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 April 27th, 2023

By:

Sean P. Barbour

Seamus Sullivan

Craig Dunn

Project Advisor:

Professor Joshua Cuneo

Abstract

7Factor is a software development consulting company that receives contracts from clients to produce personalized solutions for them. During the 2021-2022 academic year, a group of WPI students began development of software for the company that calculates the most cost-efficient setup and delegation of EC2 virtual machines to support AWS ECS clusters. The result of their work was a web-based based platform that met its stated purpose while leaving room for future improvements to be made. As such, during the current 2022-2023 academic year, we worked with 7Factor to do just that.

Acknowledgements

We would like to thank the following people for their time and assistance throughout the duration of this project:

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

Jeremy Duvall — CEO of 7Factor

Lindsay Duvall — 7Factor Coordinator

Without their support and guidance, none of this project would be possible.

Table of Contents

List of Figures

Executive Summary

7Factor

7Factor is a software development company that is contracted by a variety of clients across various industries such as finance, media, healthcare, and aviation, to produce software specific to their needs. This includes solutions for handling calculations, optimization, and data processing. In doing so they have developed everything from cloud architecture and delivery pipelines to custom systems and applications. Outside of development, 7Factor also offers advice contracts.

Previous Work

This project was a continuation of a project done last year, which was the initial creation of this software. The initial project used a recursive version of the knapsack algorithm that was capable of sorting workloads into possible server configurations. Due to it being recursive, it struggled with larger sets of workloads, and would require more time and memory to complete its calculations or would not hand back the best possible configuration.

Identifying Areas of Improvement

After reviewing the initial code, it was clear that improvements could be made in multiple areas. The most obvious of these was in algorithm performance. While the code given was technically capable of achieving its purpose, it could not do so efficiently enough to be practical. Simply put, the code was far from capable of managing. Beyond just performance issues, the existing code was also lacking in its front-end. The program had the bare minimum in terms of user interface, offering very little beyond reporting the optimal configuration. As such, we were interested in expanding what the program would be able to present to the user.

Results

Algorithm Optimization

The algorithm saw substantial improvement in both speed and accuracy, providing the best possible solution as much as seven hundred percent faster than previously. From a design perspective, the code was refactored to become simpler, better segmented, and more flexible, readable, and modifiable. Costly operations that plagued the original build were nearly removed entirely, and alternate algorithms were also developed to serve as points of comparison between differing solutions to the problem.

Data Analysis

A lightweight web app was developed to run our experiment hundreds of times in order to collect quality data. The data provided allowed for the analysis of our solutions against each other, finding the optimal algorithm for the combinational problem. By standardizing the workloads and narrowing the algorithms used, we produced quality metrics entailing optimal EC2 configurations across different workload sizes.

User Interface and Front-End Expansion

Greater functionality has been given to the user through the usage of the web application. 7Factor specifically requested the implementation of data visualization, allowing the user to better understand the magnitude of difference between multiple potential configurations. The user can also now apply greater

importance to particular demands of the workload. Behind the scenes, front-end code has been greatly overhauled to become simpler, better performing, and more manageable.

Introduction

Memory management on the corporate level has become increasingly difficult as computing has evolved. Greater amounts of infrastructure and investment have been needed as memory requirements have risen, making the feat less realistic, affordable, and tenable, particularly for emerging and smaller companies. As such, the days of local corporate mainframes are fading, and instead companies rely on cloud storage services rented from powers with the wealth and infrastructure needed, such as Amazon Web Services (AWS) and Microsoft Azure. To manage memory most effectively in this manner, it is crucial to only rent as much server space as is needed, and only the amount of processing power required for the task at hand. Doing so minimizes the rental cost. The tool being developed accomplishes this, producing the most cost-efficient way to utilize servers rented through AWS.

While the previous year of work produced a tool technically capable of achieving this goal of minimizing cost, it was not fleshed out enough to be practically usable. The front-end use of the application was bare bones and simplistic, and the algorithm for allocation was too inefficient to handle realistic use cases.

Background

Amazon Web Services and the Purpose of the Project

This project was developed with the direction of 7Factor with the purpose of producing an application that could save them costs by efficiently configuring cloud infrastructure hosted on Amazon Web Services (AWS). 7Factor makes use of the AWS Elastic Container Compute Cloud (EC2), which is a service that provides resizable cloud computation through the use of individual EC2 virtual machine instances. Such instances can be added and removed quickly, and each has their own specifications in regard to memory, vCPUs (threads of a CPU core), processor types, and networking capabilities, among others. Once instances are created, they can be linked to containers created using the AWS Elastic Container Service (ECS). These containers each individually store a set of software which is to be run within the linked EC2 instance.

