



# WPI

## **7Factor AWS Cost Analysis Tool**

**A Major Qualifying Project Report**

Submitted to the faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science

In cooperation with 7Factor, LLC

Submitted March 4, 2022

**By:**

Edward K. Carlson

Katie Lin

Joshua D. McKeen

Benjamin Staw

**Project Advisor:**

Professor Joshua Cuneo

This report represents the work of WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review. For more information about the projects program at WPI, please see: <https://www.wpi.edu/academics/undergraduate/major-qualifying-project>

# Abstract

---

7Factor is a software development company that specializes in consulting clients from other companies who contract them to create custom devops solutions. These solutions help other companies manage and streamline their process of creating, testing, and deploying programs, thereby also cutting costs. 7Factor needed a way to find the optimal EC2 virtual machine configurations to support ECS clusters provided by AWS in order to reduce costs. For this project, we developed a web-based platform that accomplishes the base requirements to fulfill their current needs.

# Acknowledgements

---

We would like to thank the following people for their involvement and help throughout this project:

*Joshua Cuneo, Computer Science Professor at WPI — Our Project advisor*

*Jeremy Duvall — CEO of 7Factor*

*Lindsay Duvall — 7Factor Coordinator*

This project would not have been possible without their help, guidance, and feedback.

# Table of Contents

---

<b>Abstract</b>	<b>1</b>
<b>Acknowledgements</b>	<b>2</b>
<b>Table of Contents</b>	<b>3</b>
<b>List of Figures</b>	<b>5</b>
<b>Executive Summary</b>	<b>6</b>
About 7Factor	6
AWS Cost Analysis Needs	6
Designing an AWS Cost Analysis Tool	6
Results	7
Meeting 7Factor’s Objectives	7
User Experience	7
Future Work	8
<b>Introduction</b>	<b>10</b>
<b>Background</b>	<b>11</b>
About 7Factor	11
AWS APIs	11
Algorithm	12
Buckets & Marbles Analogy	12
<b>Methodology</b>	<b>14</b>
Type of Platform	14
Python & Fastapi	14
Boto3 (Python AWS SDK)	15
Javascript/Typescript	16
React/MUI (Material User Interface)	16
Git	17
Docker	17
API Calls: Auth0	17
Auth0 integration	18

Database structure: Postgres	18
Figma	18
Unit Testing (for Backend Algorithm)	23
Project Set-Up	23
Workflow	24
Team Organization	24
Project Progress	24
A Term	25
B Term	25
C Term	25
<b>Obstacles</b>	<b>26</b>
Amazon Web Services Structure	26
Amazon Web Services API	27
Auth0	27
Web Development - API Endpoints	28
Docker	28
<b>Results</b>	<b>28</b>
Meeting 7Factor's Objectives	28
User Experience	30
Design Components	30
Login Page	31
Home Page	31
Optimization Page	34
Settings Page	36
<b>Future Work</b>	<b>37</b>
<b>Conclusion</b>	<b>37</b>
<b>References</b>	<b>38</b>
<b>Glossary</b>	<b>40</b>
<b>Appendix</b>	<b>41</b>
Algorithm Pseudocode	41
Set-up for the AWS Cost Analysis Project	41

# List of Figures

---

- i. Home Page
- ii. Optimization Page
1. Buckets & Marbles Diagram
2. Current Configuration Page Figma Mockup
3. Workloads Page Figma Mockup
4. Scrapped Current Configuration/Workloads Page
5. Home Page Figma Mockup
6. Optimization Page Figma Mockup
7. Workflow Table
8. 7Factor's Company Website
9. UI Sample
10. Auth0 Login Page
11. Home Button Icons
12. Home Page
13. Cluster Information
14. How to Add a Cluster
15. Home Page with Newly Added Cluster
16. How to Remove a Cluster
17. How to Reach the Optimization Page
18. Optimization Page
19. How to Adjust CPU/RAM Paddings and Instance Family
20. Total Savings Display
21. Optimization Page: Showing Total Savings
22. Settings Page Icon
23. Settings Page

# Executive Summary

---

## About 7Factor

7Factor develops secure, stable, scalable solutions for clients in many industries, including the financial sector, healthcare, aviation, and the media. They are trusted by many respected brands as well as by startups and smaller companies alike. 7Factor builds cloud architectures, continuous delivery pipelines, and custom systems and applications. In addition to performing project work, 7Factor also offers advice contracts to provide help navigating through complicated issues (7Factor).

## AWS Cost Analysis Needs

In this project we were tasked by 7Factor to develop a tool that could gather information about some of their cloud infrastructure hosted on Amazon Web Services (AWS) and provide recommendations for potential cost-saving alterations to the configuration of this cloud infrastructure. 7Factor utilizes AWS's Elastic Compute Cloud service, otherwise known as EC2, in conjunction with AWS's Elastic Container Service (ECS). AWS EC2 provides a variety of different instance types for customers to choose from and assign tasks. Each of these instance types have their own specifications, which include differing amounts of memory, vCPUs (threads of a CPU core), processor types, and networking capabilities. The EC2 instances are then linked with containers created inside AWS ECS.

As part of defining ECS containers, AWS requires that task definitions be set up. These task definitions specify minimum CPU and memory requirements in order for the task to run successfully. As a consequence, these tasks must be linked with EC2 instances that have enough physical hardware to accommodate the task's needs. Our tool analyzes any current ECS task configuration in a given EC2 cluster and attempts to optimize the types of instances being used in the cluster while still providing enough physical resources for the ECS containers.

## Designing an AWS Cost Analysis Tool

When designing the tool, we had many considerations, including how the user would use the tool and how we would create the tool from a technical standpoint. In order to decide our overall design, we discussed what tools would be appropriate for the task based on researching what other teams have used for similar tasks. Based on our research and previous experience, we decided on React and Javascript for the frontend interface of our tool, Python as our backend

server handling the algorithm logic, and Docker for containerizing our project. We also used some third party libraries such as Material UI in order to have a more unified, attractive website design. For our backend, we used FastAPI, a Python library that allows users to quickly create a fast and simple backend server. By choosing FastAPI, we were able to quickly create the important parts of the project, such as the logic handling user clusters and the algorithm to determine the cheapest configuration a user could use. We also had other tools to help streamline the development of our project, such as Figma, which allowed us to create quick mock-ups of our website to decide on an overall design before dedicating the time to creating it. We also used some boilerplate code that allowed us to skip the long process of creating basic functionality and gave us a good structure to start with. We also integrated a pre-existing user authentication service, Auth0, into our tool, as 7Factor requested that we use it in order to make the overall login experience the same across their suite of tools.

## Results

### Meeting 7Factor's Objectives

We built a web app with a backend that can be deployed from a Docker container. Our tool presents the user with the option to select any ECS cluster and run our optimization algorithm on it. It then displays the most cost-effective cluster configuration to the user. The Python algorithm we developed compares and analyzes the current task and cluster configurations to identify the best configuration based on the on-demand hourly costs. Based on the difference in on-demand hourly rates between the current cluster configuration and the proposed configuration, our tool provides a savings estimate based on 30 days of usage.

### User Experience

1. Design Components: When we designed our web application, we wanted our UI to be simple and intuitive while also matching 7Factor's company branding.
2. Pages: Our application includes four main pages:
  - A. Login
  - B. Home
  - C. Optimization
  - D. Settings

The home page of our project showcases a list of clusters the user can choose from. It allows users to add and remove clusters that can be put through the optimization algorithm.



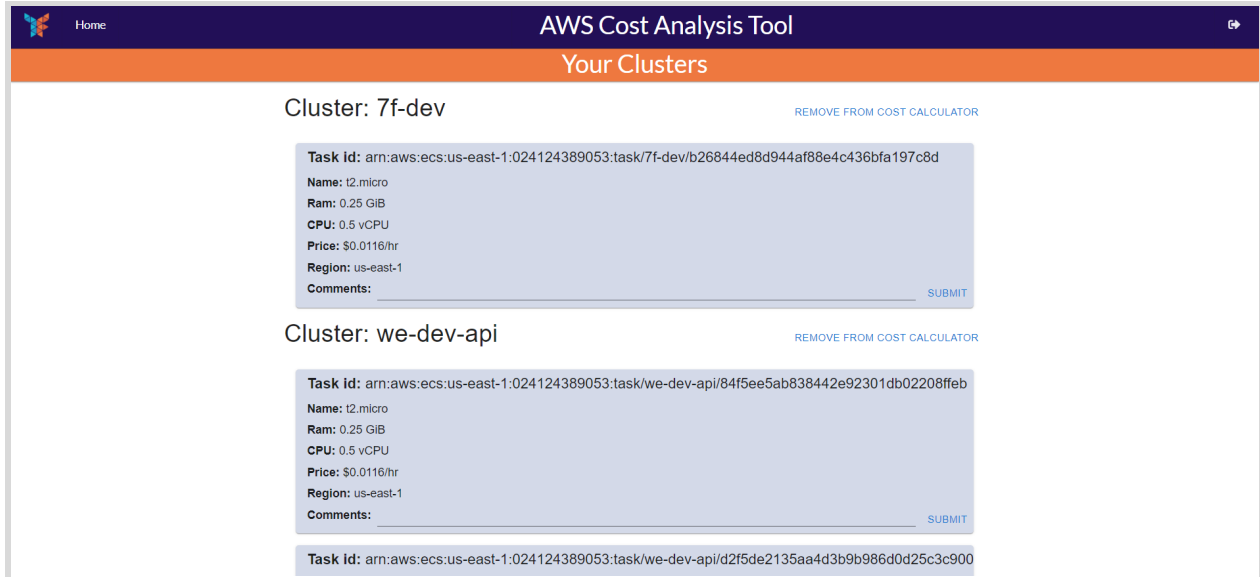


Figure i: Home Page

The optimization page shows the tasks belonging to a specific cluster on the left side and the optimization details on the right side. These details are extracted from an algorithm that assesses the needs of the tasks in the given cluster and then determines the most cost-effective configuration that can accommodate those tasks.

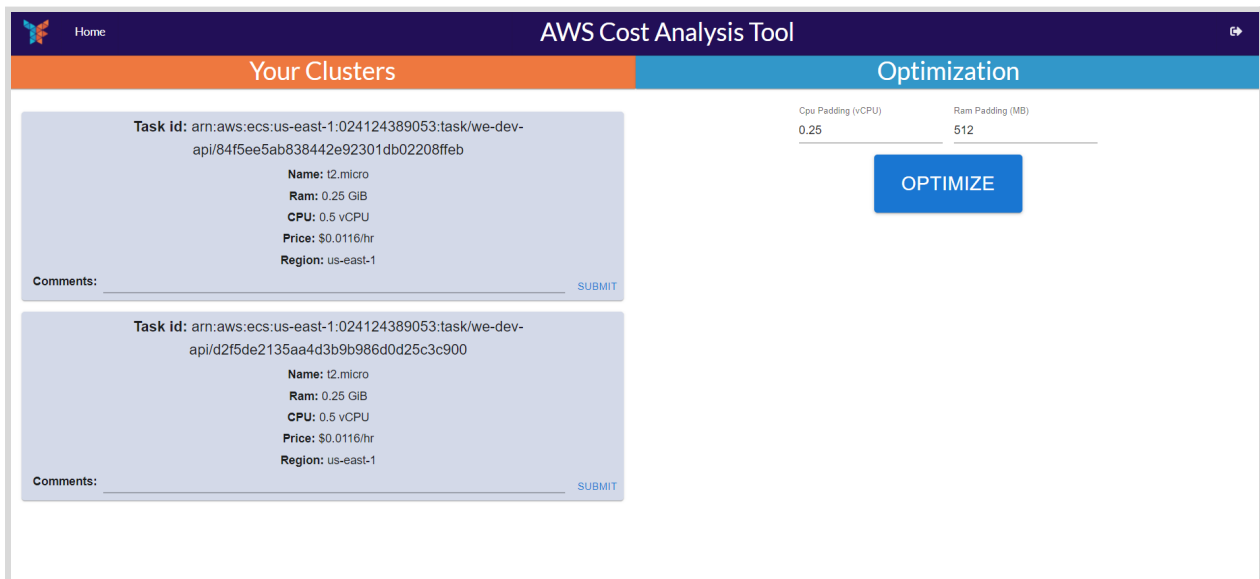


Figure ii: Optimization Page

## Future Work

There are many ways that a future MQP team or a team at 7Factor could take our project and extend it. The first way would be to improve the algorithm, which would allow it to be more efficient and take less time to run. If these customization options become very complex, a team could use machine learning to tackle the problem. Another way would be to improve the user interface of the frontend website, which would make the tool easier to use. With some modifications, the tool could be customer-facing, allowing 7Factor to handle a larger base of customers. This means that even more people can lower their cloud costs and simultaneously let 7Factor invest the money to create more tools that help customers with their DevOps issues. Lastly, a future team could implement visualizations that help users better understand how their clusters are configured, especially as clusters grow and become more complex.

