7Factor AWS Cost Analysis Tool

A Major Qualifying 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 April 25th, 2024

By:

Cameron Goodrich

Joey Rozman

Nathaniel Sadlier

Oliver Shulman

Dov Ushman

Project Advisor:

Professor Joshua Cuneo

This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on the web without editorial or peer review.

Abstract

Since 2021, groups of WPI students have been working with 7Factor to develop a tool that can analyze AWS ECS clusters and optimize their infrastructure to minimize costs; this project continues upon that work. The sponsor of this project, 7Factor, is a cloud services and custom software company for clients seeking unique solutions in their software development. The prior iterations of this AWS cost analysis tool built a solid foundation that we utilized to identify what previous efforts have been successful and what could be further improved. Over the course of the 2023-2024 academic year, we worked with 7Factor to further optimize the tool and revamp the user interface.

Acknowledgements

We would like to thank the following people for their help throughout the entirety of this project:

Joshua Cuneo, Computer Science Professor at WPI - Project Advisor
Jeremy Duvall - CEO of 7Factor
Allen Brooks - Engineering Manager at 7Factor

Their help and guidance made this project possible and we could not have completed it without them.

Table of Contents

Abstract 2
Acknowledgements 3
Table of Contents 4
List of Figures 6
Executive Summary 7
Introduction 9
1. Background 10
1.1 7Factor 10
1.2 Amazon Web Services (AWS) 10
1.2.1 Introduction to AWS 10
1.2.2 Amazon EC2 System 10
1.2.3 Amazon Elastic Container Service (ECS) 11
1.3 Python Libraries 11
1.3.1 SQLAlchemy 11
1.3.2 FastAPI 11
1.3.3 Pydantic 12
1.3.4 Boto3 12
1.4 Technologies Used 12
1.4.1 Programming Languages 12
1.4.2 React 13
1.4.3 Bootstrap 13
1.4.4 Visual Studio Code (VSCode) 13
1.4.5 Version Control 13
1.4.6 Amazon Web Services (AWS) 14
1.4.7 Docker 14
2. Methodology 14
2.1 Front End Development 14
2.1.1 Study Previous Front End 14
2.1.2 Design a New Front End Look 17
2.1.3 Develop New Front End 18
2.2 Back End Development 21
2.2.1 Study Previous Back End 21
2.2.2 API Calls 22
2.2.3 Data Processing 22
2.2.4 AWS RDS Database 22
2.2.5 Optimization 23
2.3 Docker 23
2.3.1 Front End Use 23
2.3.2 Back End Use 23

2.3.3 Docker Compose.....24
3. Results.....24
3.1 Application.....24
3.2 Front End.....25
3.3 Back End.....26
4. Discussion.....27
5. Future Work.....28
5.1 Algorithm Analysis.....29
5.2 Front End Device Compatibility.....29
5.3 Database Expansion.....29
5.4 Automation of EC2 Instance Changes.....29
6. Conclusion.....30
References.....31
Glossary.....32
Appendix.....33
Development Environment Setup.....33
How to Use.....33

List of Figures

1. Previous Team's Front End
2. Figma Designs
3. CPU Utilization
4. Home Page
5. Price Comparison Page

Executive Summary

7Factor is a cloud services company that works with clients to build custom software to fit the client's needs. 7Factor hosts the software they develop on Amazon Web Services (AWS), as they are an experienced AWS partner. Our project team worked with 7Factor to produce an application that 7Factor could use to help minimize the costs they generate from hosting the software they develop on AWS.

Our contributions to the project are a continuation of two previous teams' work completed in the past two years. The first team approached the project by developing a modified knapsack algorithm that recursively loops through the workloads of an AWS account. The recursive knapsack algorithm successfully reduced costs but was slow and inefficiently utilized memory. This weakness is what the second team aimed to improve on by developing a tool that tested possible algorithms against datasets. Their efforts led to a new algorithm that runs on presorted data to assign each workload, putting each workload into the smallest instance that it fits in. This tool eliminated the need for recursion, leading to a more efficient algorithm.

Once we reviewed the previous teams' work, we established several goals to accomplish over the course of this project. The first significant opportunity for improvement we identified from prior teams focused on the front end. As our predecessor's application displayed all its information on a single page, it required the user to scroll repeatedly to access certain parts of the application. Our team believed that this user experience could be greatly refined. Our final goal was to produce an application that 7Factor could readily and easily implement into their operations to help cut back on costs.

After forming objectives for the project, our team divided into groups, each with a set of targets. One team began by redesigning the front end, aiming to enhance user accessibility, make the visualized data easier to understand, and elevate the overall aesthetics. This team created multiple pages and a navigation bar to help users navigate the application, along with pop-ups to give more information to the user on how each page worked. Another team significantly improved the back end by setting up API calls to AWS using access keys to pull the relevant data from any AWS account. When using our tool, this flexibility allows 7Factor to receive feedback on the instances they are actively using by merely entering their access keys instead of manually downloading and uploading the data to the application.

Introduction

Amazon Web Services (AWS) is a flexible and effective set of products used for cloud computing and storage. However, the products that AWS offers are not cheap, especially when used for large-scale computations. As a result, it is pertinent for any company using AWS to reduce expenses by limiting their usage of the services to only what is essential. Finding the optimal configuration for an Amazon Elastic Container Service (ECS) cluster that produces the best results with the lowest costs is not a simple task. The best way to approach the problem is with an optimization tool.

This project is a continuation of the work from two previous MQP teams. Over the past two years the teams created a basic tool that runs different ECS cluster configurations through various algorithms to determine optimal configurations that can be visually represented. Both the algorithms and the front end display function as intended, but significant improvements could have been made regarding data collection, efficiency, and usability.

1. Background

We focused our research on basic AWS functionalities and a few of the important AWS features we would be working with such as EC2 Systems and the ECS, Amazon CloudWatch, and Amazon S3. We also utilized Python libraries from the previous MQP teams' work as well as additional packages necessary for our project. These included SQLAlchemy, FastAPI, Pydantic, and Boto3. In addition, we developed a JavaScript application through the React framework to create a front end, and used Python to establish our back end. This application was developed using the VScode IDE and a GitHub repository for version control.