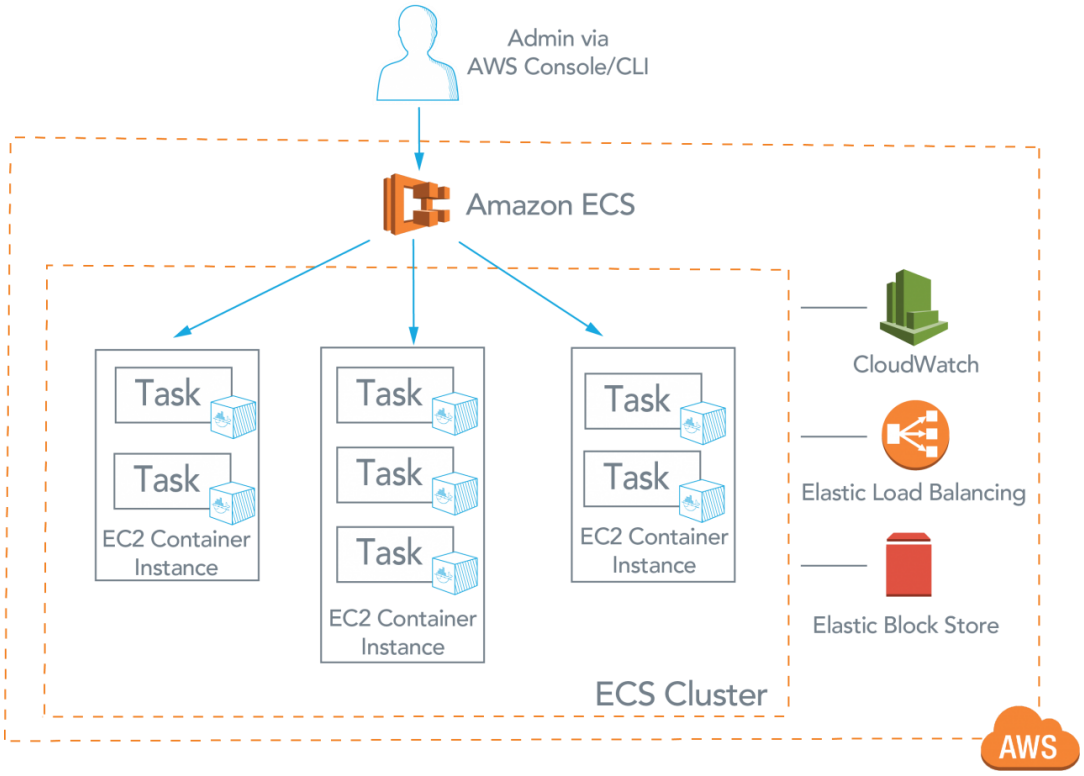

Figure 1: Amazon Web Services EC2 Diagram

Linked ECS containers all have associated task definitions, which describe the technical requirements needed to run their contents properly. As such, containers must be linked with virtual machines whose specifications meet the standard set by the task definitions (Amazon Web Services, Inc.). Multiple tasks can be handled by the same EC2 instance, so long as it can meet the requirements of all the tasks at the same time. Because of this, cost can be saved from avoiding the usage of more EC2 instances by efficiently configuring tasks to be handled within a small number of instances. Given that there are several instance types, each with their own cost and specifications, finding the way to delegate tasks that makes use of the least expensive combination

of instances can be challenging, especially with larger numbers of a diverse range of tasks. Therefore, this project is an application that uses an algorithm to produce exactly that.

The Knapsack Problem

The Knapsack problem is one of the most common and fundamental problems in the world of computer science. It involves a scenario in which there is a set of items with an associated value and weight, and a container with a limited capacity, where the goal is to maximize the value contained without exceeding the capacity. This is described in the sense of someone carrying a knapsack and having to sort objects into it based upon their value, but the knapsack can only hold a certain amount of weight. The person would then have to figure out the best possible items to put in the bag within the weight limit to gain the most value. The knapsack algorithm handles this by creating possible variations and comparing to see which one is best (Thelin).

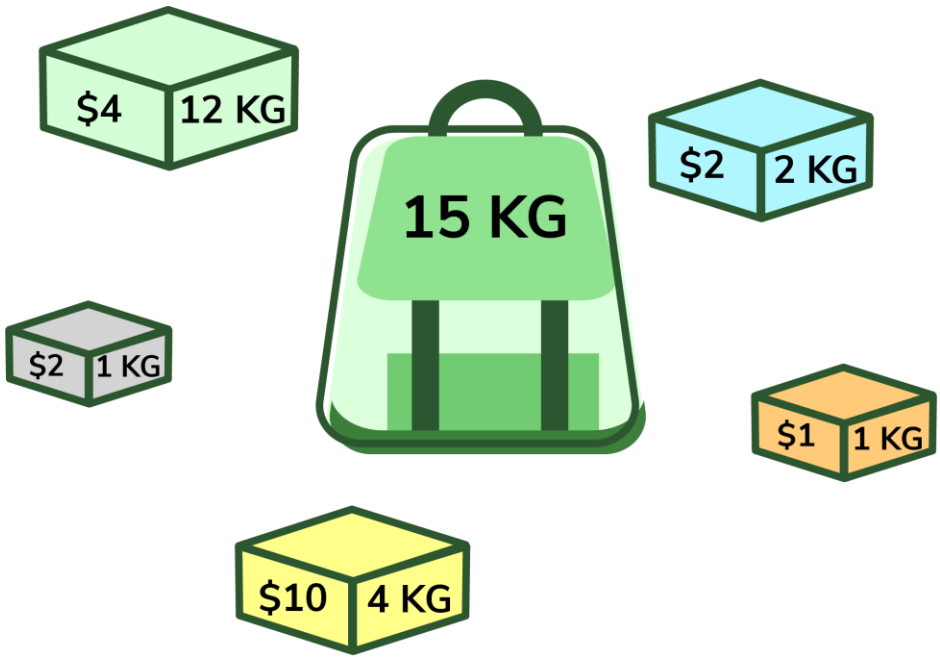

Figure 2: Knapsack Analogy Diagram

Bucket and Marbles Analogy

Another useful angle from which to consider the task at hand is the bucket and marbles analogy. The idea is that each bucket has a given capacity, and the goal is to determine how many marbles can be fit within them. In our case, the 'buckets' represent ECS instances, or individual ECS virtual machines. 'Marbles' represent individual docker containers, which hold a workload the virtual machine will be tasked with. The goal is to sort these marbles (workloads) into these buckets (virtual machines) in the way that incurs the smallest rental cost from renting servers to support the virtual machines.