- a. Improving algorithm
- b. Improving UI
- c. Improving functionality, giving more options for different requirements
- d. Making it more scalable
- e. Make able to be customer facing
- f. Implementing Machine Learning/Data Visualization → Originally chose Python for these reasons

# Introduction

---

In recent years, many companies have abandoned managing their own server farms and data centers, outsourcing this to cloud computing services such as Amazon Web Services or Microsoft Azure. One of the major reasons for this is that it lowers the cost of starting a company, as these cloud computing services tend to allow users to rent servers, with the cost depending on how many servers they rent and how strong the servers are. This means that users do not have to pay for thousands of dollars of equipment, and can instead just pay a monthly payment. However, these cloud computing costs can escalate as the business scales, eating into profit margins.

Our tool is intended to help lower user's cloud computing costs by identifying when servers are configured in an inefficient fashion, such as when a user uses too many servers or servers that are too strong. By identifying these problems, users can allocate less servers, meaning that they can cut down on how much they spend on cloud computing.

# Background

---

## About 7Factor

7Factor develops secure, stable, and scalable solutions for clients in many industries, including the financial sector, healthcare, aviation, and media. They are trusted by many respected brands as well as by startups and smaller companies alike. 7Factor builds cloud architectures, continuous delivery pipelines, and custom systems and applications. In addition to performing project work, 7Factor also offers advice contracts, to provide help navigating through complicated issues (7Factor).

## AWS APIs

In this project we were tasked by 7Factor to develop a tool that could gather information about some of their cloud infrastructure hosted on Amazon Web Services (AWS) and provide recommendations for potential cost-saving alterations to the configuration of this cloud infrastructure. 7Factor utilizes AWS's Elastic Compute Cloud service, otherwise known as EC2, in conjunction with AWS's Elastic Container Service (ECS). AWS EC2 provides a variety of different instance types for customers to choose from and assign tasks. Each of these instance types have their own specifications, which include differing amounts of memory, vCPUs (threads of a CPU core), processor types, and networking capabilities. The EC2 instances are then linked with containers created inside AWS ECS. In cloud computing, containers hold an individual set of applications/software that are necessary to run in any environment. However, they are not a complete virtual machine themselves. Instead, they rely on the operating system of the host, which in the case of ECS, is the EC2 virtual machine the instance is linked with.

As part of defining ECS containers, AWS requires that task definitions be set up. These task definitions specify minimum CPU and memory requirements in order for the task to run successfully. As a consequence, these tasks must be linked with EC2 instances that have enough physical hardware (CPU and memory) to accommodate the task's needs. This is where our tool comes into play. Our task was to make a tool to analyze any current ECS task configuration in a given EC2 cluster, in an attempt to optimize the types of instances being used in the cluster while still providing enough physical resources for the ECS containers. This can be achieved either by changing the currently used EC2 instance types for less powerful, cheaper types, or in some cases by combining multiple EC2 instances into one larger instance type. This is possible as multiple ECS task definitions may be assigned to a single EC2 instance, so long as the instance has the physical resources to accommodate them (Amazon Web Services, Inc.).

# Algorithm

We designed an algorithm that, given a certain set of inputs, can give the cheapest configuration of AWS instances for a particular set of tasks. This is similar to the knapsack algorithm, which is a common problem in computer science where someone has to maximize a value given particular constraints. We will present an analogy that will help better explain our algorithm.

## Buckets & Marbles Analogy

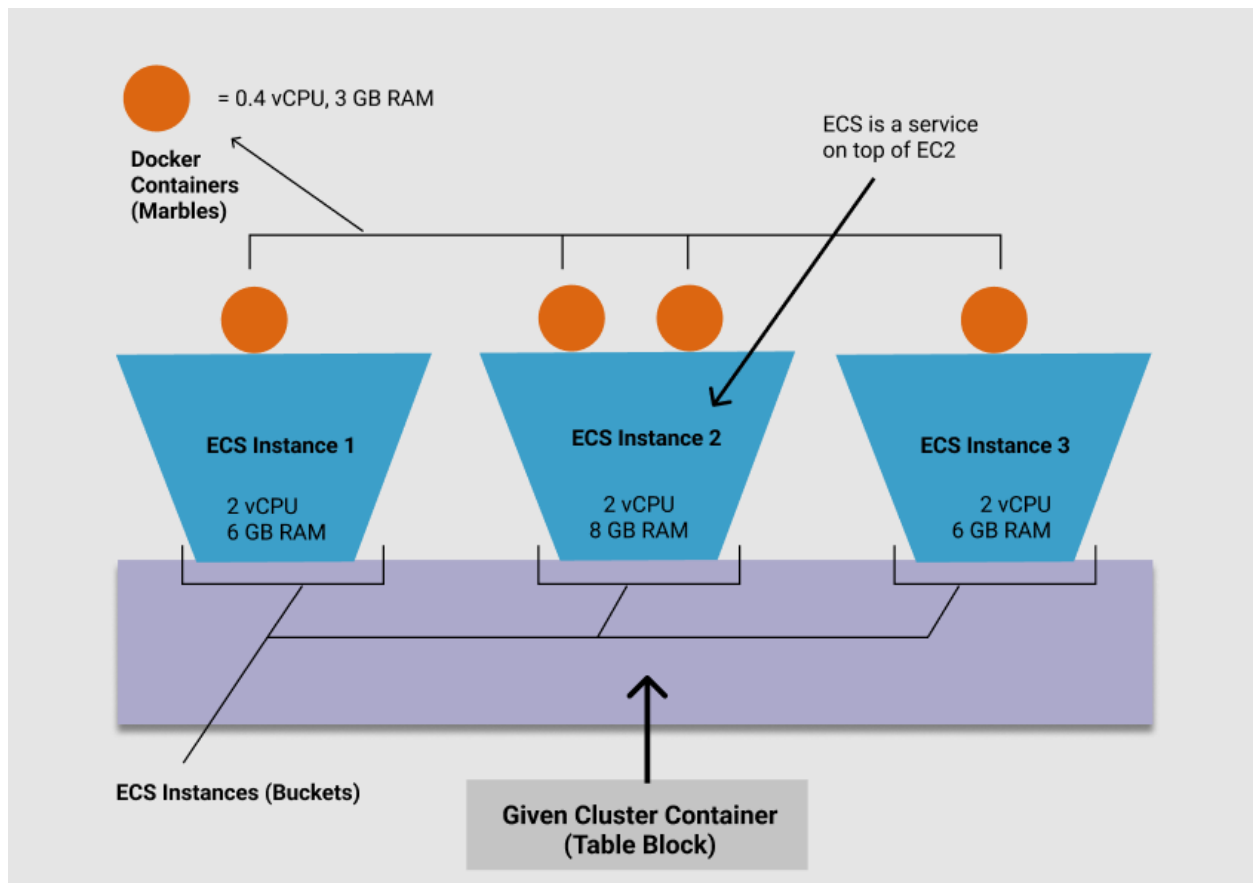


Figure 1: Buckets & Marbles Diagram

Here, we will represent the ECS cluster of machines as a table block. Each ECS instance, or individual EC2 virtual machine, within the ECS cluster would be represented by a bucket inside of the table block. If we have 10 EC2 virtual machines inside of our ECS cluster, we have 10 buckets. We also have our Docker containers, which will be represented by marbles. ECS would allow us to put marbles into buckets. Therefore, our problem comes down to this: how many marbles can we fit inside of our buckets?

The diagram above depicts our cluster with 3 ECS instances and 4 Docker containers. Using the buckets and marbles analogy, we have 3 buckets, the first having 2 vCPU and 6 GB of RAM, the second having 2 vCPU and 8 GB of RAM, and the third having 2 vCPU and 6 GB of RAM.<sup>1</sup> This means that in our cluster, the total amount of virtual CPU is 6 vCPU and 20 GB of RAM. We also have 4 marbles, each requiring 0.4 vCPUs and 3 GB of RAM.

This is where the problem that we are trying to solve becomes similar to the knapsack problem. In the knapsack problem, a thief carrying a backpack goes into a museum to steal valuable items. These valuable items each have a weight and a value assigned to them. The thief has to figure out the most valuable combination of items within the maximum weight capacity that the thief can carry. Our job is to find the best configuration of buckets that can fit all the marbles that we are given. In our example, we would be able to fit one marble in the first bucket, two marbles in the second bucket, and one marble in the third bucket.

---

<sup>1</sup> *Note: Even though each of the maximum vCPU requirements is 2 vCPU, in practice, we would need to consider the buffer that the operating system requires to run. That means the maximum vCPU for an ECS instance that has 2 vCPU would actually be around 1.5 vCPU.*

# Methodology

---

Our primary goal was to find the most efficient solution to the aforementioned problem. First, we wanted to choose the right style of application that would be best suited for this problem. We also made a goal to find the technologies that would be best suited for programming the frontend and backend of our application based on the strengths and shortcomings of each tool. Overall, we wanted to foster an environment of creativity and collaboration that would help us achieve our primary goal.

## Type of Platform

Choosing the best type of application required considering the technical requirements as well as the intended users. As a group, we decided to create our tool as a web application so that it could be used across multiple platforms and be easily available. In addition, the web app format also makes it easier to see the results of the tool compared to a command line tool.

## Python & FastAPI

Python is a popular general purpose language. Its user-friendly syntax and versatility made it a great language to consider, especially due to its prevalence “for [the] develop[ment of] websites and software, task automation, data analysis, and data visualization” (Coursera 2021). There are many tools and softwares that connect easily with Python. Even though Python programs run slower than Java, they are easier to develop, are on average 305 times shorter, and are object-oriented similar to Java (Python Software Foundation). We chose Python for this project because of its simplicity of use and easy connectivity with useful services. Python is a language frequently implemented in the backend of websites or applications, which involves the processing of data as well as the communications between databases, being able to manage URL routing, sending data back and forth from servers, and ensuring security (Coursera 2021). We also considered the directions that this project could take on in the future, such as using machine learning to calculate configurations of the AWS instances, for which Python would be well-suited. These reasons were why we decided that Python was the best fit language to develop the backend of our application.

One common Python framework that is used to make the backend of websites is FastAPI. There were many reasons we chose to use FastAPI over other backend Python frameworks such as Flask or Django, one being that FastAPI has a significantly easier learning curve, meaning more of us could work with it while investing less time into the learning process. This is due to it being more intuitive and having more features that can be autocompleted. Another benefit of FastAPI

is that it is faster than its competitors. One of the reasons for this is that FastAPI supports asynchronous processes, a feature that Python did not used to have. This allows it to run multiple processes concurrently instead of running them one after another, essentially meaning that any concurrent processes have minimal impact on the overall run time (FastAPI).

## Boto3 (Python AWS SDK)

Boto3 is the Amazon Web Services Software Development Kit, or AWS SDK, for Python that we chose to use for this project. It manages, creates, and configures the AWS services aspect of this project (Amazon Web Services, Inc). The two key Python packages that are included in the SDK are Botocore and Boto3. Botocore provides the low-level functionality shared between the Python SDK and the AWS CLI, or AWS Command Line Interface. Boto3 is the package that implements the Python SDK itself and is built on top of Botocore along with the AWS CLI . In order to receive a response from AWS, Botocore takes care of various components needed in an API request sent to AWS, which includes handling the session along with its credentials and configuration. Botocore also makes sure that all the operations that are contained in a particular service may be given permission to make API calls. It is responsible for serializing input parameters, signing requests, and deserializing response data into Python dictionaries which allows Python to interact with AWS. As a result, Botocore allows its users to not worry about all of the fundamental JSON specifications necessary to correspond with AWS APIs (Geller).

For this project, we used Boto3's ECS, EC2, and Pricing clients. The implementation of the API for this project required a good understanding of how AWS structures their ECS and EC2 services and how exactly that relates to 7Factor's problem description. We used Boto3's ECS client to obtain information about the clusters currently present in an account and the task definitions that are associated with each cluster. The information we collect from the task definition includes the minimum amount of CPU and memory required for the task to run successfully, the name of the EC2 instance type that task is currently utilizing, the unique ID number AWS assigns to the task, and the current hourly rate of that EC2 instance type. The pricing information is necessary in order to give an accurate number for the potential savings once we have identified alternative instance types.

This main call to the Boto3's ECS client by necessity also includes embedded calls to the EC2 client and the Pricing client. A call to the EC2 client is necessary to get the EC2 instance type currently being used, and a call to the Pricing client gets the cost of that instance type. Since Boto3 returns the data retrieved from the API calls as Python dictionaries, it's relatively easy to pass data between different parts of the API to gather all the information needed for our use cases.