1.1 7Factor

7Factor is a software engineering company that works with clients to build custom software that fits the client's needs. As a partner with Amazon Web Services, 7Factor hosts the software they develop on AWS. Our project worked with 7Factor to develop software that they could use to help minimize their costs from hosting the software they develop on AWS.

1.2 Amazon Web Services (AWS)

1.2.1 Introduction to AWS

Amazon Web Services is a collection of services that Amazon supplies to individuals and organizations looking for "compute power, database storage, content delivery, or other functionality" (Amazon Web Services, 2023). AWS is often used by software engineers to help support the needs of a web page or application they are working on by helping to create a web domain with a database and their Amazon EC2 system.

1.2.2 Amazon EC2 System

The Amazon Elastic Compute Cloud, or Amazon EC2, is a cloud-based platform that offers a wide range of computing platforms with more than 700 instances. Users can customize the processor, storage, networking, operating system, and purchase model to best fit their needs (Amazon Web Services, 2023). These EC2 clouds act like virtual machines that can run software

and complete tasks. The EC2 system is supported by the Amazon Elastic Container Service (ECS).

1.2.3 Amazon Elastic Container Service (ECS)

The Amazon Elastic Container Service was designed to help manage containerized applications. A Containerized application is one that runs in a package of code that contains all of the relevant dependencies meaning it does not need a specifically configured computer to run it**.** This allows for easy deployment and use of apps and web pages as it helps developers manage resources like storage and processing power for the web page or app (Amazon Web Services, 2023). It can also help manage larger workloads with multiple tasks by helping users understand what and where each task is being completed.

1.3 Python Libraries

Our team used various Python libraries for this project to create our application. These libraries were SQLAlchemy, FastAPI, Pydantic, and Boto3.

1.3.1 SQLAlchemy

SQLAlchemy is a Python SQL toolkit and object-relational mapper. It is primarily used for its persistence patterns and efficient, high-performing database access. Its features include non-opinionated, function-based query construction, modular and expendable raw SQL statement mapping, pre and post-processing of data, and database support, including AWS's Relational Database System (RDS) (SQLAlchemy, 2023). These features were important for creating a way to access data from any type of database systems. This library was used to create our connection to the database, and handle any queries from our back end to the database. A link to the SQLAlchemy documentation can be found in the References section.

1.3.2 FastAPI

FastAPI is a web framework for building APIs with Python based on standard Python type hints. This library is easy to code with due to its intuitive nature and provides fast performance, which was important for creating a robust API with the ability to connect to the front end of our application (*FASTAPI,* 2023*)*. This library was used for creating the endpoints

for our applications API requests. A link to the FastAPI documentation can be found in the References section.

1.3.3 Pydantic

Pydantic is a data validation library for Python. It uses type hints to power schema validation, and it supports JSON schema, dataclasses, customization, and organizations using Pydantic (*Pydantic,* 2023*)*. These features were important for creating a clearly defined structure for the data in our HTTP requests, creating a structure that is clearly defined. This library was used for creating dataclasses for sending the data created on the back end to the front end. A link to the Pydantic documentation can be found in the References section.

1.3.4 Boto3

Boto3 is a Python library for managing and configuring AWS services. Its features include establishing connections with an AWS account and making API calls to a connected account. These features were important as our tool analyzed AWS data and created an easy way to access this data. This library was used to retrieve real-time data from any given AWS account and send it from the back end to the front end. A link to the Boto3 documentation can be found in the References section.

1.4 Technologies Used

1.4.1 Programming Languages

In developing this application, we used a few programming languages. Starting with the front end, we used HTML and CSS to implement the look of our design. We chose these languages because we were familiar with them and they are commonly used for applications similar to ours. HTML helped us build the content of the web pages for our application while CSS helped stylize these pages to create the overall look of the application.

We used JavaScript to implement the applications front end functionality because most web browsers support JavaScript to run websites JavaScript also has a useful library, chart.js, that was used to build and display graphs in our web application (*Chart.js*, n.d.).

Lastly, we used Python to implement our back end. We chose Python due to the extensive number of libraries available. Of these libraries, the most useful were the libraries for making API calls and connecting to AWS, as described above.

1.4.2 React

In implementing our front end, we decided to use React to build our application. React is a library that is often used to develop web application elements and stitch them together to create an entire user interface (React, 2023). We chose React as a tool to develop elements of our application in both HTML and JavaScript.

1.4.3 Bootstrap

Bootstrap is a front end toolkit for developing web applications. Bootstrap even has a specific library for React, which is why we used it (Bootstrap, 2024). Bootstrap was a great source for building the more complex elements of our front end. Specifically, Bootstrap was very useful for creating the carousels and our navigation bar. Thanks to Bootstrap, we could create these elements, improving our front end.

1.4.4 Visual Studio Code (VSCode)

We chose VSCode for our development environment because it is compatible with JavaScript, Python, HTML, and CSS programming languages. VSCode also has a plugin for GitHub repositories that can fetch, push and pull directly from the repository.

1.4.5 Version Control

While working in a team on an application, version control becomes even more important. For our version control, we decided to use GitHub. With GitHub, we created a repository, which we expanded from the main branch into a front end and back end branch to keep these separate throughout development until it was time to connect them. We created individual branches for each team member to prevent the occurrence of any conflicts when developing. This also allowed us to share our code and develop the application efficiently.

1.4.6 Amazon Web Services (AWS)

AWS proved to be useful for us in many ways. The first way we used AWS was accessing a dev environment from 7Factor to collect testing data. We then utilizedAWS's S3 bucket feature to store our code and data, allowing us to connect our front and back end together. In addition to this we completed some testing using the EC2 instances available through AWS. Finally, for our database we utilized a Lambda Function trigger to take the data stored in our S3 bucket and created an RDS database through that workflow.

1.4.7 Docker

Docker is a tool that helps develop, configure, and manage application environments. Applications that use Docker can also be containerized through container services, like AWS ECS (Docker, 2024). Using Docker made creating our environment efficient and easier to manage. The application can run on different operating systems and machines while remaining in the same environment, making the application consistent across different platforms.