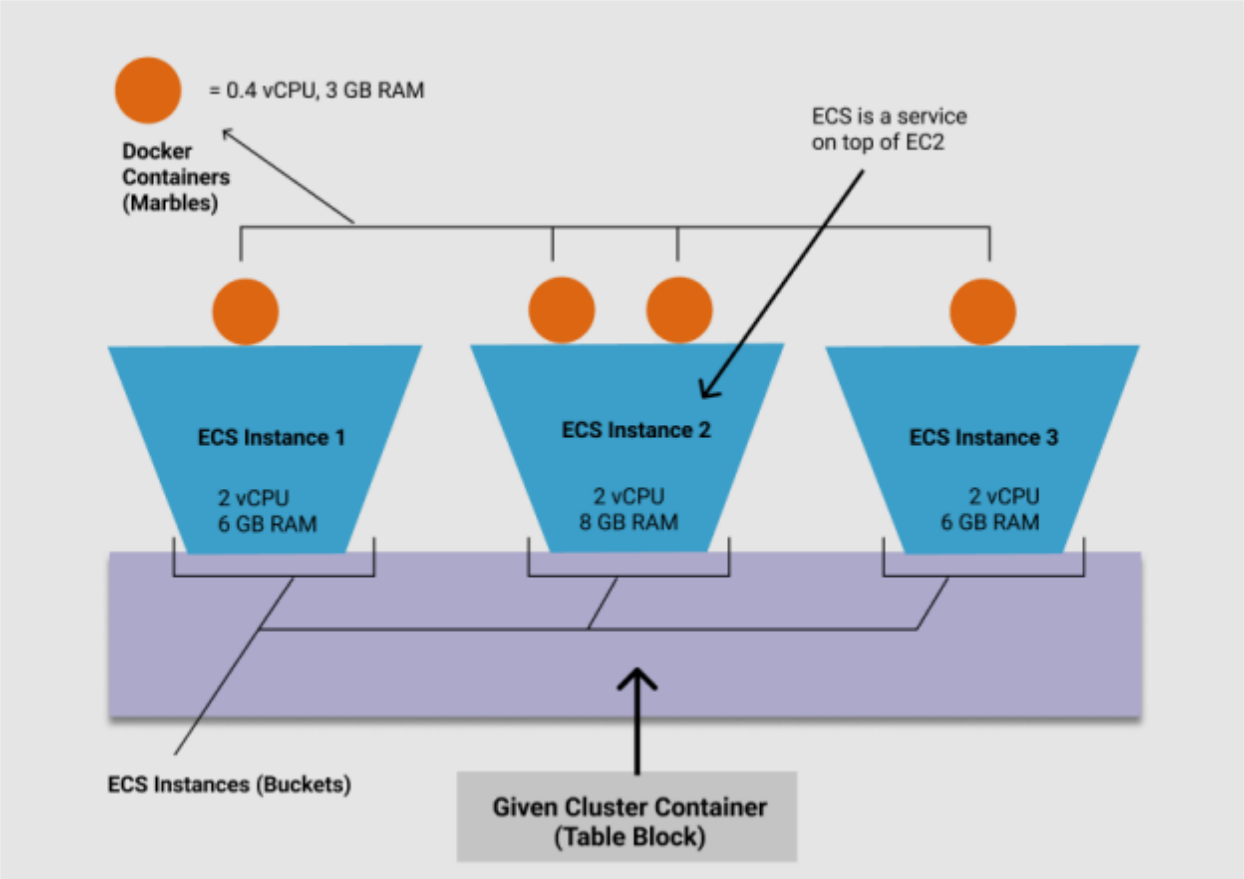

Figure 3: Buckets and Marbles Diagram

Previous Work

The previous team responsible for this project created the foundations for the algorithm, as well as the baseline for a web application that allows a user to work with it. They originally began developing a solution using a standard knapsack algorithm, which is a common approach to solve the aforementioned knapsack problem. The algorithm was then refined by implementing memoization, a technique of storing pervious comparisons to prevent wasting time by making the same comparisons multiple times. Their work was capable of handling smaller sets of data, but struggled when presented with larger, more realistic sets, and while it could usually identify cost-efficient configurations, there were certain circumstances in which it would fail to do so. In addition, the code had some organizational issues, making reading and understanding more difficult. The web application allowed a user to select between ECS clusters and produce a configuration optimized by their algorithm to reduce costs. Following an optimization, the application could report a 30-day savings estimate comparing the cost of the previous configuration with its optimized counterpart.

Methodology

Choices Made by the Previous Developers

Many decisions regarding what languages and tools that the program would make use of were already decided by last year's team. We would ultimately set these up for ourselves and continue to use many of them.

Engineering Process

1. Develop a web-application that could replicate the problem at hand on an experimental level.
2. Implement the previous solution
3. Develop new solutions alongside the current working solution
    3.1. Development through varying algorithm design theory
4. Create visualizations to visualize solutions in comparison to each other
5. Standardize the workload and implement our best solution to create valuable configuration data
6. Re-implement our solution into original architecture while serving configuration data.

Python

The previous group felt as though Python was the ideal language for the project, given its ease of use and its "connectivity with useful services". They wanted a web-based solution for accessibility purposes and identified Python as a common choice to serve as a back-end for such applications. FastAPI was utilized as a framework for this Python back-end because it is easier to learn than its alternatives, allowing them, and later us, to get into the development process more quickly. As the name implies, FastAPI is also best for running quickly, improving potential efficiency with the ability to run concurrent processes (FastAPI).

Python was deemed as the optimal language for the present project, based on the recommendation of the previous team. Its adaptability with a diverse array of frameworks, coupled with its outstanding data handling capabilities, solidifies its position as the language of choice. FastAPI was utilized as a framework for the back-end for its ease of use, allowing our team to stay focused on our project's goals (FastAPI). Python is endowed with a broad spectrum of libraries tailored to algorithmic development, manifested as data manipulation functionalities. Python's versatility is augmented by its ability to operate seamlessly across multiple platforms and architectures. Considering the project's goal of delivering a web-based solution, Python's Flask API provides a lightweight, modular architecture that simplifies both development and integration processes, culminating in an efficient and streamlined solution.

Boto3 as an AWS SDK

Boto3 is an Amazon Web Services Development Kit that is used by Python to interact with AWS. The project makes use of Boto3's ECS, EC2, and Pricing Clients. This gives the code access to information about what ECS clusters are in the account, how much processing power the EC2 virtual machines have, and what their pricing is for them.

Boto3 is built on top of Botocore. Botocore links Python to AWS by allowing for serialization of input parameters, the signing of requests, and the deserializing of response data into Python dictionaries. Botocore allowed last year's team to handle an API request sent to AWS and retrieve the needed response. Dealing with an API request handling involves working with the session as well as its respective credentials and configuration. Additionally, Botocore ensures that every operation which is found exclusively within some particular service can still have permissions to make API calls (Geller).

JavaScript/TypeScript

JavaScript and TypeScript, which is a superset of JavaScript, were used by the previous team for front-end development. There were people on their team who already had experience with the React framework, which is a JavaScript framework for front-end development. The framework is designed to ease the process of creating visuals for a user interface (Meta Platforms, Inc). The MUI (material user interface) library associated with the React streamlines the implementation of common web page assets like lists and sidebars, while also offering customization (Material-UI SAS).

Chart.js for Data Visualization

Chart.js is a popular JavaScript library used for creating interactive data visualizations on the web.

Flask Web API

Flask is a lightweight web application framework written in Python. It easily integrates with middleware and back-end software using routes and basic HTML functionality.

Git

The previous team used Git as their version control system. It allowed them to save the project at each stage of development and allowed them to push multiple unrelated changes and merge them together later. Our group began this project with the latest build in their repository, and we continued to use Git in our own development.

Visual Studio Code

For this project, our team used Visual Studio Code as our Integrated Development Environment (IDE). Visual Studio Code has extensions to support the multiple languages that the different parts of the project are written in, and interfaces well with Git. Additionally, Visual Studio Code is generally feature rich and has a lot of tools that make the development process easier.

Docker

Docker is a tool that creates an environment referred to as a container that the project code is run in to ensure that it is always run in a properly configured environment without having to recreate it locally. All the developers of the code base can access the same Docker environment to ensure that the project is being run in the same environment as everyone else, which helps the code be more portable. It also helps ensure that the project is always being run in the same environment.

Tailwind

Tailwind is an open-source CSS library. The framework grants the ability to implement classes easily within HTML files. Tailwind was used within this project to minimize clutter in front-end code.

Workflow

Team Organization

From the very beginning of project development, we prioritized setting up regular communication. We used the scheduling tool When2meet to organize regular meetings with each other and with our project advisor based on mutual availability. We designated individuals to lead development, documentation, and communication. A project Discord server was created as a place to communicate about the state of development and share resources used to research possible programming solutions and to set standards for deadlines and documentation.

Development Timeline

We made a timeline that listed development goals. As the academic year progressed, we would continually reevaluate and update the timeline based on what we had accomplished by that time and what we deemed was realistic to expect going forward. The table below displays the general idea of what was done throughout each term.