Another heavily-used part of the integration with AWS was Boto3's Pricing API, which was used



to obtain information about all potential EC2 instance types. This information is passed directly to our algorithm to determine which configuration (or configurations) are optimal. 7Factor did not provide us with many restrictions on which types of instances to consider in our algorithm, so most EC2 instances are considered. 7Factor did specify that they were only interested in instances with processors with conventional x86-64 architecture, as opposed to ARM architecture. Additionally, they specified they weren't in need of any extra preinstalled software, as AWS provides customized versions of some instance types with SQL database software preinstalled. The Boto3 Pricing API has some filtering capabilities built-in, which helped us to only obtain instance results from the API which fit 7Factor's needs.

## Javascript/Typescript

Javascript and Typescript, a superset of Javascript, are considered the standard for frontend web development languages (Gardón 2021; Tutorials Point). While we considered using WebAssembly to develop the frontend, we decided against it since the complexity of our frontend was such that the potential performance increase from using WebAssembly was not worth the additional overhead. We also wanted to use a familiar frontend development framework, and there were already members on our team who were familiar with the Javascript framework React.

Node.js is a server side runtime environment that allows JavaScript to be used for the frontend web server. Using Node.js made it easier to produce our web pages and render them. The Node Package Manager (npm) made it easy to quickly install additional dependencies as the need arose.

## React/MUI (Material User Interface)

React is a framework for JavaScript designed to make it easier to create the visuals on a web page in order to build a user interface (Meta Platforms, Inc). React is designed to populate a web page quickly through using reusable components. Another library associated with React is MUI (Material User Interface). The MUI library contains common components like lists or sidebars, and is very modular, allowing for a more customized look. Given the prevalence of web pages in the project, the team felt that using React would be a good choice (Material-UI SAS). Additionally, the team felt that using MUI would increase the visual quality of each page. By using the React framework instead of raw JavaScript/HTML/CSS, the code would be easier to maintain by 7Factor or a future MQP team.

## Git

Git is a standard way to cooperate on large coding projects, and so we decided to use it as our version control system. When making many unrelated changes to the codebase at once, Git was able to keep each part of the project separate so that it could be tested independently. When the parts were ready to go, we were able to merge them into the final product. Git was also able to save previous states of the project so that they could be returned to if something went wrong. Git also allowed us to see what edits were made at what time, which helped find the source of problems.

## Docker

Docker is a tool that makes it simpler for multiple developers to work on a project in the same environment without having the same local configuration. Docker creates an environment called a container that meets the dependencies for a specific codebase. When a developer works on that codebase, they can run it in the Docker container to ensure that the environment uses the same standards as everyone else. There are also many off-the-shelf Docker images that we used in order to speed up our development without having to write boilerplate code and to make sure our code is portable and reliable. Using Docker in this project ensured that all the parts ran in a uniform environment.

## API Calls: Auth0

For managing credentials and security, we used Auth0, a third party platform that 7Factor uses to manage a variety of their tools. By using Auth0 we were able to help secure our tool without having to worry about the variety of attacks that hackers use to access accounts and resources. The protection that Auth0 provides is twofold: one part helps secure our frontend React web pages, and the other part helps secure the endpoints that have access to various AWS resources and information about what clusters are being run.

In order to make sure only authorized users could use our back end, we first have the front end request an access token from Auth0 that corresponds to the logged-in user. Then, this token would get attached to the API call that gets sent to our backend, where the token would get decoded and then again checked with Auth0 to see if it was a valid token. Once it is verified, we check the user ID against our database, allowing the user to access only their information in a safe and secure way without their private credentials ever going through our system.

## Auth0 integration

For our frontend integration of Auth0, we have the login page and credentials managed by Auth0, where if a user tries to access a protected page, their credentials are verified first. If a user is not logged in, they are then redirected to Auth0's login page, where they can create an account or login. Once it is approved, they are redirected back to the page they were initially trying to access, now with the correct credentials.

## Database structure: Postgres

For our database, we used Postgres, a relational database that allows us to keep track of our data in a way that is consistent and secure. One of the reasons we chose a relational database like Postgres over a nosql database like MongoDB was because more of the members in the team were familiar with Postgres, as this would allow us to use it with less of a learning curve. Another reason we chose Postgres was due to its convenience. The template that we used to generate much of the boilerplate code for the website came with support for a Postgres Docker image. Lastly, the ACID (atomicity, consistency, isolation, and durability) properties that traditional databases come with suit our application well—we did not want to have inconsistent clusters or risk losing data from a non-durable database.

## Figma

For our UI/UX design components, we wanted to put together a simple, intuitive, and usable design. In order to accomplish this, we used a collaborative online interface design tool, Figma, to brainstorm and create several mock-ups to represent initial ideas that would evolve into our final pages. First, we began creating an initial mockup (Figure 2) that represented the page for a current configuration. We also included a top navigation bar that would redirect to the home page, the client name/id page, the workloads page, and an optimizer/calculator page. On the left side, we had another navigation bar to allow the user to switch between the different configurations used for 7Factor's clients.

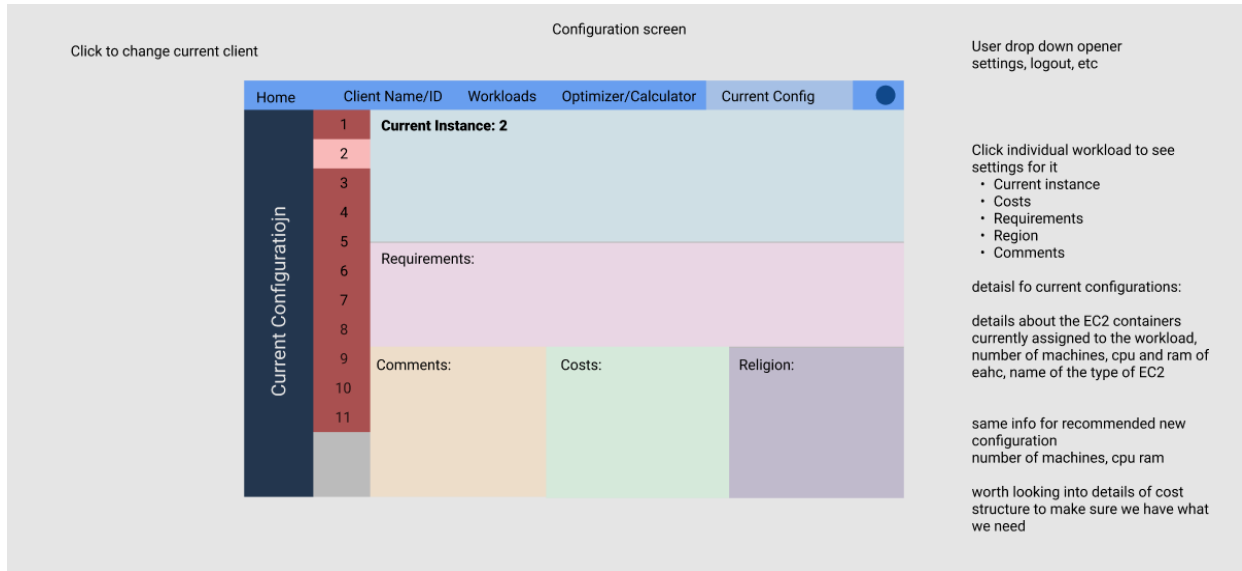


Figure 2. Current Configuration Page Figma Mockup

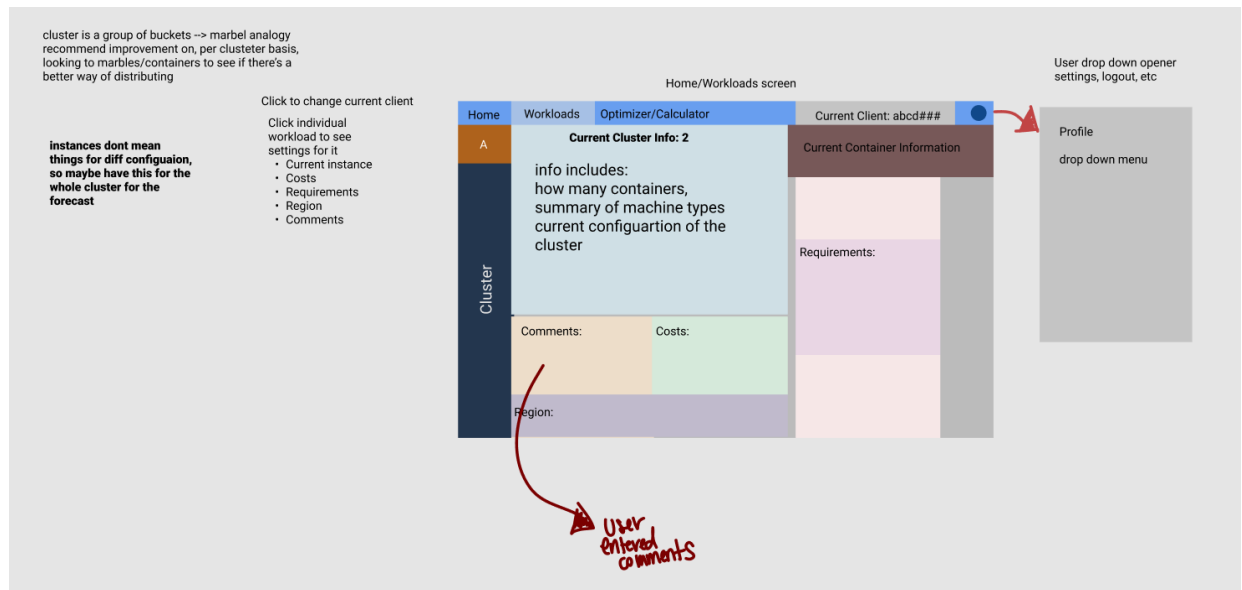


Figure 3: Workloads Page Figma Mockup

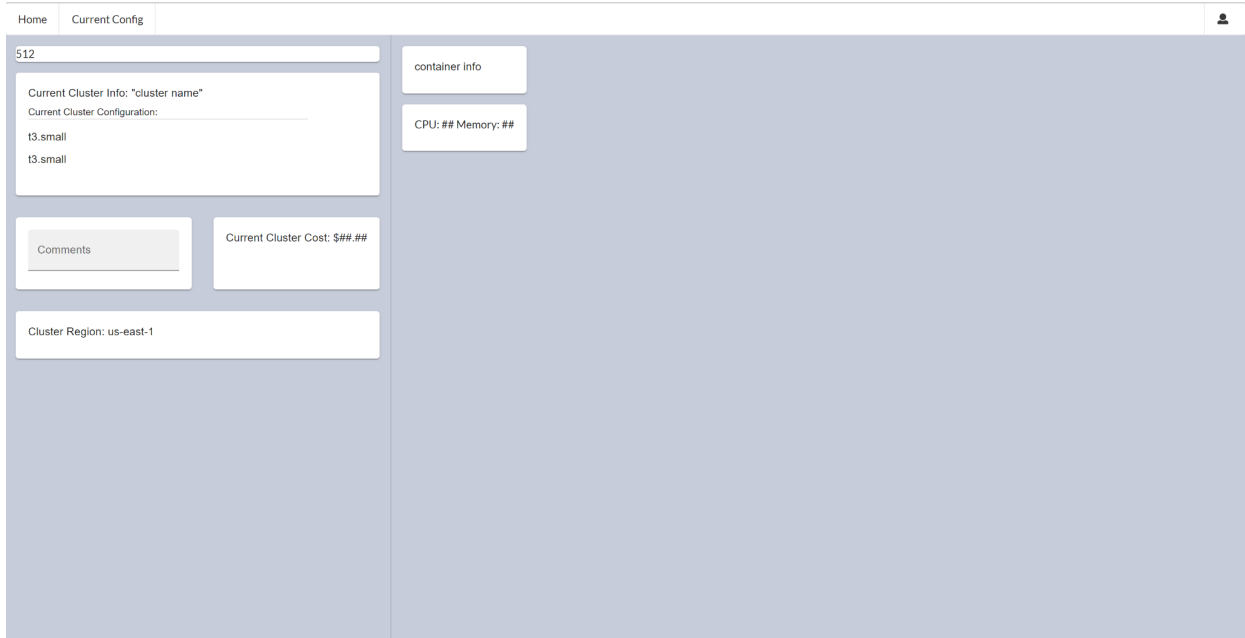


Figure 4: Scrapped Current Configuration/Workloads Page

This initial design idea was scrapped once we were able to determine and confirm with our 7Factor advisor the differences in the definitions between a workload, a cluster, and a configuration. With this information, we redesigned our mockup, Figure 3, to show the specific information on a currently selected cluster in the workloads page. We removed the bar that would have allowed the user to switch between clusters assigned to different 7Factor clients once we realized that we would not have access to this information. We eventually implemented and started on a version of this current configuration workloads page, which is shown in Figure 4. However, we scrapped this idea for the workloads page once we decided to combine its functionality with that of the home page.