2. Methodology

We had two major goals during this project - creating a new front end and expanding the back end. We split our team into two groups to accomplish these tasks in tandem and the process each team went through is detailed in the following sections.

2.1 Front End Development

We took many different steps in our development of the front end. Most of these steps follow standard human-computer interaction methods. The first step is to review the existing design and then design a new look based on the review of the old front end design. The last step is to apply this feedback to the design, developing a fully functional front end. Each subsection covers one of these steps in more detail.

2.1.1 Study Previous Front End

When studying the current front end, there were a few questions we asked ourselves to help us brainstorm what we wanted to include in the new front end design:

- What did we like about the front end?
- What did we not like about it?
- What did we want to keep?
- What could we remove?
- What did we learn?

Using these questions, we figured out the essential pieces of the current front end and what was unnecessary. After examining last year's project, we broke down our findings into the questions we asked ourselves.

What did we like about the front end? We liked how information about the data point was displayed when hovered over. We also thought that the separation of different sections of the application helped make it easier to follow and find specific information.

What did we not like? We thought the design was unintuitive. The entire application was on one long page with buttons that had no explanation of their purpose outside of a name for the buttons. The use of headers to sort information was an un-ideal organization tool, as navigating the application by scrolling is inefficient, as shown in Figure 1 below. While we liked the use of graphs to display data, the existing graphs could have been more intuitive. The graphs lack a title and axis legends,making it unclear what was being displayed. Additionally, except for a few data points, the vast majority of data points were located at about zero on the y-axis. This is caused by a few outlying data points with a higher y-axis value. As a result, it is impossible to tell where most data points are located along the y-axis.

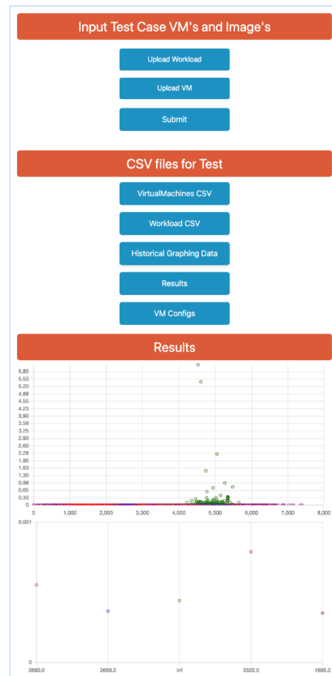


Figure 1: Previous Team's Front End

What did we want to keep? We thought the graphs were a vital tool in displaying the data. However, the graphs could be improved from the previous front end. To help the graphs serve their purpose better, we planned to make changes to allow the graph to be intuitive for the user and improve its function as a visualization tool. Another thing we liked was the buttons. Again, we liked the idea and wanted to revamp the way they are used in our front end. We wanted to use similar buttons in our design; however, they would be used for a different purpose than in the previous front end.

What could be removed? We believed the majority of the functionality provided by the buttons should be removed, such as the buttons to upload and download data. They allowed the user to download private information that should not be downloaded to a personal machine. Storing this data on a personal machine increased the possibility of this private data being unintentionally spread from this machine to others.

What did we learn? We learned a lot about how we wanted our app to be designed outside of what was currently in use. We learned how important it is to have the app be

understandable to any user and wanted to ensure that our app would have instructions on how to use it. We also wanted our graphs to be more detailed and readable. Lastly, we realized how beneficial a multi-page tool was. With the current app, everything is on one page with minimal spacing. This makes the application inefficient and difficult for the user to understand. A multi-page tool would allow for the user to have the information displayed over multiple pages helping to find specific information more easily. This could be achieved through buttons and the implementation of a navigation bar.

2.1.2 Design a New Front End Look

After reviewing the current front end we began designing a new one. To do this, we used Figma, a website used to create mockups and prototypes of application designs without having to code them. Our Figma design can be seen in Figure 2 below. Using Figma and what we have learned about the current design, we began creating a new functional design. For our website, we created multiple designs with slight differences so we could see how our app would look based on different ideas we had.[1] We also used the prototype functionality to see how all pages were connected and to confirm that a user wouldn't be stuck on a page at any given point.

[1] This is the link to the full Figma page for future groups to use to help design future iterations of the app: https://shorturl.at/mCGM0