A-Term

As the first term we had to work on the project, the objective was mostly just to gain an understanding of what we had been given to work with from the previous MQP group and what we were expected to contribute to it throughout the remainder of the year. We had a few meetings with Jeremy Duvall, CEO of 7Factor, to discuss what they felt was lacking from the project as it currently stood and what directions we should consider taking it through further development. Reviewing the documentation left by the last team on what they had accomplished, something that we had technically begun doing before the term had started, was a vital starting point in dissecting the application code to figure out how it all worked. Working through the setup to get the application running for ourselves presented some unforeseen difficulties, which required us to meet with the previous MQP group to resolve. Knowing that the primary goal of our efforts was to improve the algorithm that was at the core of application functionality, we began to research potential means of doing so.

B-Term

After finally being able to run the application after meeting with its previous developers, our second term saw us lay the foundations for project work. Early in the term, we committed to a set of development goals, prioritizing algorithm optimization before moving on to data visualization/analysis and further front-end development. At this point, algorithm development mostly involved identifying the aspects of the code that most glaringly added to the runtime, and refactoring in a way that allowed for their removal. The documentation also saw its start in B Term, as we laid out a set of headers and a table of contents which was largely based on the previous report, while accommodating for our goals for the project and the suggestions of our project advisor. We also began to document introductory and background information. By the end of the term, we had gotten much further into algorithm development, writing multiple different algorithms and comparing them to decide between different ways of moving forward.

C-Term

During C Term we implemented the results of our experimentation into a more finalized, well--refined algorithm after determining precisely what choices in development would produce the best results. We would also implement data visualization, a major goal we had set out to accomplish back in the beginning of the previous term. To further polish the algorithm, and to be able to measure and highlight its performance advantages, we developed better means of simulating different scenarios for the algorithm to optimize configurations for--and better means of storing information about--the algorithm's performance through each one of them. The report was also filled out largely throughout this term, as we documented the obstacles we had faced and what we had experienced the far in the development process. What remained to add to the report was divided between group members. Front-end and user interface development had a rocky start as we refactored significant portions of the code to be better modifiable, but by the end of the term it was in a better position to be finished in the following term.

D-Term

By D term, the algorithm and the data visualization were essentially finished. What remained was finalizing documentation, implementing front-end changes and preparing for the project presentation day. The rest of the report that had not been written yet was finished up with added visuals, the project presentation day poster was put together, and the user interface was reintegrated into the newly refactored front-end code.

Obstacles

Transition Between Teams

Our group continued work on an existing project, that had been started by another group. As such, in addition to learning about the problem our group was tasked with solving, we also had to understand and set up what had been developed thus far. Our group had little exposure to many of the tools and software used by the previous group in their work, and initial efforts to set up the project as was given to us (based on their provided instructions) were plagued with issues in doing so. As such, our group had to establish contact with the previous team to sort out these issues, all of which stood in the way of further development.

Deprecated Code

A problem we encountered in the code early in development was that parts of the code had become deprecated, meaning that they had lost their function since the last team implemented it. An example of this is that code relied upon by the program to install Poetry, was no longer supported. Poetry is a tool for dependency management and packaging in Python that much of the program is built on using, so our team could not set up and work with the application. In trying to set up the program, following the directions left for us by the previous group, it was not obvious that there was this deprecated code preventing the application from functioning. As a result, we were left with errors we had no explanation for. Because of this, our team had to meet with the previous team to find what the issue was and where the deprecated code could be found, so that it could be updated to function with Poetry as it is now.

Developing Through Dependencies

In both the front-end and the back-end, the project is tied to a web of different programming languages, frameworks, SDKs, and other tools. Making changes to what has already been written, even simple ones, can

often involve complications that involve multiple such connections at the same time. As such, in many cases it wound up being more efficient to reconstruct larger sections of the code base altogether to make it more cooperative with the changes or additions that were being made.

Depth vs. Efficiency

The two primary goals of the project--to improve speed and to give more information to the user--ended up at odds with each other. Particularly with data visualization, it proved difficult to present as much data as possible without severely impacting the performance of the algorithm. We worked around this issue by creating our own web-based tool to simulate the project. This tool lets us develop new solutions at a greater rate and visualize them in real time. Ultimately, the algorithm developed in our tool was implemented within the previous team's framework.

Approach

Commencing development within a heavily deprecated code base presented a myriad of challenges. To circumvent these issues while maintaining the integrity of the previous project, our team resolved to construct a separate application capable of emulating the functionality of the pre-existing tool, independent of the app architecture. Here we would experiment through the creation and comparison of multiple solutions, each representing different strategies to fulfilling the task. The solutions we devised in this auxiliary application were subsequently integrated into the original code base. This approach of locally testing solutions facilitated greater flexibility throughout the development process, allowing for efficient and effective problem-solving. Through this process of comparison, we had a much faster way of knowing what trains of thought were worth pursuing further and which would only serve as a detriment to the application.

Results

Algorithm Changes

Elimination of Recursion

The original algorithm relied heavily on the use of recursion. Recursion is a costly operation both in terms of memory and processing time, and thus the recursive elements of the algorithm severely inhibited overall performance (Bhargav). As such, the removal of recursion was the first priority in development, as the benefit was obvious and substantial. By the end of the project, no recursion is present in the current algorithm.

Partitioning Algorithm Processes

It is not only important for the function to perform well, but it is also important for its code to be written in a way that is readable and well designed, making it easier both to understand and maintain. This is why the next step in algorithm development was to clean up the algorithm to make it clearer and better organized. We began this task by writing our algorithm in a way that visually separates the steps the algorithm takes to produce its solution. In other words, tasks within the algorithm were broken up into their own visually distinct sections that make it simpler to follow what the purpose of any given line of code is. Following the DRY principle (Don't Repeat Yourself) was also very impactful, as eliminating duplicated code helps greatly in keeping the algorithm modifiable (Blakely). Documentation through commenting is both consistent and concise, and algorithm functions share a consistent format. The implementation of these design principles helped simplify all further algorithm development.

Reducing Intensive Operations

Once the code was cleaned up, the next task was to pinpoint aspects of the algorithm that made use of computationally expensive operations and find ways to achieve the same outcomes in a more efficient manner. Many such operations had to do with memory management. As an example, the original algorithm made extensive use of 'deep' copies. In Python, deep copying is a recursive process of creating a clone of a collection--such a list of costs and resource usage of different virtual machines in our case--and populating this clone with its own copies of each object in the original collection. This copying allows for the algorithm to modify the copied collection without affecting the original, but, especially for larger collections of data, it is an expensive operation (Bader). Minimizing the presence of this kind of copying contributed greatly to algorithm efficiency. This was done in two ways. The first of which was by removing it when it was already unneeded. The second way the amount of deep copying was reduced was by refactoring the code, so it is no longer needed. Not every instance of deep copying could be worked around in this way, but for the most part the algorithm could achieve the same results without it. In a similar vein, limiting the amount of data actively handled by the algorithm itself removed unnecessary computational stress. The original was often found to take in information that was not actually relevant to the task at hand and select what was needed from it throughout the process of execution. Now, the algorithm only receives the data it needs to complete its task, and so it doesn't end up copying, sorting, or maintaining anything more than necessary.