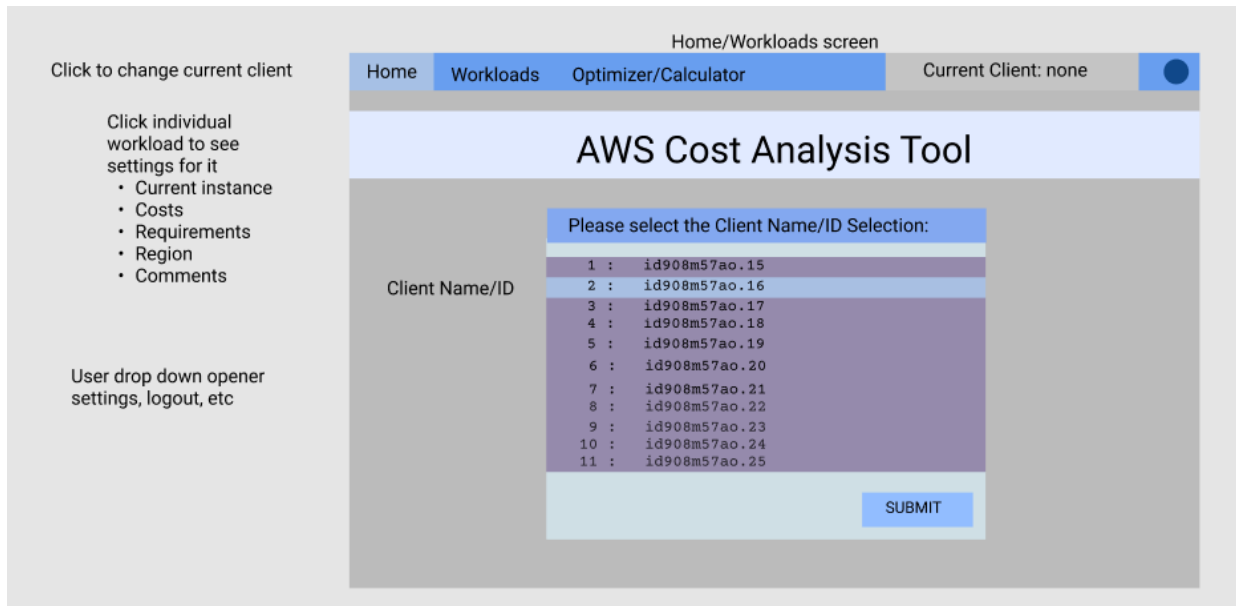


Figure 5: Home Page Figma Mockup

The initial mockup for this homepage, Figure 5, was created with the goal to allow users to select a given cluster and be redirected to its workload optimization page, which is shown in Figure 6. The decision to combine the functionality of our original designs for the home page and the workloads page was made to reduce the need for extra pages and to make this page more intuitive. This would allow users to see detailed information about a given cluster before being connected to its optimized configurations.

Here, we began with creating our cluster optimization page, shown in Figure 6. This page shows the recommendations made after the algorithm is run on a certain cluster, which would show the original cluster configuration and the proposed cluster configuration side-by-side. This would allow the user to see the changes between the original cluster information and the savings made.

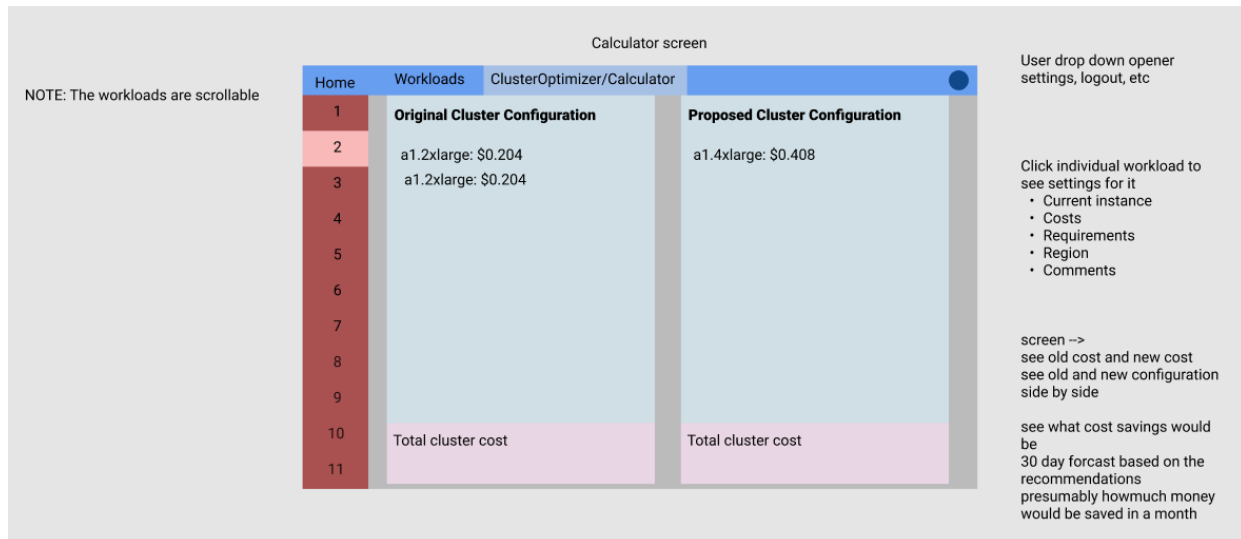


Figure 6: Optimization Page Figma Mockup

The goal of our algorithm was to take in a list of AWS tasks that need to be run and a list of AWS instance types that can be chosen from, and output a list of configurations that are the cheapest possible. By doing this, we are able to find the cheapest way that a company can organize their servers, letting them cut costs and re-invest the money to better grow the company.

In order to design our algorithm for this problem, we took inspiration from a similar problem called the knapsack problem. In the knapsack problem, the goal is that given a set of items, each with a specific weight and value, and a knapsack with a weight limit, we want to find the combination of items with the greatest value without going over the weight limit. The naive solution to this problem is to iterate over the set of items while calculating the total weight and reward of every combination, keeping track of the configurations that maximize reward.

However, this has a runtime of  $O(2^n)$ , making it unacceptable for most scenarios.

One popular way to make this problem more tractable is to use memoization, a process in which the problem is broken down into multiple smaller problems and the intermediate results are stored for later use. In this scenario, there is a two-dimensional grid where the columns represent the different capacities—zero through the capacity of the knapsack—and the rows represent the different items. Each cell holds the best possible configurations of items and the max value that it can hold with the items up to that point and that particular weight capacity. To solve the problem, the user fills out the first row representing the case in which there is only one item, and fills each cell with a 0 if the item cannot fit or the value of the first item if it can. On all subsequent item rows, the user chooses to fill the cell's value with either the value of the previous configuration with the same max capacity, or the sum of the new item's value with the total value of the cell that previously represented the total (max capacity - the new item's weight). From here, the greater of the two is chosen and put into the new cell. Once all of the

cells have been filled, the user has the max value that can be carried with a particular knapsack and items. This memoization allows the knapsack problem to be solved in  $O(n * j)$  with  $n$  being the number of items and  $j$  being the total capacity of the knapsack, meaning that it is in polynomial time and can be used on larger inputs without taking as long.

In our implementation we use memoization in order to make the run time of our algorithm more tractable. We do this by creating a dictionary that stores intermediate results as the key and a tuple of the cheapest configuration and price as the value. By doing this we are able to cut down on redundant calculations, speeding up our algorithm and making it more usable on larger sets of AWS tasks and AWS instance types.

## Unit Testing (for Backend Algorithm)

In order to test our backend, we set up many unit tests in order to ensure that our program works as intended and also to try to prevent any unexpected errors from occurring once the program is deployed and being used. We did this by setting up unit tests for our various functions and then by testing with Pytest we are able to see our code coverage.

## Project Set-Up

In order to set up our project, we used a template developed by the creator of FastAPI, Sebastián Ramírez, which was made up of some Docker images and general boilerplate code that a full stack project would need. We then replaced the Vue frontend with React. We also updated the Docker image for the database to handle location data, in case we or future projects needed to be able to store the location data of AWS regions. We also added the Boto credentials that we need in order to access the information about clusters on AWS to our Docker image. Lastly, we installed the necessary software and tools such as npm for managing frontend libraries and Docker to run our Docker images.

Please refer to the Setup Supplementary material in the Appendix for further details on the project set-up.

## Workflow

### Team Organization

As a team, we wanted to make sure that our workflow was efficient and productive, so we established a few methods in order to accomplish this goal. First, we organized our team by splitting into frontend and backend teams. We then used the When2meet survey tool to



coordinate each team member’s weekly availability and established a daily meeting schedule per school term. When2meet was also used to establish weekly meetings with our project advisor. Furthermore, we created a shared Google Drive folder that allowed us to store project-related content. We also created a Discord server to facilitate open communication in an organized manner. Last, we incorporated a hybrid workflow model to accommodate for obstacles, such as scheduling conflicts, COVID-19 restrictions, and unpredictable weather circumstances.

## Project Progress

In order to keep track of our progress, we created and mapped out an overall project timeline that included goals that needed to be met by the end of each term. These milestones are listed in the table below.

School Term	Milestones
A Term	<ol style="list-style-type: none"> <li>1. Complete the main broad project plan.</li> <li>2. Begin preliminary project work.</li> </ol>
B Term	<ol style="list-style-type: none"> <li>1. Complete the interface prototype.</li> <li>2. Complete the bulk of base implementation.</li> <li>3. Continue improving the algorithm’s efficiency.</li> <li>4. Begin work on writing the project report.</li> </ol>
C Term	<ol style="list-style-type: none"> <li>1. Polish the main project to a presentable state.</li> <li>2. Complete the final report.</li> <li>3. Implement stretch goals identified by 7Factor if possible.</li> </ol>

Figure 7: Workflow Table

### A Term

Since these were the early days of our project, our main milestones for A term were to come up with the big picture plan for our project and to begin preliminary work. In order to achieve this, we began with a meeting with 7Factor in order to discuss broad project objectives and ask for recommendations on topics we needed to research. With this guidance, we then started on the project research, planning, and direction by exploring the AWS Boto API to investigate how data is provided. Together, we set up our project GitHub and established the backend tools and Docker images that we used. Meanwhile, we started on a Figma UI prototype design based on the research we were carrying out. This course of action allowed us to build a strong foundation for the main implementation of our project.

## B Term

Our main milestones in B term were completing the interface prototyping, completing the bulk of base implementation, improving the algorithm's efficiency, and starting on the project report. In pursuit of this, we first continued to work on and improve the Figma interface prototyping while consulting with 7Factor for input and feedback. We then began implementing the minimum requirements that 7Factor had set out for our project. This involved creating a working version of the algorithm to analyze one ECS cluster and come up with the recommendation and basic cost projection, interfacing with the AWS API to pull necessary data for one cluster, and completing the basic Auth0 implementation. After the working version of the algorithm was established, we then worked towards improving the algorithm's efficiency by the end of B term. Once a decent portion of our project was completed, we began to write the background research portion of the project report and started planning out future sections to write based on previous MQP reports.

## C Term

Moving into the final term of our project, our main milestones were polishing our main project, finishing our report writing, and fulfilling as many stretch goals as possible. Therefore, to make our project presentable, we made sure that all of the functionality of the optimization page and the algorithm instance handling was completed. Afterwards, we addressed the stability issues by making sure there were no React errors or any other errors that would break our program. We also walked through our UI in order to make sure that the look and feel of it was neat and consistent. We went back and refined parts where necessary. Meanwhile, we assigned each other different sections of the report to write and revised them as a team by reading each section aloud. In order to keep track of our writing and hold ourselves accountable, we set strict deadlines for completing each section and ensured that there were multiple sets of revisions sent to our project advisor, Professor Cuneo, for feedback. These were the stretch goals we planned:

1. Get algorithm as efficient as we can, if possible and not already achieved in B term
2. Adjustable padding for the CPU and RAM, ensuring that enough system resources are reserved for the host operating system on the EC2 virtual machine and not used for tasks.
3. Provide nice interface for portraying "reasoning" behind cost savings
4. Ability to "bulk" analyze multiple clusters
5. Visualizing any unused CPU/RAM resources

Out of all of these goals, we were able to implement stretch goals 1 and 2. The other goals are discussed in more detail in the future work section.