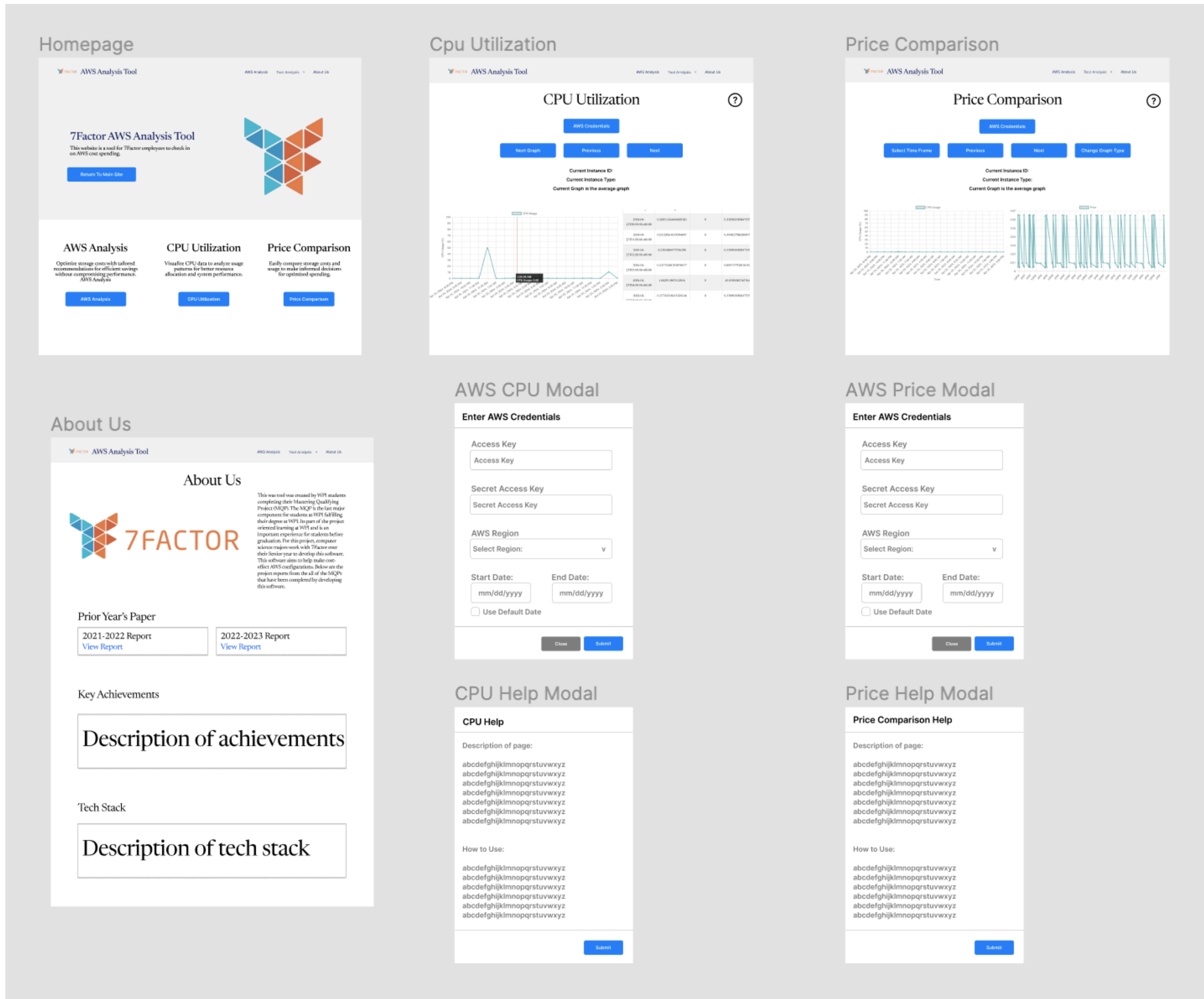


Figure 2: Figma Designs

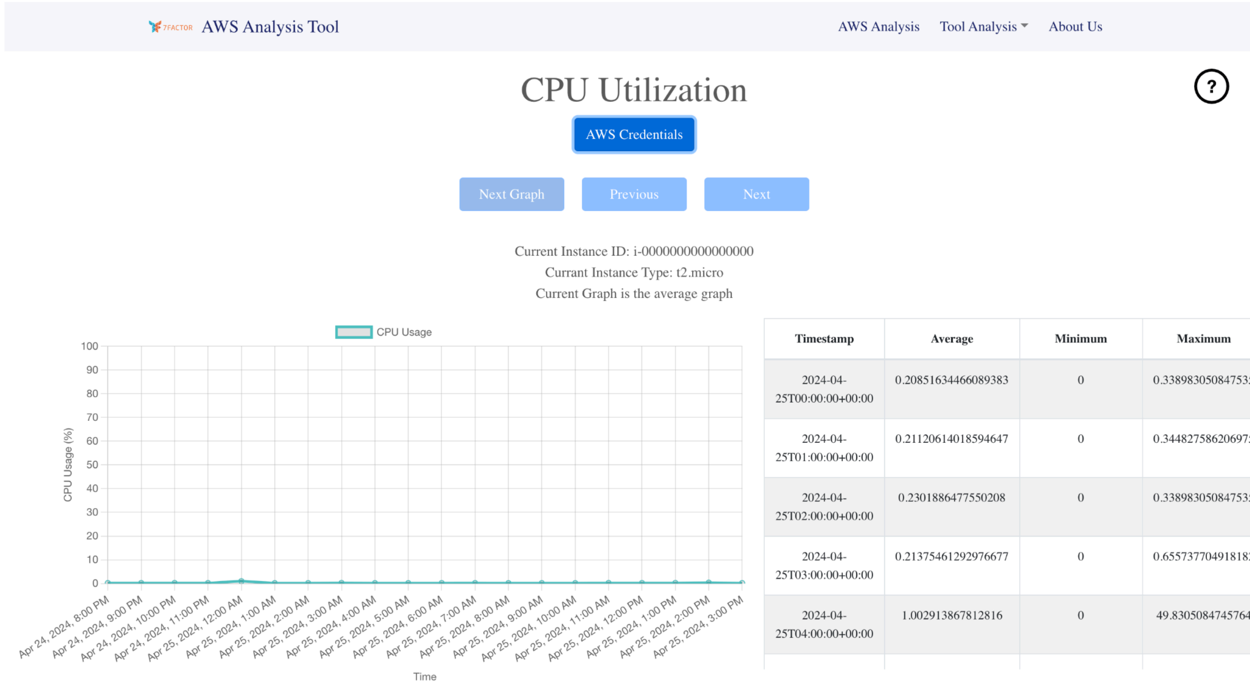
2.1.3 Develop New Front End

After completing the new front end design, our next step was to convert the Figma mock-up to functional React code. Initially, our idea was to manually code all of our mock-ups into proper React code, but we were hopeful that there was a more efficient way of doing this. Our first thought was to use a front end framework (such as Bootstrap) to make it so we would have an easier time. Our research led us to the software Builder.io, which takes a Figma mock-up and makes the general outline in proper code (*Builder.io, n.d)*. For example, if we had a Figma page with a title, buttons, and images, the software would convert it to React code with non-operational buttons. Fixing the buttons was very easy, as the software (in most cases) knew they were buttons, and we just had to change the functionality so that when pressed, they would

lead to a new page. When Builder.io did not realize the Figma mock-up was trying to display a button, it would mess up the code generated, but even still, it provided helpful information such as layout spacing.

While this way of creating our front end was incredibly useful, it did have its faults. As a result, we used a combination of this software and React Bootstrap to make our front end. The creation of the pages themselves were coded using the React Bootstrap framework, but by using this information provided by Builder.io, the pages were easier to make. Using Bootstrap allowed us to have a consistent format using their components to help make pages easy to replicate. We developed this to allow for future additions to be added quickly and cleanly. For more complex scenarios on our front end, such as graphs being displayed and updated as the AWS analysis occurred, we coded React components that had their own back end to retrieve the data needed and would complete their task. An example is that the graphs we displayed were created in separate components. When we needed to create a graph on a page, we simply called the component and passed in the data, and then we knew the graphs would be created properly.

