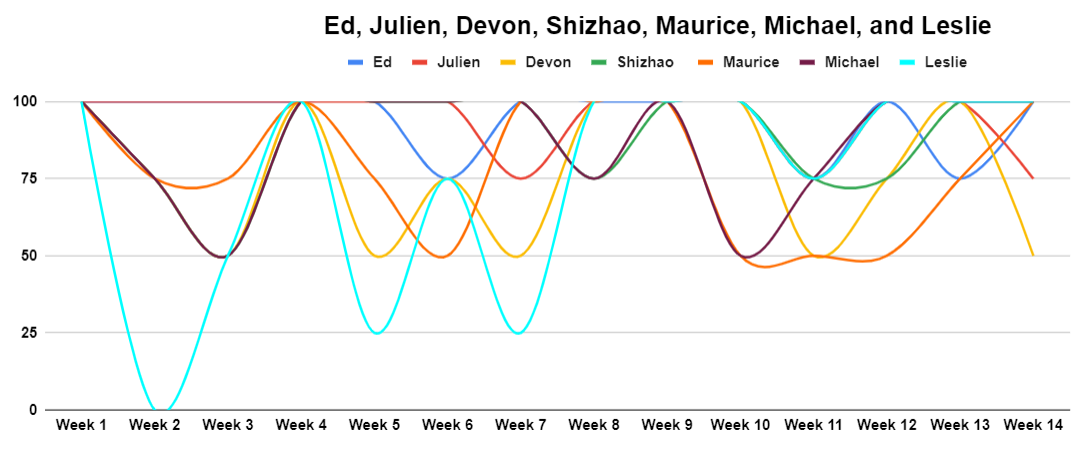
All users had confusion pertaining to the event buttons; this confusion seems to be caused by the lack of functionality in its current state and how it is presented on the page. The same user who suggested moving the event buttons to the right of the diagram to look more like a toolbar said that they couldn’t tell from looking at the page that the event buttons were associated with the block diagram. The majority of users suggested having the two main graphical components on the left-hand side and the text-rich components on the right-hand side to emphasize the visuals.

Another notable recommendation was that the upload button should be more user-friendly and should be replaced with a reset button once a file is loaded to avoid overriding the results in case a user accidentally drags and drops a file while already monitoring a previous state machine. Overall our website ranked low on how intuitive it was to use and to understand what features were important and the purpose of each.

## Team Reporting

**Team Tasks Chart**

The chart below shows the percentage of tasks completed for each team member per week. Team members that performed at the same rate are shown on the same line. Our front-end team members are Devon, Shizhao, Michael, Maurice, and Leslie. Our back-end team members are Ed and Julien.



Edward:

1. Week 10
2. [Commit 1:](https://bitbucket.org/stefika/cs472fall2021team2/commits/0875bc751a247ab3511fc24a1dfc7ca0ae3cd498) Add Linux fix
3. Work on a parser from the autocoder (complete)
4. Week 11
5. [Commit 2:](https://bitbucket.org/stefika/cs472fall2021team2/commits/7f47be4a553b2c4c374baa8d7f4d2d4af18ca5bc) Allow the program to run on Linux Ubuntu 20.04
6. [Commit 3:](https://bitbucket.org/stefika/cs472fall2021team2/commits/58aa3dfdbcf4de76521e05ff3ac3eee39a4bc0ae) Change GeneratedHSM autocoder to run in command-line mode
7. Finalize interface specification (complete)
8. Start presentation slides (complete)
9. Week 12
10. [Commit 4:](https://bitbucket.org/stefika/cs472fall2021team2/commits/e61901659268a142ec5c3ba56fe45c1441a075a5) Add Active State signal to Device autocoder
11. [Commit 5:](https://bitbucket.org/stefika/cs472fall2021team2/commits/cf9f15b427e5e5e2df23593eac7e613a13bd8ce0) Clean up autocoder
12. Week 13
13. [Commit 6:](https://bitbucket.org/stefika/cs472fall2021team2/commits/653daf4f330946e1da401bee4b27cdeef13fcdaf)  Add JSON format strings to output
14. Revise flow diagram and sequence diagram (complete)
15. Week 14
16. [Commit 7:](https://bitbucket.org/stefika/cs472fall2021team2/commits/f0537bd60a91b51a414c5fd516cc21561079f7bb)  Add JSON transitions
17. [Commit 8:](https://bitbucket.org/stefika/cs472fall2021team2/commits/136401026f221d9e08a214ef5e3aee9f4eb795a4)  Include compiled event\_loop executable