# Obstacles

---

Throughout the course of working on this project, we encountered several challenging aspects. Most of these challenges arose mainly because we as a group were unfamiliar with the technologies involved, and some of these technologies involved a steep learning curve. Working together as a group to do the necessary research, ask questions, and experiment with different solutions helped us ensure that these obstacles were not a permanent barrier and didn't impact the final result of our project.

## Amazon Web Services Structure

One of the largest obstacles we had was right at the start of the project: understanding exactly what the problem definition from 7Factor meant in the context of AWS's many cloud services. For a group of people relatively unfamiliar with cloud computing, AWS can be daunting to understand at first. Jeremy from 7Factor did a good job of explaining how the EC2 instances related to the ECS clusters and task definitions; however, even after hearing his explanation and doing some of our own research, there was still some uncertainty in the group about how all the pieces fit together. This was evident in our initial mockups of the user interface, as they contained some elements that reflected misunderstandings about how tasks related to the clusters and EC2 instances, and they also included information that we did not have access to. As we began to experiment with the features of the AWS API to see what kind of information was available, we were able to ask the right questions and come to a clearer understanding about what information we needed from the API and the problem definition as a whole. In addition, Jeremy was able to give us access to the AWS web-based console, which helped us get a better sense visually of how tasks, clusters, and instances were all related to each other.

## Amazon Web Services API

Another challenge we faced was learning how to use AWS's API services and developing the correct calls to get all the information we needed. In particular, the structure of the Pricing API made it rather difficult to develop the correct calls, as that API returns a complex JSON data structure with many levels. In order to get a piece of information such as the on-demand hourly cost for an instance type, it was necessary to step through all these levels to get a clear understanding of how they all fit together. In addition, some parts of this JSON data structure contained lists of AWS ID strings we had no way of knowing prior to making the API call. We had to develop a way to step into the first element of a JSON list without knowing the name of that element.

In addition, the Pricing API comes with some built-in filtering capabilities in order to curate the results, so only instances with certain attributes are returned. However, we encountered multiple obstacles with this filtering functionality. EC2 instance types can have many SKUs (stock-keeping units), with each SKU representing a variation of the instance type based on the operating system, preinstalled software, and pricing model. We discovered that every single possible SKU of a particular AWS product has data which may be returned through the Pricing API. It took us several attempts to identify a set of filters that gave us only one result per instance type while also not completely filtering out some instance types. Some of these filtering challenges came down to less-than-ideal design choices made by AWS when loading data into the API. For example, we had an initial filter to get instances where `'processorArchitecture'='64-bit'`; however we discovered that some instance types had the attribute `'processorArchitecture'='64-bit or 32-bit'`. Because the API matches based on string equality, any instance with the second attribute was filtered out even though it did have a 64-bit processor option.

Furthermore, we had to develop one “manual” filter outside of the API call to provide additional filtration the API couldn't provide. One of 7Factor's requests was that no instance types with ARM processors be included in the results; however the `'processorArchitecture'` attribute could be set to `'64-bit'` regardless of whether the processor architecture was x86-64 or 64-bit ARM, and no additional attribute was present within the Pricing API to differentiate between the two. As a consequence, we have an additional condition that checks for whether the processor model has the substring “Intel” or “AMD” in it, which successfully filters out instances with Amazon's own ARM-based chips.

## Auth0

A unique challenge we faced was setting up Auth0 integration into both the frontend and backend of the system. One of 7Factor's requests was to set up the credentials and login system to use Auth0 in order to allow users at 7Factor to use it seamlessly with other services they have that use Auth0. In order to implement this, we used a certain credential library that Auth0 recommended. However, our boilerplate code had a similar library with the same name but different functionality as the one that Auth0 recommended. It was difficult to debug and manage the code until we discovered this.

## Web Development - API Endpoints

Another challenging aspect of the project was developing the frontend to connect to the backend through the use of endpoints we set up for ourselves. While some of us on the team had some prior web development experience, for the rest of us, it was a new experience working with

FastAPI to link a Python backend to a JavaScript frontend. Fortunately, we were able to develop a style of API call that works well. We made an asynchronous call to an API endpoint accompanied with an Auth0 token, then used a React state to store the returned data to build elements on the page.

## Docker

While the use of a Docker image to run our backend server does a good job of ensuring that the server may be set up and run easily in different environments, we did run into some challenges getting the configuration of the Docker set up in a way that consistently worked for everyone involved with the project. In particular, while working on the project, we ran into issues whenever we tried to check out a new copy of the project from our GitHub repository. Without modifications to a certain shell script file, the Docker image wouldn't start properly. As it turns out, the root cause of the issue was Git substituting the incorrect type of line feed character into the shell script file. Once we knew to adjust this file to use regular LF (line feed) endings instead of CRLF (carriage return line feed) endings, we were able to work around this problem (YouTube, 2018).

# Results

---

## Meeting 7Factor's Objectives

At the beginning of the project, we met up with 7Factor to receive a briefing on AWS and to discuss the project objectives. Throughout the project, we continued to meet with 7Factor through Jeremy's office hours to receive further guidance on our progress. What follows is the list of objectives 7Factor presented to us and a description of our success in meeting those objectives.

**Build out AWS cost optimization software designed to look at the current capacities of EC2 instances running versus their assigned statistics and label them “over-provisioned” or “under-provisioned”.**

We built a web app with a backend that can be deployed from a Docker container. Our tool presents the user with the option to select any ECS cluster and run our optimization algorithm on it. Once the algorithm has completed running, it presents the user with the most cost-effective cluster configuration, which will correct for any situation where a cluster is over-provisioned or under-provisioned. The fact that the software is a web app allows optimal flexibility for deployment and device compatibility.

**Use the AWS API to retrieve metrics about currently running ECS instances on a per-cluster basis, look at all tasks deployed to a particular cluster, and assess their currently assigned CPU and memory limits.**

Our tool uses the AWS Boto3 API to retrieve the specifications of the EC2 instance types being used as part of an ECS cluster. This information includes the amount of memory, the number of vCPUs, and the hourly on-demand pricing for the instance type. These values are used within our algorithm to help determine if the cluster is under-provisioned or over-provisioned. The Boto3 API is also used to retrieve information about the ECS task definitions for a cluster. This includes the CPU and memory limits called for by the task definitions. This information is provided to the algorithm in order to determine which cluster configurations are valid for the given set of tasks.

**Design an algorithm that can label a cluster as over-provisioned or under-provisioned depending on the current cluster configuration and task definition.**

The algorithm we developed, built using Python, compares the current task and cluster configurations within an ECS cluster to potential alternative cluster configurations. It analyzes these alternative configurations and identifies the best configuration based on the on-demand hourly costs. If more than one configuration is tied for lowest cost, they're all presented on the web app.

**Provide an estimated cost savings based on the algorithm and a 30 day forecast**

Based on the difference in on-demand hourly rates between the current cluster configuration and the proposed configuration, our tool provides a savings estimate based on 30 days of usage.

## User Experience

### 1. Design Components

When we designed our web application, we wanted our UI to be simple and intuitive. We also wanted our UI aesthetics to match 7Factor's company branding. We believed that this design choice would further solidify to users that the application we created was owned by 7Factor and was intended for its internal use. Therefore, we implemented design elements that drew inspiration from 7Factor's company website, which is shown in the Figure below.

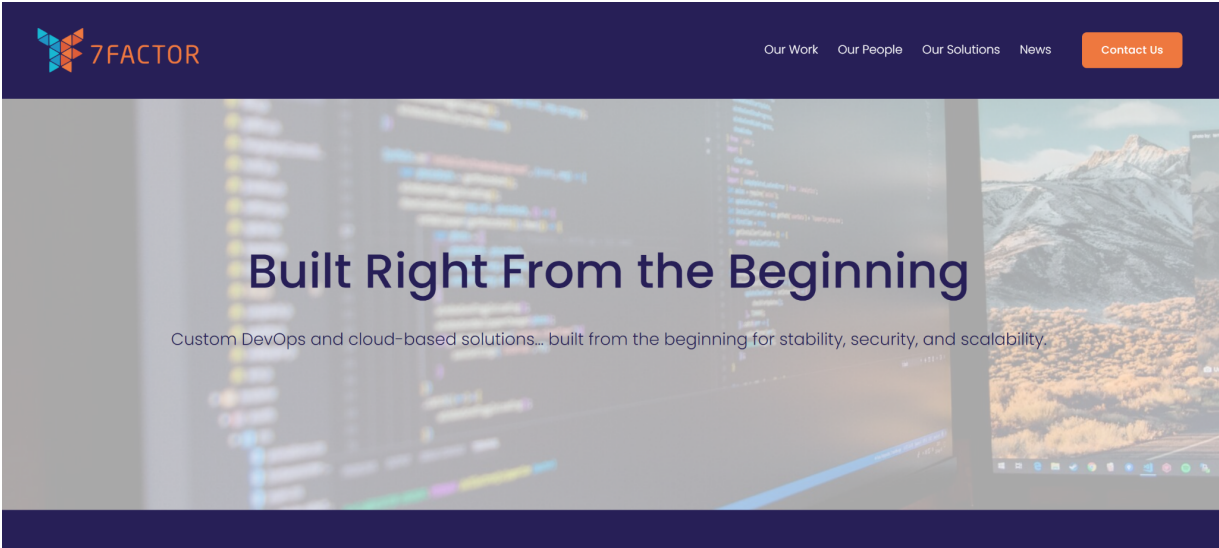


Figure 8: 7Factor's Company Website

This UI sample from our application, pictured below, conveys how we applied the inspiration from these aesthetics.

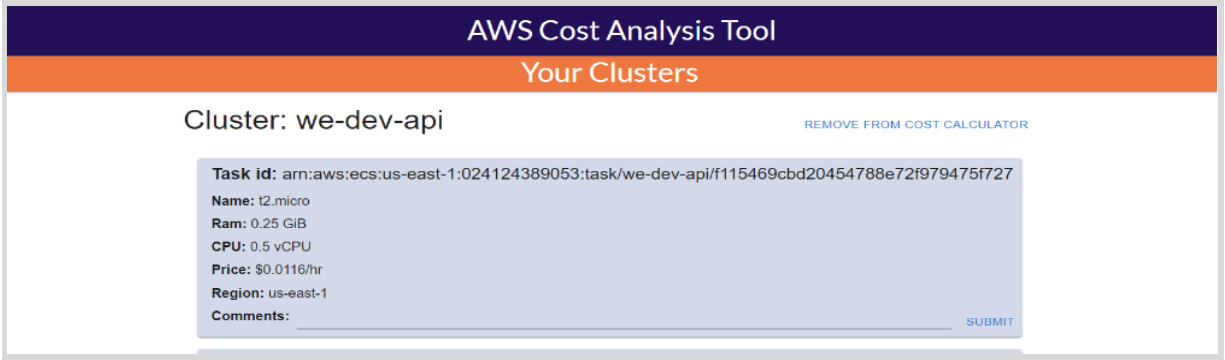


Figure 9: UI Sample

The current color scheme of our application are as follows:



## 2. Login Page

When a user first opens the site, they are prompted to log in using Auth0. Auth0 handles all of the accounts and user authorization for the project and keeps user data secure. After logging in, the user would be directed to the home page.

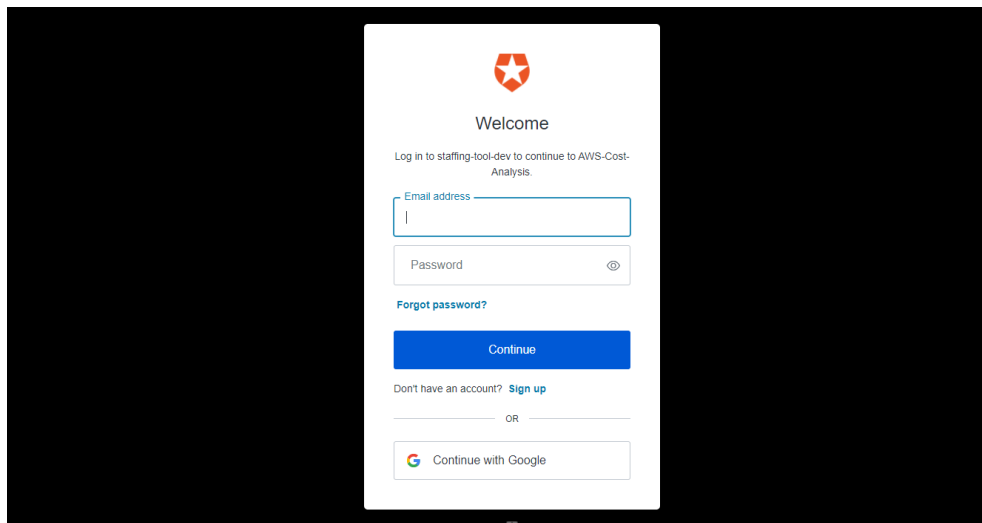


Figure 10: Auth0 Login Page



### 3. Home Page

The home page of our project showcases a list of clusters the user can choose from. It includes a submission field for users to add the names of their cluster to be put through an optimization algorithm. The user can also reach the home page by clicking either on the “Home” button at the top left of the toolbar or the “AWS Cost Analysis Tool” button in the center of the toolbar.