We also wanted to ensure that our web application would look and run the same on different computers and laptops. In order to accomplish this, we used Docker, an environment tool development and management tool (Docker, 2024). This enabled us to continue to develop our web application without needing to worry about compatibility with different operating systems and web browsers.



Timestamp	Average	Minimum	Maximum
2024-04-25T00:00:00+00:00	0.20851634466089383	0	0.33898305084753
2024-04-25T01:00:00+00:00	0.21120614018594647	0	0.34482758620697
2024-04-25T02:00:00+00:00	0.2301886477550208	0	0.33898305084753
2024-04-25T03:00:00+00:00	0.21375461292976677	0	0.65573770491818
2024-04-25T04:00:00+00:00	1.002913867812816	0	49.8305084745764

Figure 3: CPU Utilization Page

When working on our project, our team divided ourselves into front end and back end teams. This enabled our front end team to begin visualizing our data immediately, without having to wait for our back end team to set up real-time AWS API connections. An example of this is shown in Figure 3 above. While the back end group wasn't finished with all of their data manipulation, they were far enough along to know what information they would need from a user. Initially, the back end team retrieved data for this page by inputting all AWS credentials manually. This manual setup phase provided our team access to multiple CSV files that the front end team could then utilize for testing the generated charts and tables. This process was later automated, such that when the user entered in their AWS credentials, the front end would pass the credentials to the back end, which would automatically return necessary data for the front end to display.

The last objective we wanted to achieve was to ensure that the front end of our application would run on any laptop or computer in the same way. To achieve this, we used Docker to set up the front end environment. Using Docker ensured that the front end environment was always built the same way without any issues due to different operating

systems or hardware. This also helped allow the web application to appear the same on any browser, meaning the user can run the application on any computer.

2.2 Back End Development

There were three main components to our back-end development to produce the desired result of an optimal AWS EC2 cluster configuration. The first aspect of this process was creating and running API calls based on your AWS credentials to pull the data on each EC2 instance in your environment. Next was converting and storing the data into a usable structure for the following step. The last step was running the converted data through an algorithm to locate suboptimal utilization of EC2 instances. After running the data through the algorithm, a potential optimization was displayed if any non-optimal setups were detected.

2.2.1 Study Previous Back End

Before we started working on the project's back end, we took some time to analyze what last year's team did. The key aspects of last year's codebase included the functionality to import CSV files as a data input and applying algorithms to that data which used RAM and CPU as main parameters. Unfortunately, prior teams did not test with AWS data and instead applied their algorithms to hypothetical datasets of their own design. This structure worked, but we aimed to improve user-friendliness and utilize real-time AWS data rather than files on a local machine. In our application, we switched to an API call-based data collection method instead of manually inputting data. With this system in place, so long as the user has a valid AWS account with EC2 instances, they can run the app without any further knowledge and without the risk of formatting issues. The second alteration we made was to specifically employ CPU utilization data from these API calls as a utilization metric for EC2 instances. We focused on this data because EC2 instances are billed based on the time they are used, not how intensively they are used. AWS also offers many ways to track the CPU utilization of any EC2 instance, so we wanted to be able to use that information to provide better optimizations. We also added functionality for the app to pull real-time pricing data for EC2 instances to compute how much money the recommendations could save.

2.2.2 API Calls

The first step in our process was to create a generalized form of an API call to AWS CloudWatch that allowed the app to take in AWS credentials for the account with the clusters to optimize. Allowing user input for the credentials was critical to enable the application to be run on any machine while still being secure. The most impactful data point pulled from the API call was the CPU utilization. To send the API calls, we used the Boto3 Python library to streamline the process, as it has methods that can establish an AWS connection and send an API request. The following section discusses how we processed the data returned by the API.

2.2.3 Data Processing

The Boto3 AWS API call to CloudWatch returns data as a combination of Python lists and dictionaries, so we needed to parse through each response and find the exact part we needed. The part we looked for was the CPU utilization which itself was a list of pairs of timestamps and values. Each list of CPU utilizations was added to a separate new list. This new list was what we sent back to the front end, and it contained a list entry for each EC2 instance that included a list of timestamp and CPU utilization value pairs.

2.2.4 AWS RDS Database

In managing the price changes for all possible EC2 instance types, our team recognized that the querying of this data would be resource-intensive. To assist in decreasing our application processing time, our team implemented an AWS RDS database to store large datasets. To accomplish this, we first saved API data related to CPU utilization of EC2 instances and AWS pricing to an AWS S3 bucket. Our team then created an AWS Lambda function, which would run Python code designed to transfer data from our S3 bucket into a newly established RDS database. This Lambda function was programmed to trigger after any new file was uploaded to our S3 bucket, and the data in the file was then imported into the appropriate table in the RDS database. This process demanded a tremendous amount of trial and error, as there were many incompatibility issues between Lambda Python and the packages we attempted to make use of. In addition, this process required specific security rules to allow the Lambda function to properly process data from our S3 bucket into the RDS database.

2.2.5 Optimization

Our initial optimization method was to pass in all of the CSV files and calculate, based on average CPU utilization, whether or not the EC2 cluster's size should increase, decrease, or stay the same. This is not the best logic to test the data, since it only considers the average utilization, but it serves as a solid baseline. In reality, the utilization may spike at different times, meaning the cluster shouldn't decrease in size when the logic tells it to. To remedy this, we started to incorporate the maximum and minimum utilizations to see how the utilization fluctuates. This helped produce a better optimization result. The algorithm is far from perfect, but we wanted to focus on polishing the structure so that future groups could easily understand and modify it.

2.3 Docker

Docker is a containerization software that allows developers to build, share, and run applications anywhere. In order to create a container, a developer must first create a Dockerfile, which can then be used to create a Docker Image which can be sent anywhere, and with this Docker Image a Docker Container can be run. Using the Docker Container made for easier collaboration across different laptops and computers with different operating systems. Furthermore, it makes it fast and simple for the application to be deployed.