Changing Data Structures

Another way the algorithm was improved was by rethinking the way data was represented to be more multi-faceted and simpler to work with. In the original build, much of the data regarding workloads was stored in stacks. Stacks only allow access to the data in a last-in, first-out fashion. To gain access to data entered earlier, later entries have to be removed from the top of the stack individually. While stacks have their advantages in certain circumstances, given that access is faster, its limitations in our case presented a net detriment to overall performance, and limited the number of options the group had in further development (Siapno). As a result, a key element of the development process was to reconsider the data structures. In the end, we found that simple lists, which in Python are a kind of mutable array, worked much better for what the algorithm is trying to accomplish.

Implementing Presorting and Heuristics

The final and most sophisticated way our algorithm was developed was by incorporating Heuristics. In computer science, a Heuristic is anything that allows a function in code to be guided in finding good solutions. In the previous algorithm, different configurations needed to constantly be compared with all the other different configurations of virtual machines generated to see which is the most efficient. This obviously contributes greatly to the time it takes to run. In utilizing a Heuristic, we aim to reconsider the algorithm to limit the number of needed comparisons without sacrificing how optimal the cost is that is produced (Khan Academy). To this end, we first decided to presort the virtual machines by cost and the workloads by CPU requirement. This adds some additional overhead to the algorithm as it is another set of tasks the algorithm now needs to run through. However, they allow us to make use of our heuristic, which leads to a net gain in performance. By first sorting this information before getting into the algorithm, our heuristic is to make comparisons with recent configurations rather than with all of them. Because the costs have been sorted beforehand, locally optimal configurations end up being very cost efficient overall, which allows us simply to use what is locally the best answer rather than trying to make every possible comparison between every possible configuration.

Current Algorithm Description

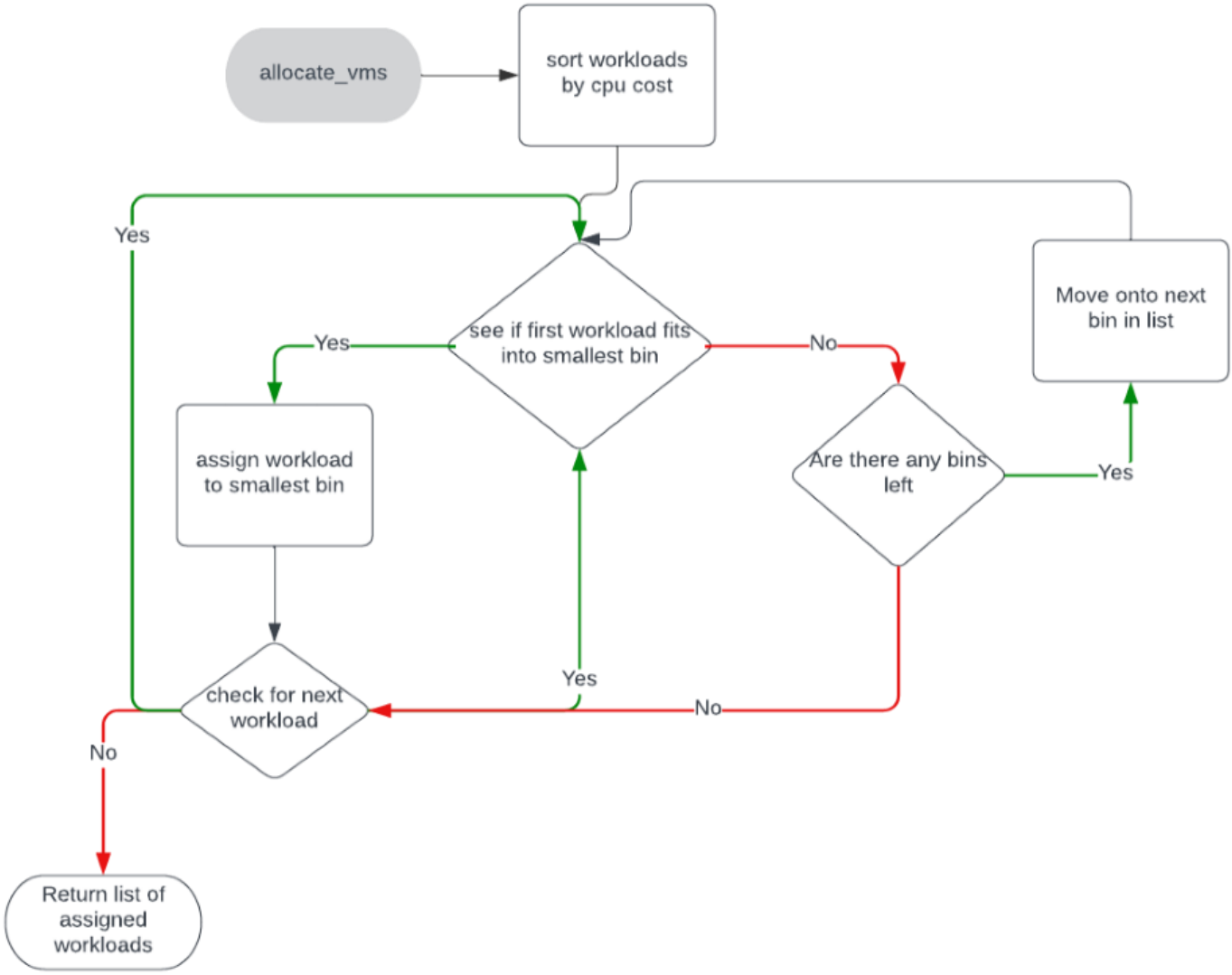

Figure 4: Algorithm Flowchart

The idea behind the algorithm now is quite simple. As displayed in the figure above, the process begins by sorting the workloads by CPU cost. Then, starting with the first workload in the list, the algorithm iterates through bins, from smallest to largest, until finding the smallest bin large enough to accommodate the workload. Once this is done for each workload, the list of assignments is returned. The resulting runtime is O(n), where n is the total number of workloads.

Analysis of Efficiency

To analyze the results of algorithm development, the performance of the current version of the algorithm was measured and compared, not only to the original build, but also to other algorithms that represent alternative approaches to the task. As of now, the algorithm surpasses all the points of comparison while optimizing for cost and run time within small-medium clusters.