Figure 11: Home Button Icons

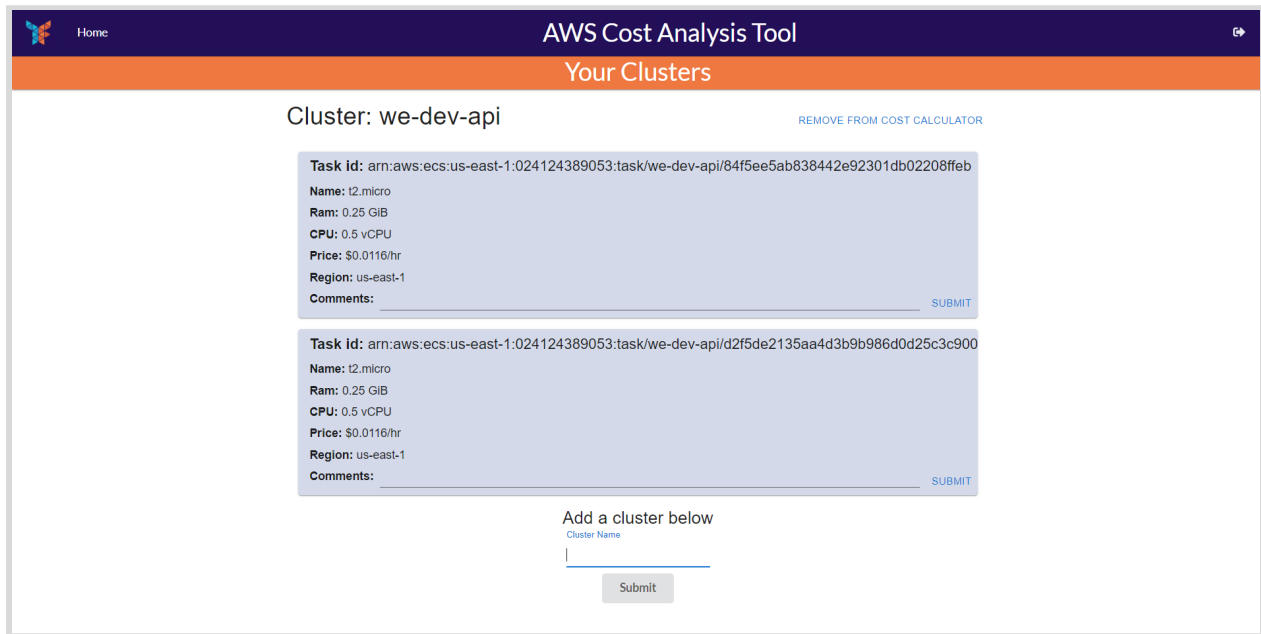


Figure 12: Home Page

Each cluster includes a task ID, the name of the cluster, its RAM and CPU information, its price, and any of the additional comments that are associated with the given cluster. Figure 8 shows the following cluster information for the “we-dev-api” cluster.

Cluster: we-dev-api [REMOVE FROM COST CALCULATOR](#)

**Task id:** arn:aws:ecs:us-east-1:024124389053:task/we-dev-api/f115469cbd20454788e72f979475f727

**Name:** t2.micro

**Ram:** 0.25 GiB

**CPU:** 0.5 vCPU

**Price:** \$0.0116/hr

**Region:** us-east-1

**Comments:**  [SUBMIT](#)

Figure 13: Home Page

When a user decides to add a new cluster to their list of current clusters, they can type the name of the cluster into this submission field. In our example, we’re going to type the name “7f-dev” to add the 7f-dev cluster into our list of clusters.

### Add a cluster below

[Cluster Name](#)

7f-dev|

---

Submit

Figure 14: Cluster Information

Upon adding a new cluster, the user would be able to see the cluster in their list. As seen in Figure 8, the “7f-dev” cluster is now pictured in the list of clusters.

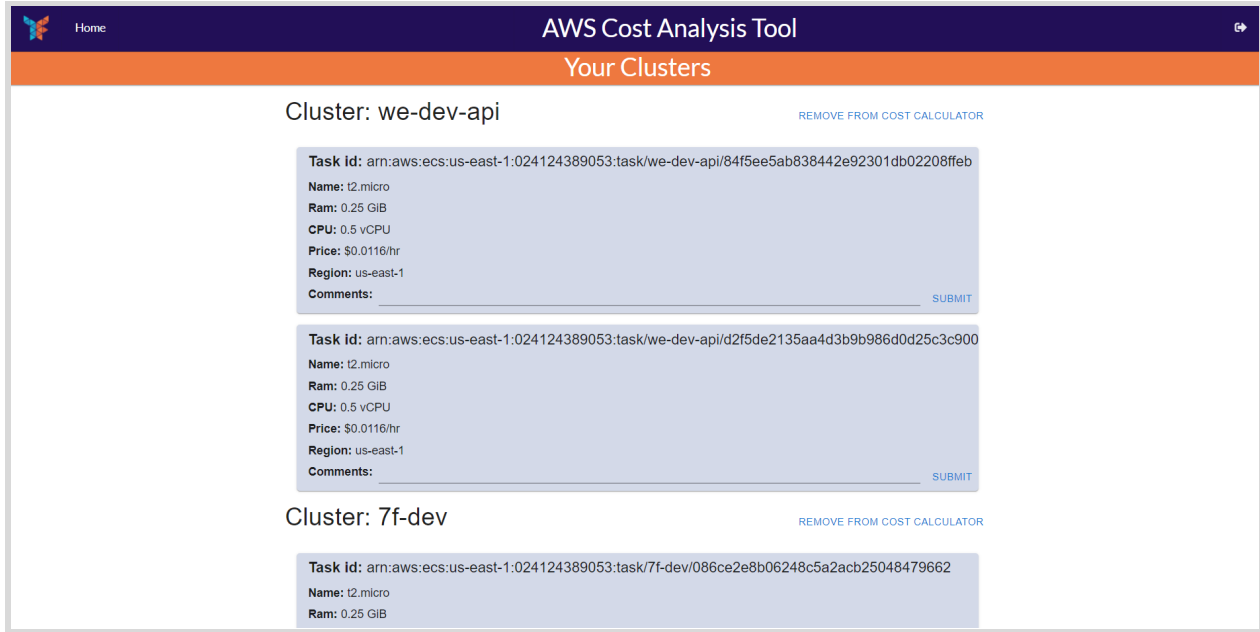


Figure 15: Home Page with Newly Added Cluster

To remove a cluster from the list, the user may click this “Remove from cost calculator” icon at the top right corner of the cluster list.

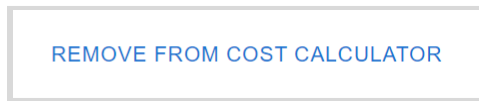


Figure 16: How to Remove a Cluster

#### 4. Optimization Page

Once the user clicks on the task id of their desired cluster, this field will turn blue, and they will be redirected to that given cluster’s optimization page.



Figure 17: How to Reach the Optimization Page

The optimization page shows the tasks belonging to a specific cluster on the left side and the optimization details on the right side. The optimization uses an algorithm to assess the needs of the tasks in the given cluster and then determine the most cost effective configuration that can accommodate those tasks.

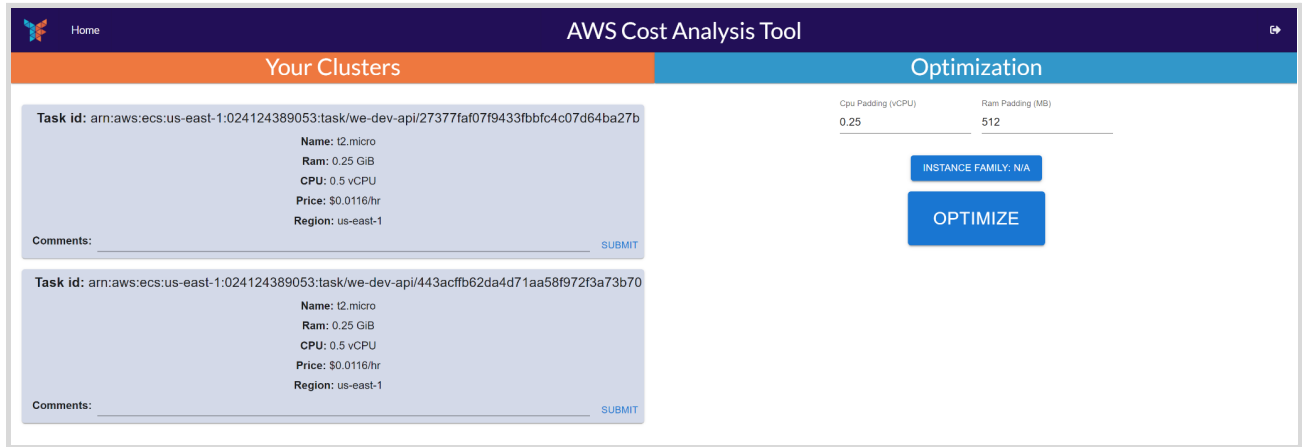


Figure 18: Optimization Page

To find out the most efficient configuration of the selected cluster, the user would click the blue optimization button displayed under the CPU and RAM paddings. The user would also be able to adjust the CPU and RAM paddings by either manually typing in the fields or using their designated adjustment arrows. Additionally, an instance family may be selected, limiting the scope of the considered EC2 instances to a particular instance family if desired. If the instance family is left as “N/A”, all instance types are considered by the algorithm.

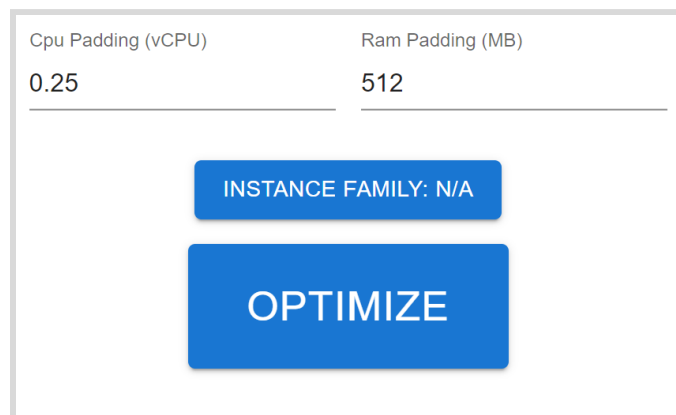


Figure 19: How to Adjust CPU/RAM Paddings and Instance Family

After the “Optimize” button is clicked, the total savings per month of the selected cluster would be displayed to the screen. Here, the user would be able to run through the various configurations that are also available to scroll through under the instance type card.

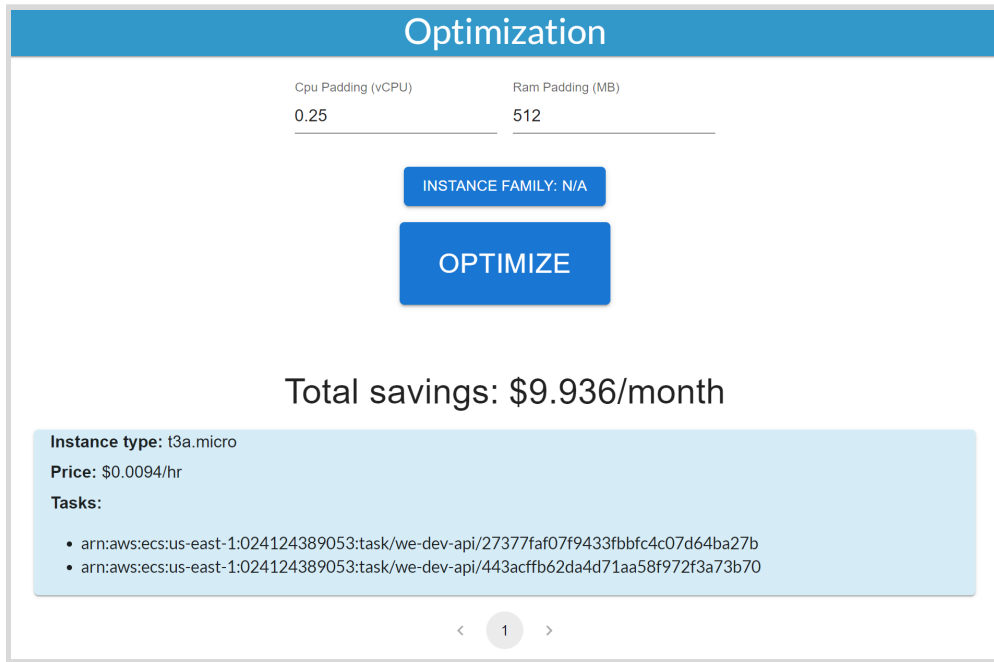


Figure 20: Total Savings Display

Figure 21 gives a complete view of the optimization page after the optimize button is clicked.