2.3.1 Front End Use

We used Docker on the front end to help ensure the front end looks the same on any laptop or desktop. Our Dockerfile used to implement this includes node:17-alpine to create the node application. This also runs the install commands for the React libraries, copying all of the files from the project, and then building the application. This also helped remove the installation steps from the user, as all of the libraries and packages that may have been installed manually are now all installed through the Dockerfile.

2.3.2 Back End Use

Similarly to the front end, we used Docker to ensure that the back end is implemented properly on any laptop or desktop. The Dockerfile includes Python:3.9-slim to create the Python

application. It also copies all of the files from the project, runs the install commands for the Python libraries before building and running the server. This also makes it easier for the application to be installed by running all the required installations from the Dockerfile.

2.3.3 Docker Compose

With both the front end and the back end containing Dockerfiles, we utilized Docker Compose to build the front end and back end together. The compose file contains both the front end and back end services, where we predetermined the ports for both the front end and back end, and contain any environment variables we need for the back end in a .env file.

3. Results

In this section, we will discuss the results of our work. We will start with our application as a whole and what we were able to accomplish with it. We will then discuss the specific components of our front end and back end and what we were able to accomplish with them.

3.1 Application

The application we built successfully completed the methods we wanted it to be capable of. The first method we implemented was a tool to track CPU utilization of EC2 instances for an AWS account. This is used to pull data from an AWS account and look at how the CPU (s) are being utilized over time by each instance. The second method we implemented was an EC2 instance price comparison. This method is used to see the cost of an instance compared to the CPU utilization over time to see how the overall cost changes. The last method we implemented was the AWS recommendations. This method utilizes the previous two methods to give recommendations of changes to be made on the AWS account's EC2 instance in order to cut back costs. This is the practical use of the software for 7Factor to use when creating or updating software they're hosting on AWS.

3.2 Front End

We decided to redesign the front end of the software to make it easier for a user to use and navigate. This meant improving the navigation throughout the front end. The previous front end was done using one page and used scrolling to find most of the utility. To help fix this, we implemented multiple pages with a navigation bar that can be used to return to any of the pages which can be seen in Figure 4 below. This helps to make each page readable without needing to scroll in order to find different tools of the software. The navigation bar was built to help the users navigate back to the homepage quickly and easily to start using a different tool.

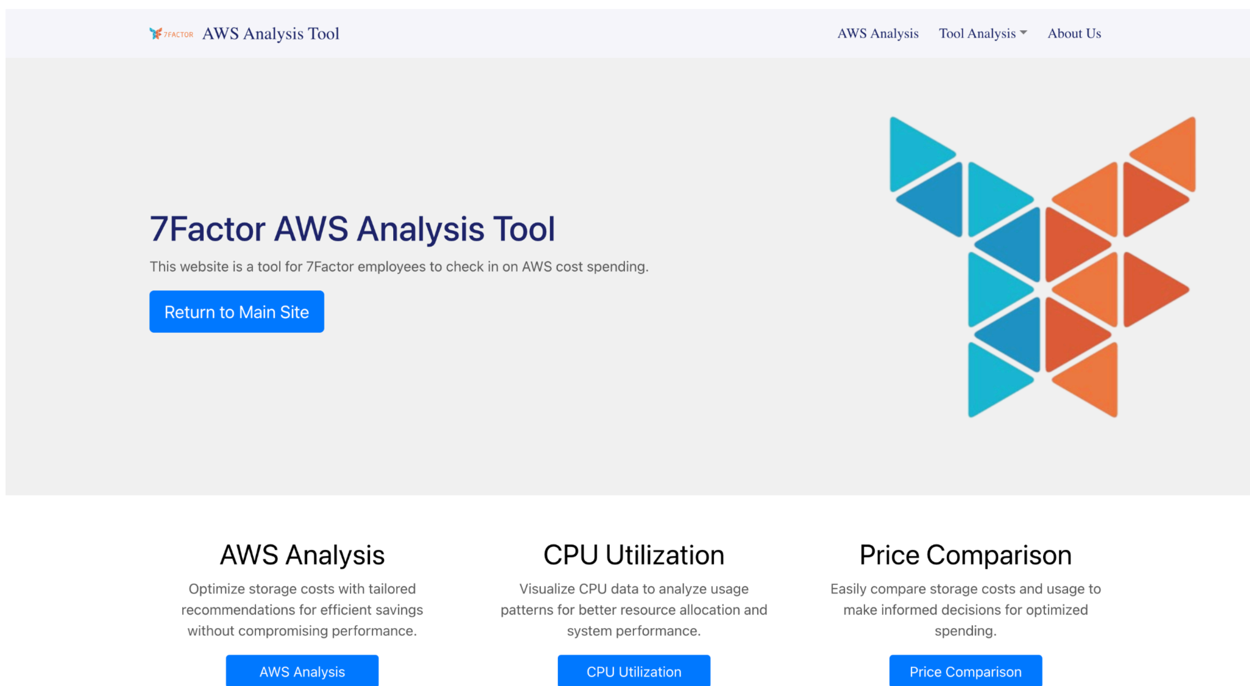


Figure 4: Home Page

We also spent time improving the graphs on the application. We looked to display more data while making the graphs easier to read for the user. We achieved this by using a carousel to display three graphs representing CPU utilization and cost over time for each EC2 instance. The three graphs we displayed are the maximum, average and minimum over time. We also made the graphs easier to see the change over time and data points by editing the scaling of the graphs. We also added a slider to the graph to help the user read the data in the graph. The new graphs can be seen in Figure 5 below.