Comparison with Previous Performance

This, of course, is the most relevant comparison to the goal of the project, between the algorithm as we had received it and the algorithm as we have left it. The newer model follows a simpler and more greedy approach, iterating through and comparing costs far fewer times. The penalty for this is that on its own, the optimal solution is not always found. However, by supplementing such a strategy by presorting the data which enables some heuristic considerations, it can consistently produce the best cost without much compromise on speed. This solution maintains such a high level of performance in a variety of virtual machine configurations and can handle large clusters.

By contrast, the original algorithm lags, both literally, as it is much slower, and figuratively, as it is less accurate in producing a cost-efficient solution. The optimal solution is not always guaranteed because of certain assumptions embedded into the code that end up excluding configurations that could potentially be the best. While the algorithm is otherwise thorough, it comes at a massive cost in terms of runtime. The application runs slowly even with small data sizes and really struggles with larger clusters. It is quite demanding of resources, both in terms of processing and memoization. The margin of improvement between the old algorithm and the new one is massive. It isn't possible to give an exact ratio between the time it takes each algorithm to run. This is because the time complexity of our new algorithm is O(m*n), where m is the number of workloads and n is the number of VMs, while the algorithm used last year has a time complexity of O(2^mn), meaning it is exponential while ours is linear. As such, as the number of workloads increases, the gap in performance between the algorithms also increases. Run time differences between the algorithms can also vary based on other qualities of a given run such as the variation and order of the workloads being used. In our testing through simulated runs of 100 distinct workloads and 25 distinct virtual machines, we found our algorithm to generally run about five times faster than the previous one and produce a configuration that is about one fifth of the cost.

Comparison with Basic Knapsack Algorithm

Another point of comparison useful in analyzing performance is with a generic implementation of the knapsack algorithm. The previous team developed the original algorithm as a modified version of the knapsack, and having performance metrics allows us to compare performance with a common and naïve

approach often used for this kind of problem. Because of the multiple variables at play (needed CPU and RAM), implementing a basic knapsack algorithm is more complex than it would be otherwise, which also makes it difficult to further modify. As should be expected, performance is far from ideal this way, especially for large clusters. There is also a plethora of scenarios in which inferior solutions are produced because several virtual machines are left only partially utilized.

Comparison with Static Algorithm

The remaining two algorithms represent our experimentation with data structure. As stated before, the original algorithm relied on stacks, a data structure that presented many challenges for further development. In choosing how to move forward, it was initially unclear whether the focus should be on static or dynamic data structures. Static structures preserve the original benefit of utilizing stacks, as they provide quick access to the information they store. However, they are of a fixed size, which limits their flexibility and ease of use (Shivam). Still, given that static structures share more in common with the original stack structure, this algorithm was developed to see if this quicker access could finally be properly taken advantage of. In this algorithm, we attempted to use tuples, a static type of array in Python, to store and manage virtual machine data. Unfortunately, this effort did not bear much fruit. The set size of static data structures does not lend itself well to the task at hand, given that there are many possible scenarios that have to be accounted for. Scalability was rather limited, and the memory usage was inefficient. This hurt the static algorithm significantly in terms of cost and speed optimization.

Comparison with Dynamic Algorithm

The dynamic algorithm is the result of further data structure experimentation. Unlike their static counterparts, dynamic structures can be changed in size throughout the course of the algorithm being run. Accessing and modifying the data in a dynamic structure, however, is slower given that memory can need to be allocated or deallocated. In developing the dynamic algorithm, this loss of speed is accepted in hopes that the greater flexibility from resizing can contribute to a net gain in performance. With this algorithm specifically, this came in the form of utilizing lists, a mutable type of array in Python. This sacrifice ultimately paid off, as the nature of dynamic structures better complimented the task, making it easier to adapt for different situations. The logic behind the dynamic algorithm is like the original recursive knapsack but manages to outperform it as the problem scales larger. While that does make the dynamic algorithm a reasonably good solution, it is still inferior to the finalized design, both in terms of cost and speed.