Julien:

1. Week 10
2. Write a parser for autocoder (complete)
3. Week 11
4. [Commit 1](https://bitbucket.org/stefika/cs472fall2021team2/commits/cd3c96290339ae055e56d1194bfe49cd5a5c8453): Revise parser method to output JSON, adjusted drag and drop to accommodate.
5. [Commit 2:](https://bitbucket.org/stefika/cs472fall2021team2/commits/791b602f8c2596f8f3bed37868f96334757156e8) Add routes for HTML/CSS to host with flask, and combined index/window HTML display
6. Week 12
7. Merge all code together for a demo
8. Week 13
9. Fix backend to connect to the front end
10. Week 14
11. [Commit 3:](https://bitbucket.org/stefika/cs472fall2021team2/commits/96d04dbe3ce89094f2322727a2b698c78c45f6ff)  Make test branch for testing WebSocket performance

Devon:

1. Week 10
2. Create a new folder for demo and clean up files (complete)
3. Week 11
4. [Commit 1:](https://bitbucket.org/stefika/cs472fall2021team2/commits/6b758dfe2b4d6042a68e8b9324cf7a88d6257bc2) Merge in code from demo branch
5. [Commit 2](https://bitbucket.org/stefika/cs472fall2021team2/commits/0745e24a8bd69628c98247363b00d9be2d6780b8): Add Horizontal Scroll
6. [Commit 3:](https://bitbucket.org/stefika/cs472fall2021team2/commits/cfa9487969559eb6d6dfbea27e5ef58d43e22475) Clean up the CSS, not in the styleshee.css and multiple body's added comments
7. [Commit 4:](https://bitbucket.org/stefika/cs472fall2021team2/commits/1757ae3f5ca2d459292529c1a8c466b5d657025d) Update the demo branch to work off HTML tables and merge lost code
8. Week 12
9. [Commit 5:](https://bitbucket.org/stefika/cs472fall2021team2/commits/9109367385949dd276e77eb8da3205fe93411223) Resize console log, Block window, Result, and Timeline
10. Week 13
11. Week 14

Shizao:

1. Week 10
2. [Commit 1:](https://bitbucket.org/stefika/cs472fall2021team2/commits/124bed853b54c088b78a964e86709fd1ae1ba068) Rewrite drawing to use SVGs instead of canvas pt 1
3. [Commit 2:](https://bitbucket.org/stefika/cs472fall2021team2/commits/5dfa4124f28675a919ea2db970fab5db49af16f3) Rewrite drawing to use SVG instead of canvas pt 2
4. Experiment with XState (complete)
5. Week 11
6. [Commit 3:](https://bitbucket.org/stefika/cs472fall2021team2/commits/9001d1bd4495c30402ea8430e7f504636c7d8a7f) Rewrite code and add dynamic buttons, log menu, and zooming function with SVGs
7. [Commit 4:](https://bitbucket.org/stefika/cs472fall2021team2/commits/ce7089c0ccce9d09b937228cb2e47882e4a2deba) Add draggable window, state highlighting, and guard coloring
8. [Commit 5:](https://bitbucket.org/stefika/cs472fall2021team2/commits/385a781e8e37147bd08f5bce7f2fd7963015606b) Add arrows to transitions and highlight a state the viewer will try to scroll so that state is in the center
9. [Commit 6:](https://bitbucket.org/stefika/cs472fall2021team2/commits/03ccefcf79d81da020d0d1fffaee7a549e0af639) Merge everything with drag n drop and the new parser
10. Week 12
11. [Commit 7:](https://bitbucket.org/stefika/cs472fall2021team2/commits/e07ebf6b74d96d1bca989c6a75f006b0f75f0e88) Make page responsive below 700px in width and bug fix
12. [Commit 8:](https://bitbucket.org/stefika/cs472fall2021team2/commits/351c10134ba5316f69add3cb1d9da72003ec5b93) Split JS into multiple scripts for easier managing
13. [Commit 9:](https://bitbucket.org/stefika/cs472fall2021team2/commits/bd223cfe99a55fb28cd93d93ac5479e55a3fa461) Fix some bugs with resizing and renamed variables to be more readable
14. Week 13
15. [Commit 10:](https://bitbucket.org/stefika/cs472fall2021team2/commits/8445fb3a75880e60f1b989886f41254e1bf83b18)  Fix bugs
16. [Commit 11:](https://bitbucket.org/stefika/cs472fall2021team2/commits/ce1808d195f77f1ceec1e3fa1e799146781ef197) Fix guard menu
17. Week 14