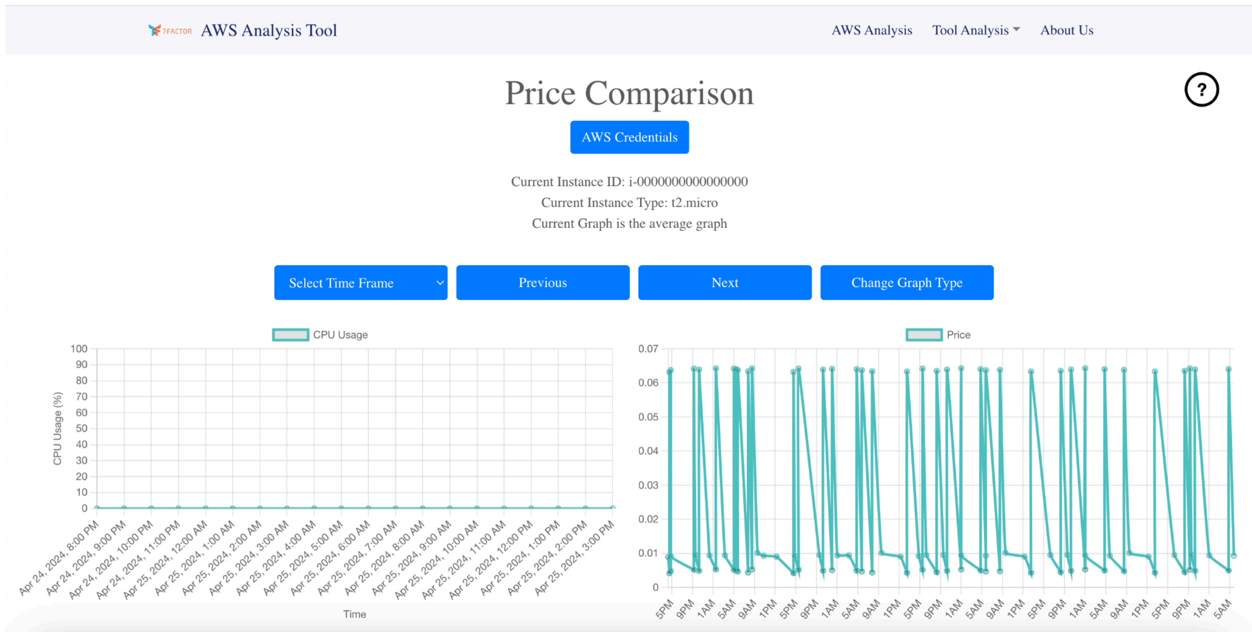


Figure 5: Price Comparison Page

3.3 Back End

Our back end code takes in AWS credentials as user input and returns a list of the optimal EC2 instance configurations. The optimizations are presented as on-screen text and will recommend either reducing the instances to a smaller type if it is being underutilized or leaving it as is if reducing it would impact the performance. We created a baseline algorithm that finds periods of low utilization and recommends reducing the instance type if possible. We arbitrarily set the threshold of low utilization at 45% so that we could test it, but there is plenty of room to improve upon it. We also didn't try to update the algorithm yet, as the only data we were able to acquire in abundance was from 7Factor's development AWS environment. This was great for making sure the app could run, but it was not data that we wanted to fine-tune the algorithm around. The data we wanted to use was client data, but that is something only 7Factor would be able to get. We ultimately wanted to create a strong, modifiable baseline so that any future teams will be able to jump-start the process without needing to deal with pulling and parsing the data.

4. Discussion

This section is dedicated to discussing our team's dynamics, focussing on lessons learned and opportunities for improvement. First, we will start by reviewing a unique hurdle for our project in that we spent almost an entire term without being able to access the previous team's work on this project. With this in mind, we spent our first few months speculating on how the other team's application functioned purely through its description in their report. After receiving this code, our entire project pivoted into dissecting our predecessor's application and brainstorming areas for improvement. We plan to make it easier for the next team by ensuring that our code repository is in the project advisor's and 7Factor's hands to give to the team next year.

Once we started working with the previous team's code, we decided to set up weekly meetings as a group outside of meeting with the project advisor. With these weekly meetings, we were able to plan out who would work on different timeline goals each week to keep the work spread out. It was also a time for us to share our progress from the previous meeting and discuss any changes that we needed to make from our original plan. This kept us on track and in good communication, especially when our team split into front and back end teams. We were able to talk about the progress on the front and back ends as well as communicate what each needed from the other to complete the application.

One issue we ran into was setting up meeting times. To help with this, we used when2meet to find out each other's schedules in order to find the time to set up meetings. With each of us having schedules that sometimes changed week to week, it became harder to ensure everyone could make it to the meeting. This issue also made it difficult to set the times we were available for the coming week(s), as some things changed with our responsibilities outside of this project, which meant planning meetings much later than we preferred.

Another issue we ran into occurred when we created our survey for the front end user and we discovered that this survey needed to receive approval from the IRB. This ended up slowing down our process on the survey tremendously, and we didn't really get a chance to use the feedback to make adjustments because of it. Due to this experience, we would suggest that future teams think about starting their work on any survey as soon as possible to make sure they can improve the front end based on the feedback they get.

Finally, a constant and significant challenge in our project was the struggle of learning and implementing many different Amazon Web Services and their interactions with each other. While making API calls, setting up EC2 instances, and establishing S3 buckets was fairly simple, automating our program to utilize all of the services this project required was arduous. For example, our team encountered a substantial bottleneck in processing large amounts of data from our S3 bucket and then populating graphs in our front end with that data. Our solution to this problem was to establish an RDS database that would store this information, which could be queried much faster than the files in our S3 bucket. This proved to be a demanding process to automate. Our first approach was to employ the use of an Amazon service called AWS Glue to create a pipeline that would transfer data from our S3 bucket into tables within our RDS database. AWS Glue uses independent functions called ‘crawlers’ to prepare data within S3 buckets to be structured as tables that can then be placed into a database. This function on its own however did not transfer the data into an RDS database, instead for this process our team needed to take advantage of another function called an ‘ETL job’. This function was intended to finalize the pipeline as follows: API calls are made to collect EC2 and pricing data with the proper credentials, this data is saved to an S3 bucket with separate credentials, a crawler is triggered to restructure this data into tables, an ETL job then transfers these tables into our RDS database, which our front end then visualized. Despite all these efforts, our team could not successfully establish this pipeline. Instead, we then were able to accomplish this same task utilizing AWS Lambda which had its own host of difficulties in implementing. This investigation of AWS Glue is just a sample of the numerous services we attempted to take advantage of for this project. As our team had virtually no experience with AWS, it was clearly demanding and frustrating to learn and implement these services.

5. Future Work

With the work that we’ve completed, there is still much more that can be done to improve this software. Here we are going to list some of the improvements we think could be made to our software.