Data Visualization

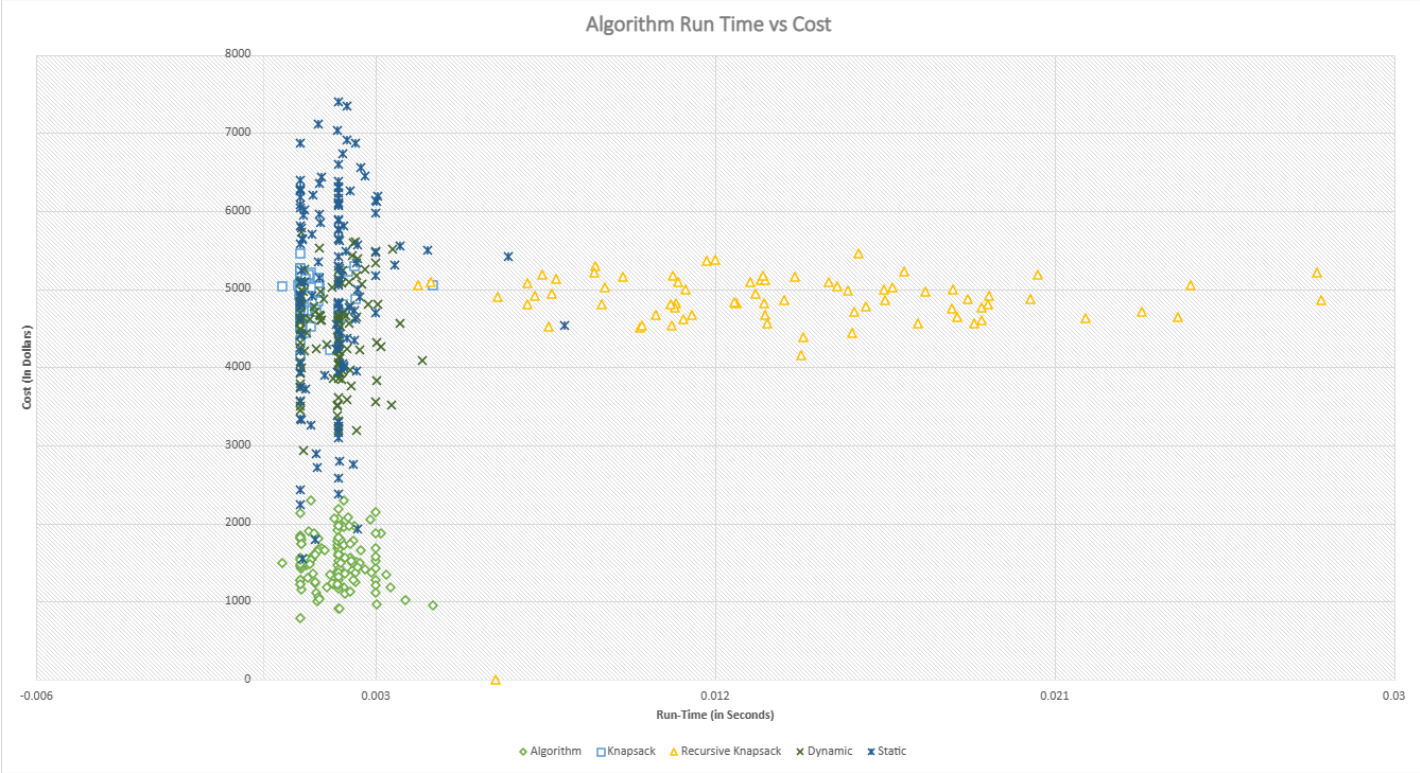

Figure 5: Simulated Algorithm Runs Scatterplot

As can be seen in the figure above, our finished algorithm is able to minimize both run-time and cost, both compared to the previous version (the Recursive Knapsack), and other potential solutions. The data displayed in the graph was collected by simulating 200 workloads across 25 virtual machines. Each competing algorithm underwent 100 such simulations to produce the figure. Each item on the scatterplot represents the result the associated algorithm produced when ran on one of these simulations. Its location on the X axis represents the time the run took to complete. The Y axis shows the cost of the configuration that that run was able to return. For privacy reasons, workload data that was used within the simulated runs does not represent the actual workloads that 7Factor would hypothetically have the application work with. As such, the data in the scatterplot is more useful to how the algorithms perform relative to each other than it is to show the actual run time and costs 7Factor should expect them to produce.

Our Development Tool

As stated in the methodologies section, most algorithm development took place in a lightweight development application designed to allow us to quickly write and test different algorithms and algorithm strategies. Our web applications UX was originally modeled after the established tools UX. This meant that while functional, it was extremely simple and did not follow design standards. The UX now offers an intuitive

design for interfacing with the tool. The ability to upload custom workloads and virtual machine configurations allows for the tool to optimize for different factors. Static result files are served, and a data visualization is created to show the results of the experiment.

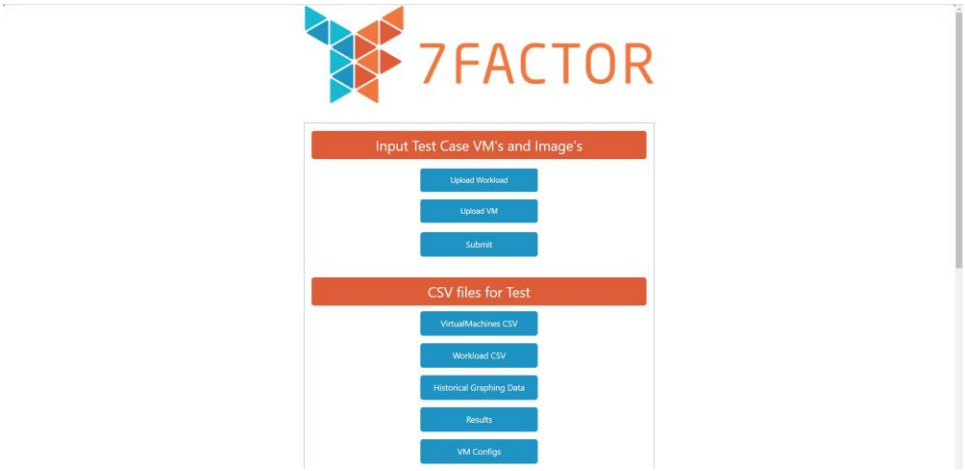

Figure 6: Development Tool Options

The figure above shows the main options for the development tool. Here the user can upload their own workloads and virtual machine configurations, submit them to be run by the different algorithms developed for comparison thus far, and view the results. Result information -- both of a given run through submitted workloads and historical results from previous runs – are stored for later viewing. This information is both given to the application and saved by the application in CSV format.

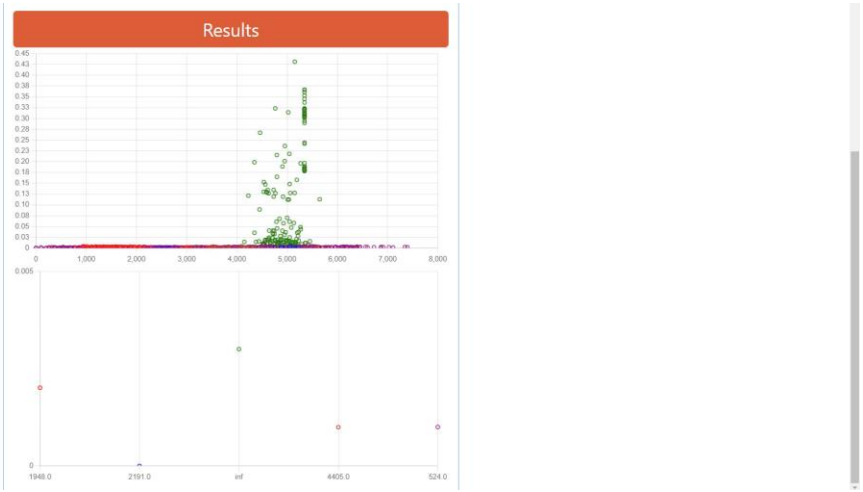

Figure 7: Development Tool Results Page

This figure shows the results of simulated runs on submitted workloads. It can show all of the results saved from every simulated run or be cleared to show only new data. It is visualized in a manner similar to the earlier scatterplot in this report, except the axes are reversed. The y axis represents the time it took the run to complete, and the x axis represents the cost in dollars of the configuration produced by the run.

Main Application

The appearance of the main application has not changed much since we were given it. It operates in generally the same way, except that now it is utilizing our improved algorithm instead of the original. New functionality -- such as data visualization – is currently consigned to our development tool. Front-end code was cleaned up with some unused code being removed but this area was not the focus of our efforts on the project.

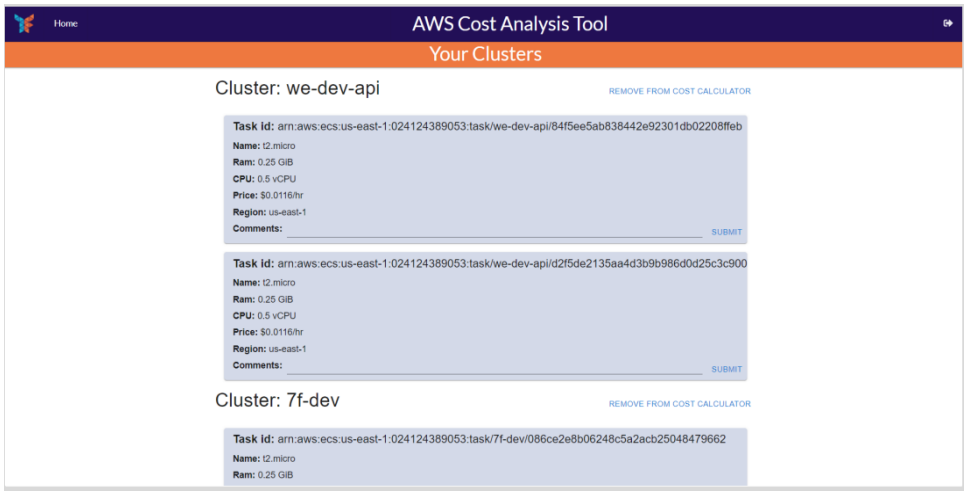

Figure 8: Main Application Cluster Page

Just as before, the main application allows the user to add, view and remove their own ECS clusters. The page displays relevant information such as the name of each cluster, Ram and CPU requirements, and the associated task ID for each container in the clusters.