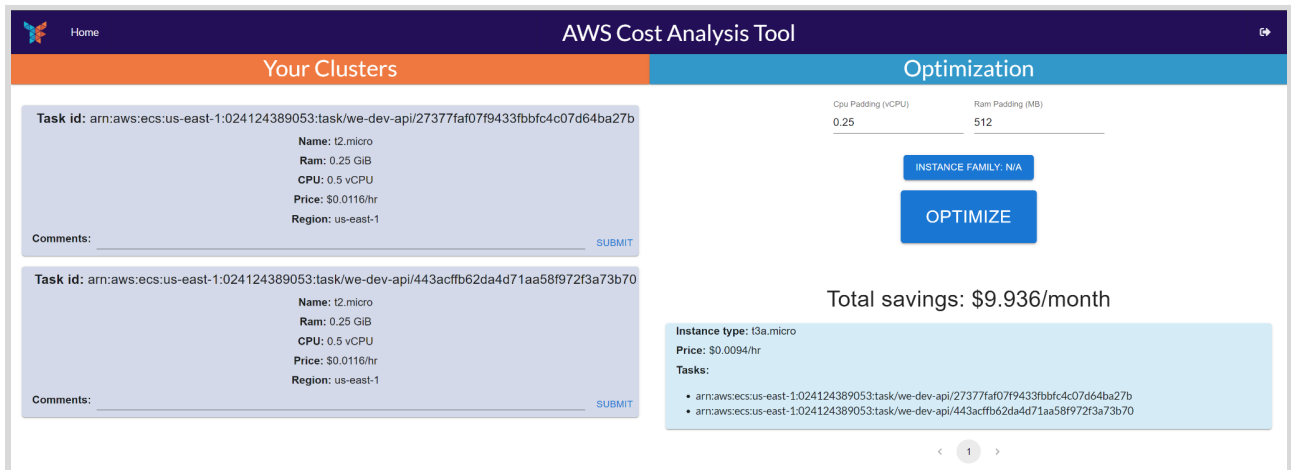


Figure 21: Optimization Page: Showing Total Savings

## 5. Settings Page

Included in the code for our project is a settings page. This is where a user can change the settings and log out when they are done using the software. This page is currently not being included in the final accessible view of our project, because the current implementation of our

project has fixed settings. However, this page may be further built upon in future iterations of the project if it is passed down.

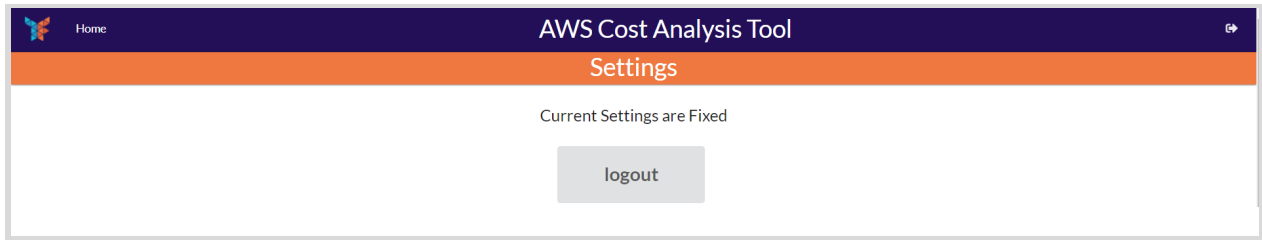


Figure 22: Settings Page

# Future Work

---

There are many ways that a future MQP team or a team at 7Factor could take our project and extend it. Below is a list of some possibilities:

- Future teams could add additional optimizations to the algorithm, which would allow it to be more efficient and take less time to run. Additionally, a team could improve the algorithm by adding new features, such as allowing some tasks to only be on certain instance types, making it so that instances must be together, or including some other customization features that a user may want.
- If the algorithm customization options become very complex, a team could incorporate machine learning. This involves getting a quick approximation for the cheapest configuration instead of finding the true cheapest configuration, something that may take a while and may not be a reasonable thing to wait for. This stretch goal influenced our choice of Python as our backend programming language to better facilitate this possibility.
- The user interface of the frontend website could be improved. This could be done in many ways, such as user testing in order to see what parts of the layout are confusing, fixing the interface based on these issues, and repeating the process.
- Future teams could ensure that the frontend website is accessible for everyone. This could ensure that it is able to be read by a screen reader or the colors work well for a colorblind user. This would allow for a greater number of people to be able to use the tool, making it more valuable.
- With some modifications, the tool could be customer-facing, meaning that 7Factor would be able to help a larger base of customers. This way, even more people could lower their cloud costs and simultaneously let 7Factor invest the money to create more tools that help customers with their DevOps issues.
- Future teams could work on the ability to run a bulk analysis on multiple clusters at once, as our tool only allows users to analyze one cluster at a time.
- A future team could implement visualizations that help users better understand how their clusters are configured, especially as clusters grow and become more complex. For example, these visualizations could give a better picture of virtual machines in clusters that have unused CPU and memory capacity. In addition, they could provide the ability to see the exact reasoning behind how an optimized cluster configuration would provide cost savings.

# Conclusion

---

We were tasked by 7Factor to develop a tool that analyzes AWS ECS clusters and recommends more cost-efficient configurations. We developed this tool as a flexible web-based application where users can specify a cluster and receive a recommendation for an optimized cluster configuration. The tool also provides a monthly savings projection based on the new hourly rate for the optimized configuration. Additionally, it allows the user to specify custom CPU and memory buffers to leave capacity for the VM operating system.

Our application uses Python with FastAPI for the backend and HTML/JS with React for the frontend. The Boto3 API is utilized to retrieve information about AWS ECS and EC2 instances and pricing. Additionally, Postgres is used for our backend database, and the entire application is packaged as a Docker container, ensuring easy setup and portability.



# References

---

- 7Factor. (2020). *7factor software devops and cloud-based solutions*. 7Factor Software DevOps and Cloud-Based Solutions. Retrieved February 17, 2022, from <https://www.7factor.io/>
- Amazon Web Services, Inc. (n.d). *AWS SDK for Python - Amazon Web Services (AWS)*. aws. Retrieved January 20, 2022, from <https://aws.amazon.com/sdk-for-python/>
- Amazon Web Services, Inc. (n.d). *What is Amazon EC2?*. aws. Retrieved February 2, 2022, from <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>
- Amazon Web Services, Inc. (n.d). *What is Amazon Elastic Container Service?*. aws. Retrieved February 2, 2022, from <https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-welcome.html>
- Coursera. (2021, September 22). *What is python used for? A beginner's guide*. Coursera. Retrieved January 20, 2022, from <https://www.coursera.org/articles/what-is-python-used-for-a-beginners-guide-to-using-python>
- Gardón, D. S. (2021, November 29). *Webassembly vs. JavaScript: The Complete Guide*. Snipcart. Retrieved January 21, 2022, from <https://snipcart.com/blog/webassembly-vs-javascript>
- Geller, A. (2021, July 19). *Explaining boto3: How to use any AWS service with python*. dashbird. Retrieved January 20, 2022, from <https://dashbird.io/blog/boto3-aws-python/>
- Material-UI SAS. (n.d). *The React Component Library You always wanted*. MUI. Retrieved February 2, 2022, from <https://mui.com/>
- Meta Platforms, Inc. *React – a JavaScript library for building user interfaces*. React – A JavaScript library for building user interfaces. Retrieved February 2, 2022, from <https://reactjs.org/>
- FastAPI. (n.d.). *FastAPI - Performance*. FastAPI. Retrieved February 17, 2022, from <https://fastapi.tiangolo.com/#performance>
- Python Software Foundation. (n.d). *Comparing python to other languages*. Python.org. Retrieved January 20, 2022, from <https://www.python.org/doc/essays/comparisons/>

Tutorials Point. (n.d). *TypeScript - Overview*. Tutorials Point Inc. Retrieved January 20, 2022, from [https://www.tutorialspoint.com/typescript/typescript\\_overview.htm](https://www.tutorialspoint.com/typescript/typescript_overview.htm)

YouTube. (2018). *What Is Docker? | What Is Docker And How It Works? | Docker Tutorial For Beginners | Simplilearn*. YouTube. Retrieved February 2, 2022, from <https://www.youtube.com/watch?v=rOTqprHv1YE>

# Glossary

---

1. **7Factor** — A software development consultant company specializing in creating devops solutions; our MQP Sponsor.
2. **AWS (Amazon Web Services)** — A broadly adopted, comprehensive global cloud computing platform that offers 200+ services and features from infrastructure technologies to emerging technologies.
3. **AWS CLI (AWS Command Line Interface)** — An open sourced unified tool that can be used to manage AWS services from a command line shell.
4. **Boto** — A Python AWS software development kit that manages, crates, and configures the AWS services aspect for this project using two of its key Python packages, Botocore and Boto3.
5. **Container** — In cloud computing, containers hold an individual set of applications/software that are necessary to run in any environment. However, they are not a complete virtual machine themselves, since they are dependent on the operating system of the host.
6. **Docker** — An open source software tool that makes it simpler for multiple developers to work on a project in the same environment without having the same local configuration. It facilitates the management and execution of Docker Images.
7. **Docker Image** — An immutable, read-only file encompassing the source code, libraries, dependencies, tools, and other files that are required in order for an application to run.
8. **EC2 (Elastic Compute Cloud)** — A scalable computing capacity that provides a variety of different instance types for customers to choose from, effectively virtual machines in which tasks can be assigned to by customers. Each of these instance types have their own specifications, which include differing amounts of memory, vCPUs (threads of a CPU core), processor types, and networking capabilities.
9. **ECS (Elastic Container Service)** — A fast and highly scalable cloud container management service that can be used to run, stop, and manage containers on a cluster.
10. **FastAPI** - A common Python framework that is used to make the backend of websites and supports asynchronous processes.
11. **Figma** — A collaborative online interface design tool often used for UI/UX design components.
12. **Material-UI** — A robust and customizable React UI library of foundational and advanced components that can be used to build a design system to develop React applications faster.
13. **React** — A declarative, component-based Javascript framework designed to make it easier to build user interfaces by creating the visuals and populating a web page quickly through using reusable components.

14. **Service** — In the AWS ECS context, a service is a configuration used to concurrently run and maintain a specified number of tasks that are in a cluster.
15. **Virtual Machine** — A virtual environment that uses software in place of physical hardware to function as a computer system with its own CPU, memory, and network interface.