5.1 Algorithm Analysis

Our application allows for 7Factor to use the app to cut back on costs for their AWS EC2 instances. In this manner, we believe there is still room to improve the application's algorithm in order for the application to increase efficiency. This would help the application run efficiently on larger datasets that 7Factor may have. In addition, while applicable to any ECS cluster, our team solely tested our application on 7Factor's developer environment. Although 7Factor may internally use this application in their production environment, it would be highly beneficial if future teams could work with 7Factor in this implementation and improve upon the application's algorithm based on this new, more applicable data.

5.2 Front End Device Compatibility

For this future work, we propose looking into different devices that the user may want to use. With these devices in mind, test and build out a front end that works well for these devices. The current front end was built with only desktop use in mind, and that may make users with handheld devices have a harder time navigating and using the software.

5.3 Database Expansion

We found that hosting the application using an S3 bucket on AWS was likely causing some of the functionality of the application to run slowly. We created and implemented an RDS database to store the data to query. This allowed the application to run more efficiently. The database could be expanded to include more data types to increase the accuracy and expand the scope to other AWS features. The database could also be analyzed to make sure the database's structure is optimal.

5.4 Automation of EC2 Instance Changes

As we reached the ending point of our application, we realized the possibility of automatically updating EC2 instances to the recommended instances from our algorithm. This would be a feature to add to the application that would recommend changes and have a button

for the user to confirm or deny the changes recommended. This feature could also be developed into building out a script(s) for certain times of the day to run and change or even shut down instances to further save money.

6. Conclusion

Throughout our work on this project, we faced challenges and found success. We ran into challenges in connecting the front end to the back of our software, but we managed to connect them successfully. This allowed us to improve the tool from last year. On the front end, the implementation of multiple pages makes user navigation simple and easy. The user can access any page from another page with the navigation bar. We also created a help pop-up to help the users understand and use the application effectively. On the back end, we set up API calls to AWS using access keys to pull data from an AWS account to display in the graphs on each page of the application. This makes our application practical for 7Factor to use, as they would no longer need to manually pull data from AWS accounts, making it quicker and easier for them to get the information instances that can be changed to save money. Overall, we built software that was improved from the previous team on both the front and back end. The front end is easier to navigate and understand for the user, while the back end automatically runs API calls to pull data from an AWS account.

References

7Factor Software. (2023, August 4). *7Factor work: Technology consulting: How we've helped our customers*. 7Factor Software. https://7factor.io/aws-partner/

Amazon Web Services. (2023, December). *Cloud computing services - amazon web services (AWS)*. Amazon Web Services. https://aws.amazon.com/

Amazon Web Services. (n.d.) *Boto3 documentation.* Amazon Web Services. https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

Builder.io (nd). builder.io. Retrieved December 8, 2023, 2023, from https://builder.io/

Chart.js. (n.d.). Chart.js | Open Source HTML5 Charts for your website. Retrieved February 10, 2024, from https://www.chartjs.org/

Docker. (2024, April 15). *Accelerated Container Application Development*. Docker. https://www.docker.com/

FASTAPI. FastAPI. (2023). https://fastapi.tiangolo.com/

Mark Otto, J. T. (2024). *Bootstrap*. Bootstrap · The most popular HTML, CSS, and JS library in the world. https://getbootstrap.com/

Pydantic¶. Pydantic. (2023). https://docs.pydantic.dev/latest/

React. (2024). https://react.dev/

SQLAlchemy. (2023). *Sqlalchemy*. SQLAlchemy. https://www.sqlalchemy.org/features.html

Glossary

1. **7Factor:** The sponsor of this MQP. They're a company that develops software solutions for their clients, which are mainly other companies that contract them
2. **Amazon Web Services (AWS):** Amazon's platform to host hundreds of cloud computing services and features in websites
3. **React:** JavaScript and HTML/CSS framework for building web applications that was used to create the user interface
4. **Bootstrap:** Another framework that we used on top of React to create the user interface. It proved extremely useful for more complex components of the user interface
5. **Docker:** A tool used to host a shared environment allowing for easier configuration and downloading of dependencies, making it easier for multiple developers to work on the same codebase at the same time
6. **FastAPI:** Python API framework used to develop and allow for asynchronous processing
7. **Figma:** User Interface designing tool used to build front end design models to experiment with designs, navigation, and ease of use for the front end

Appendix

Development Environment Setup

The following tools are required before running the AWS Cost Analysis Tool:

1. Docker (version 4.12.0)
2. Git (version 2.39.2)
3. Python (version 3.9)
4. NodeJS (version 17)
5. NPM (version 8.3.1)

How to Use

1) Clone the repo to the user's machine.
2) Download and set up Docker Desktop (a user will need to create an account; Docker guides you through this process).
3) Using the terminal, enter the highest layer of the application named “7FactorMQP”.
4) Create a .env file within the server directory, with the content

```
BUCKET_ACCESS_KEY_ID = “key_id”

BUCKET_SECRET_ACCESS_KEY = “secret_access_key”

DATABASE_URL = “url”
```

   With your given credentials.

5) Run the command

```
docker compose up
```

   The first time this command is run, many packages will be downloaded, and it may take a few minutes to run. Once it has been done once, it will speed up by caching the installations it has already made. It will only take more time when changes are made to the front end or back end making the cached section need to be reinstalled.

6) Navigate to docker and ensure the “Container” section contains two objects, one named front end and one named backend. Next, navigate to the Images tab and ensure the frontend and backend objects are there. Lastly, by navigating to the “Container” section and pressing the “8080:80” port located on the frontend row, the user will be directed to their browser, with the URL being http://localhost:8080/.

7) From here, you can navigate the site via the navbar at the top of the screen or the buttons on the home page.
8) The “AWS Analysis”, “CPU Utilization” and “Price Comparison” pages require the user to enter their AWS Credentials, and the contents of the pages are displayed. For more information on these pages, there is a question mark in the top right corner explaining the functionality of the pages and how to use them.
9) When making changes to the app, the user must go to their Docker “Container” section and delete both the frontend and the backend. Next, the user must navigate to the “Images” tab and delete the frontend and backend. Once these have all been deleted, the user may retype “docker compose up” in the terminal and rerun the app.