Maurice:

1. Week 10
2. Week 11
3. Edit presentation slides based on feedback (complete)
4. Week 12
5. [Commit 1:](https://bitbucket.org/stefika/cs472fall2021team2/commits/9e507d9af0db6af23be1ce928c30a98cc19fae02)  Clean up code and add documentation
6. Week 13
7. [Commit 2:](https://bitbucket.org/stefika/cs472fall2021team2/commits/d242922c143ab8758ab204f36aa6b761332cc4c2) Update help page
8. [Commit 4:](https://bitbucket.org/stefika/cs472fall2021team2/commits/dfc5d089717e547bab0f1f659f9ee9916c888a6e) Fix formatting on the help page
9. [Commit 5:](https://bitbucket.org/stefika/cs472fall2021team2/commits/9386cfb73bdc8745f3d55faf757a6c8910ffef7b) Bug Fix
10. Help with fixing class UML (complete)
11. Week 14
12. Edit the motivation section on Design Doc 3 and fix any typos (complete)

Michael:

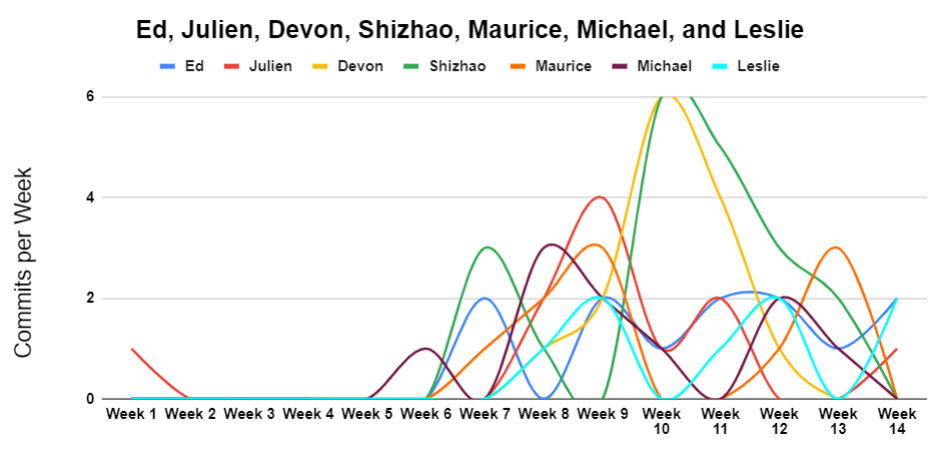
1. Week 10
2. Look into XState to see if it can be used (complete)
3. [Commit 1:](https://bitbucket.org/stefika/cs472fall2021team2/commits/df34974010ab2c5d12c31240f706c2bc953d089a) Implement XState with React and embedded visualizer
4. Week 11
5. Clean up page layout HTML/CSS (complete)
6. Edit class UML (complete)
7. Week 12
8. [Commit 2:](https://bitbucket.org/stefika/cs472fall2021team2/commits/10e7612976539b962a61e0e1073db763bd5c84ea) Add title and description to the webpage
9. [Commit 3:](https://bitbucket.org/stefika/cs472fall2021team2/commits/914cd6dbc464ab4123ada97d050165a0da958202) Make the welcome message hide when the input file is dropped onto a webpage
10. Week 13
11. [Commit 4:](https://bitbucket.org/stefika/cs472fall2021team2/commits/1196d1f6f75f4c939dca9fd3746db07cea269951)  Re-Add title/description for updated code
12. Help with fixing class UML (complete)
13. Week 14
14. Create the poster for presentation (complete)