# Appendix

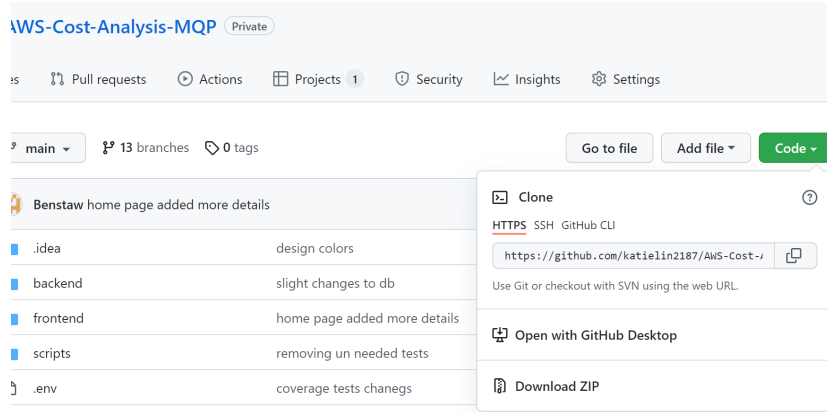
---

## Algorithm Pseudocode

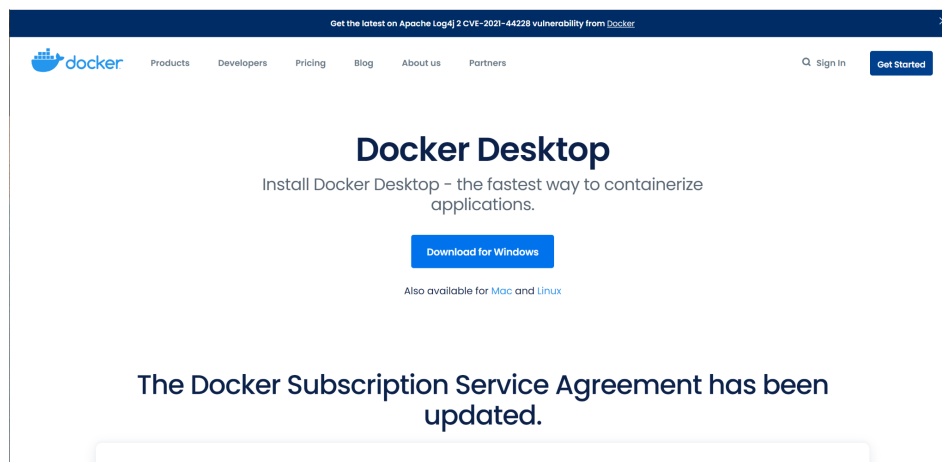
1. Give every task a unique id that is based on the ram and cpu requirements
2. Configuration = {}
3. Price = 0
4. Memoization = {}
5. If there are workloads to still process:
  - a. Remove workload w from the queue of workloads to be processed
  - b. C = {}
  - c. If w not in memoization:
    - i. For every instance type i that workload w fits in:
      1. Append new instance type {i, w} to configuration, return to 4, store results in c
  - d. For every AWS instance in configuration that w can be placed into:
    - i. Add w to the instance, return to 4, store results in c
  - e. Find cheapest configurations, return multiple in case of a tie
6. If there are no workloads to process:
  - a. For every instance in the configuration:
    - i. Add {for workload added to instance -> price} to memoization
  - b. Return current configuration and price

## Set-up for the AWS Cost Analysis Project

1. <https://github.com/tiangolo/full-stack-fastapi-postgresql>
2. Create the brand new repo
3. Download Git
4. Download GitHub Desktop
5. With the created repo, set up the repo for the other members, members can clone repo to local repository



6. Install Docker, <https://www.docker.com/products/docker-desktop>



7. Run “docker-compose up” on windows cmd prompt (or git terminal). Change the directory to the AWS-Cost-Analysis Project folder

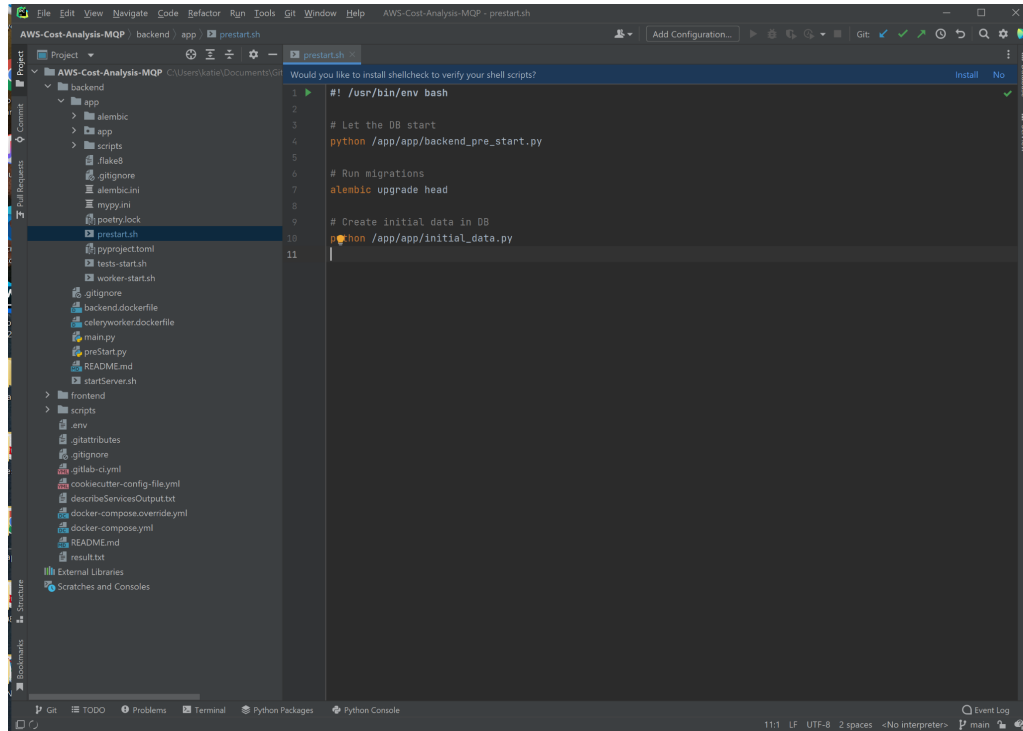
```
Command Prompt - docker-compose up

Directory of C:\Users\katie

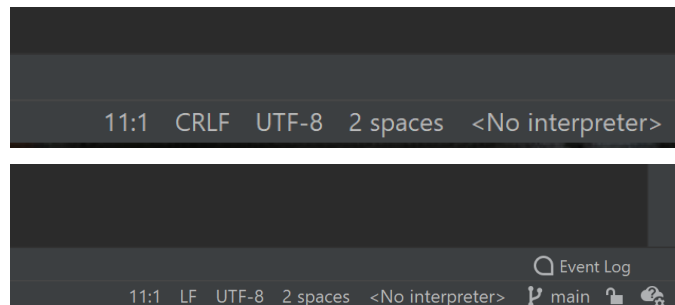
01/13/2022  11:20 AM  <DIR>      .
01/13/2022  11:20 AM  <DIR>      ..
01/13/2022  11:20 AM                28 .bash_history
01/07/2022  12:47 PM  <DIR>      .docker
01/05/2022  10:26 PM                207 .gitconfig
01/05/2022  10:04 PM  <DIR>      3D Objects
01/12/2022  11:52 AM  <DIR>      Contacts
01/13/2022  11:08 AM  <DIR>      Documents
01/13/2022  11:22 AM  <DIR>      Downloads
01/12/2022  03:10 PM  <DIR>      Favorites
01/12/2022  03:10 PM  <DIR>      Links
01/12/2022  11:52 AM  <DIR>      Music
01/13/2022  10:58 AM  <DIR>      OneDrive
01/12/2022  11:52 AM  <DIR>      Saved Games
01/12/2022  11:52 AM  <DIR>      Searches
01/12/2022  12:12 PM  <DIR>      Videos
                2 File(s)      235 bytes
                14 Dir(s)  940,689,448,960 bytes free

C:\Users\katie>cd Documents
C:\Users\katie\Documents>cd GitHub
C:\Users\katie\Documents\GitHub>cd AWS-Cost-Analysis-MQP
C:\Users\katie\Documents\GitHub\AWS-Cost-Analysis-MQP>docker-compose up
WARNING: The following deploy sub-keys are not supported and have been ignored: labels
WARNING: The following deploy sub-keys are not supported and have been ignored: labels
WARNING: The following deploy sub-keys are not supported and have been ignored: labels
WARNING: The following deploy sub-keys are not supported and have been ignored: labels
Creating network "aws-cost-analysis-mqp_default" with the default driver
Creating network "aws-cost-analysis-mqp_traefik-public" with the default driver
Creating volume "aws-cost-analysis-mqp_app-db-data" with default driver
Pulling queue (rabbitmq:3)...
```

8. If there is an issue with the backend being able to properly start up...  
In the prestart.sh file:

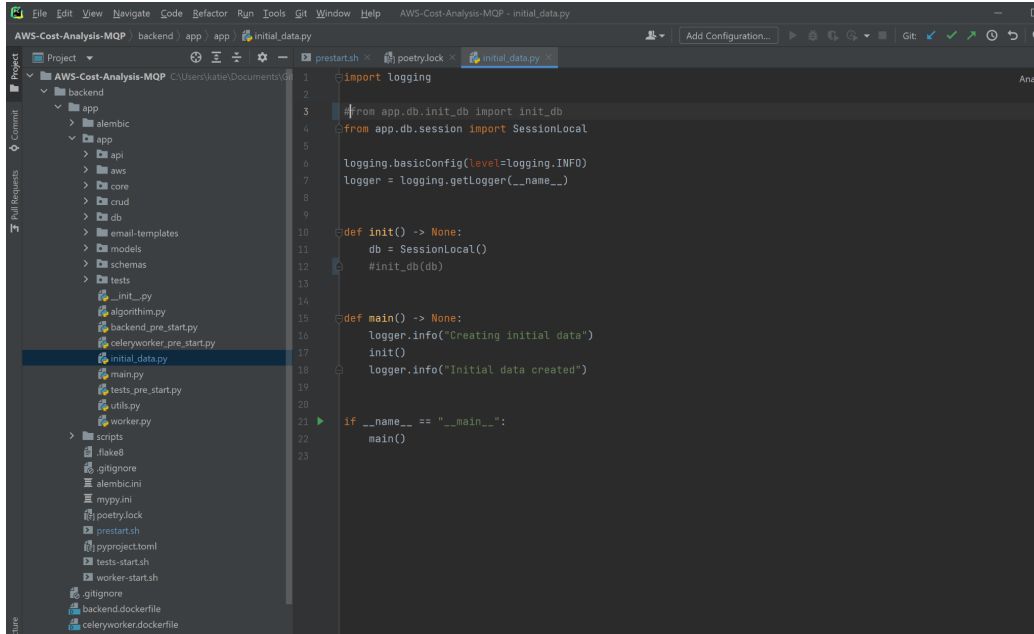


Change the option from CRLF → LF:



9. If backend still fails to compile, comment out lines #3 & #12





```
1 import logging
2
3 #from app.db.init_db import init_db
4 from app.db.session import SessionLocal
5
6 logging.basicConfig(level=logging.INFO)
7 logger = logging.getLogger(__name__)
8
9
10 def init() -> None:
11     db = SessionLocal()
12     #init_db(db)
13
14
15 def main() -> None:
16     logger.info("Creating initial data")
17     init()
18     logger.info("Initial data created")
19
20
21 if __name__ == "__main__":
22     main()
23
```

## 10. Install Boto 3 inside the docker image

- a. When running for the first time, run `docker-compose up` in the root directory to start the docker swarm.
- b. Then, run `docker-compose exec backend bash`. Then run the setup for the AWS CLI (<https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html>) with the commands:
  - i. `curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"`
  - ii. `unzip awscliv2.zip`
  - iii. `./aws/install`
- c. Once the AWS CLI is installed run
  - i. `aws configure`
- d. In the backend terminal, where you will be requested to supply the credentials.
- e. Currently for our region we are `us-east-1` and our output is JSON format.

```

Command Prompt - docker-compose exec backend bash
Microsoft Windows [Version 10.0.19043.1165]
(c) Microsoft Corporation. All rights reserved.

C:\Users\katie>cd Documents

C:\Users\katie\Documents>cd GitHub

C:\Users\katie\Documents\GitHub>cd AWS-Cost-Analysis-MQP

C:\Users\katie\Documents\GitHub\AWS-Cost-Analysis-MQP>docker-compose exec backend bash
WARNING: The following deploy sub-keys are not supported and have been ignored: labels
WARNING: The following deploy sub-keys are not supported and have been ignored: labels
WARNING: The following deploy sub-keys are not supported and have been ignored: labels
WARNING: The following deploy sub-keys are not supported and have been ignored: labels
root@c4c40fd18aef:/app#

```

```

creating: aws/dist/cryptography/hazmat/bindings/
inflating: aws/dist/cryptography/hazmat/bindings/_openssl.abi3.so
root@c4c40fd18aef:/app# ./aws/install
You can now run: /usr/local/bin/aws --version
root@c4c40fd18aef:/app# aws configure
AWS Access Key ID [None]: AKIAQLHPM626WNWUDMLG
AWS Secret Access Key [None]: vmNAV40wN3XLDfZH91GJttHh0qT0WAPwvimGBRj
Default region name [None]: us-east-1
Default output format [None]: json
root@c4c40fd18aef:/app#

```

## 11. Install node.js

The screenshot shows the Node.js website's 'Downloads' section. It highlights the 'LTS' (Recommended For Most Users) and 'Current' (Latest Features) versions. Under 'LTS', there are links for Windows Installer, macOS Installer, and Source Code. A table below lists the available binaries for each platform and architecture.

Platform	Architecture	File Name
Windows	32-bit	node-v16.13.2-x64.msi
	64-bit	node-v16.13.2-x64.msi
macOS	64-bit / ARM64	node-v16.13.2.pkg
	64-bit	node-v16.13.2.pkg
Linux	64-bit	node-v16.13.2.tar.gz
	ARM64	node-v16.13.2.tar.gz

## 12. Running front end:

- a. cd into frontend:
  - i. npm install (NOTE: first run)
  - ii. npm start