Leslie:

1. Week 10
2. Look into XState to see if it can be used (incomplete)
3. Work on scrollable console log (complete)
4. Week 11
5. [Commit 1:](https://bitbucket.org/stefika/cs472fall2021team2/commits/3cdee7f801831a4b36a12e5b1fb05cca2c5d1ab1) Add results window with scroll bar
6. Add a slide for team reporting for presentation slides (complete)
7. Week 12
8. [Commit 2:](https://bitbucket.org/stefika/cs472fall2021team2/commits/a87c8669e761db6893a67691cb25230b023baac7) Add a new folder layout experiment to change/clean up the layout of the page. Unfinished
9. [Commit 3:](https://bitbucket.org/stefika/cs472fall2021team2/commits/d160cb614d86fba9524afc1790ab854a41be4726) Update help page and add an image
10. Updated presentation slides based on feedback (complete)
11. Week 13
12. Update help page (incomplete)
13. Week 14
14. [Commit 4:](https://bitbucket.org/stefika/cs472fall2021team2/commits/95d65f0c484f2a0b3c7f668e600445ae2d23cfba) Update graphic on the help page to show changes in color
15. [Commit 5:](https://bitbucket.org/stefika/cs472fall2021team2/commits/333382a922279d777b84c1d7756fc4d73c555242) Bugfix
16. Update Team Reporting section for Final Doc 3 (complete)

**Team Commit Chart**

The chart below shows the number of commits per team member per week directly from BitBucket. The Y-axis is the number of commits per week in the range of 0 - 3 commits. This range was picked arbitrarily based on the maximum number of commits for all the weeks, which happened to be three.



**Success Report**

The following table will be used throughout this project and will help determine if a team member has met the commit requirement.

| **Team**  **Member** | **Doc 2** | **Obtains**  **Grade** | **Doc 3** | **Obtains**  **Grade** | **Total**  **Commits** | **Passes**  **CS 472** |
| --- | --- | --- | --- | --- | --- | --- |
| **Ed** | **4** | **Yes** | **8** | **Yes** | **12** | **Yes** |
| **Julien** | **7** | **Yes** | **3** | **Yes** | **10** | **Yes** |
| **Devon** | **3** | **Yes** | **5** | **Yes** | **8** | **Yes** |
| **Shizao** | **4** | **Yes** | **15** | **Yes** | **19** | **Yes** |
| **Michael** | **6** | **Yes** | **5** | **Yes** | **11** | **Yes** |
| **Maurice** | **6** | **Yes** | **5** | **Yes** | **11** | **Yes** |
| **Leslie** | **3** | **Yes** | **5** | **Yes** | **8** | **Yes** |

## Conclusion

## Stagnation of state machine software has left a gap in the industry applications for our team to capitalize on. Under the tutelage of NASA’s Jet Propulsion Laboratory employees, Len Reder and Garth Whitney tasked us with building a web-based graphical interface of the previous semester’s project, the NSM Autoencoder. Len Reder and Garth Whitney at JPL instructed us to give it a stable, flexible, and intuitive interface to assist engineers and give them a powerful tool for designing and planning complex systems. The tool will take a state machine described with Quantum Machine open-source QP Framework XML description and create a simple clean graphical front-end for the design team to interact with and test their system. The entire project was built with standard open-source web technologies to help ensure that the system will run a wide variety of platforms. This also ensures future improvement or additions to the project can be done using commonly used programming languages.