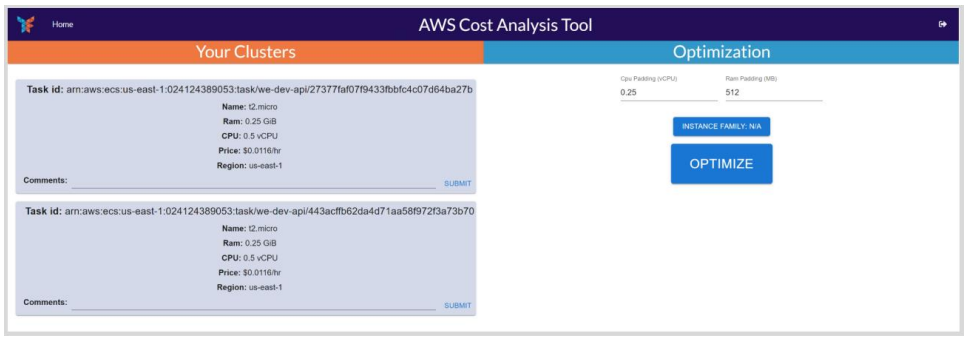

Figure 9: Main Application Optimization Page

It is in this optimization page shown above that the application fulfils its purpose. After selecting an ECS cluster from the cluster page, the ‘optimize’ button will run our new algorithm to produce a cost-efficient virtual machine configuration for the workloads contained in that cluster.

Future Work

There are many potential avenues a future team of students could pursue to improve the project.

Further Algorithm Development

While the algorithm has been very well optimized compared to the previous version, there is always room for further improvements to be made. That being said, the need to do so is lesser because the performance of the algorithm is much more practically viable, whereas the earlier build was far from achieving this.

Configuration Data

By using our tool with a single algorithm and standardized workload, we can produce data about AWS EC2 configurations. This data is ripe for manipulation into a machine learning model to then produce the best possible configurations depending on a varying workload size.

Improvement and Re-Framing of User Interface

The user interface is very simplistic, as it has essentially only been developed to the point of being able to demonstrate the functionality of the algorithm. Visually, the UI leaves much to be desired. Beyond that, the interface itself could be redesigned to be more user oriented.

Performance Evaluation

Data analysis of algorithm performance was a large part of development. Being able to measure the efficiency of the project in a variety of situations and compare the outcomes of differing approaches is vital to understanding what adds value to the algorithm and what is currently lacking. In our case, data was only able to be stored for 100 results of each approach being compared, and not all the information that could be stored in running the algorithm is, which limits the depth that the analysis could take. Therefore, further development in data retention, analysis, and visualization could contribute significantly to the project overall.

Upkeep and Tool Management

As mentioned earlier in the report, part of the code at the beginning of development was deprecated, and part of the setup process was not supported anymore. Given the number of dependencies, libraries, and external tools the project relies upon, in the long term the need for maintenance becomes a necessity, as all of these can lose support or change in their use in a way that will need attention for the project to still function.

Conclusion

We were given a tool developed last year to produce the most cost-effective configurations of AWS ECS clusters to handle potential workloads by 7Factor Development, with the task of improving it beyond a mere proof-of-concept into something practically capable. Using data analysis to compare different strategies, we were able to substantially boost performance, while producing valuable metrics about optimal EC2 configurations. The application now produces more optimal solutions in less time. Additionally, we provided further user options and tools such as data visualization and weights. Work in refactoring the code base has made it simpler and more modifiable in the future in both the front-end and back-end.

References

Thelin, R. (n.d.). *Demystifying the 0-1 Knapsack problem: Top solutions explained*. Demystifying the 0-1 knapsack problem: top solutions explained. Retrieved October 27, 2022, from https://www.educative.io/blog/0-1-knapsack-problem-dynamic-solution

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

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

Bhargav, W. by: N. (2022, August 1). *Recursion and looping*. Baeldung on Computer Science. Retrieved April 20, 2023, from https://www.baeldung.com/cs/recursion-looping

Blakely, C. (2019, March 2). *The Junior Developer's Guide to writing Super Clean and Readable Code*. freeCodeCamp. Retrieved February 15, 2023, from https://www.freecodecamp.org/news/the-junior-developers-guide-to-writing-super-clean-and-readable-code-cd2568e08aae/

Bader, D. (2023, April 1). *Shallow vs deep copying of python objects*. Real Python. Retrieved April 6, 2023, from https://realpython.com/copying-python-objects/

Khan Academy. (n.d.). *Heuristics & Approximate Solutions | AP CSP (article)*. Khan Academy. Retrieved November 13, 2022, from https://www.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/solving-hard-problems/a/using-heuristics

Siapno, N. (2022, October 25). *Everything you need to know about stacks in Python*. Medium. Retrieved November 19, 2022, from https://medium.com/geekculture/stack-in-python-33617350b6d2#1f84

Shivam KD. (2023, February 22). *Static data structure vs Dynamic Data Structure*. GeeksforGeeks. Retrieved March 15, 2023, from https://www.geeksforgeeks.org/static-data-structure-vs-dynamic-data-structure/

Glossary

1. **7Factor -** Our sponsor for this MQP. A software development company that specializes in creating devops solutions for other companies that contract them.
2. **Amazon Web Services -** An Amazon managed platform that hosts upwards of two hundred cloud computing services and features.
3. **Boto –** A Python Software Development Kit for Amazon Web Services. Using two of its packages, Botocore and Boto3, the project is able to manage the AWS related components of the project in Python Code.
4. **Container –** Containers hold a set of applications and software that machines are then tasked with running.
5. **Docker –** A tool that can hold a shared environment in a container that can then serve as a running environment for a project for multiple developers at the same time. This rids the need of these developers to maintain identical local configurations to serve as their own respective running environments.
6. **Dynamic –** A category of data structure that is not of a fixed size. It can change size by allocating and deallocating memory to store its data.
7. **Elastic Compute Cloud (EC2) -** A part of Amazon Web Services that allows a user to rent computation capacity in the form of a variety of virtual machine instances. Instance types have their own amount of memory and virtual CPU cores.
8. **Elastic Container Service (ECS) -** A tool in Amazon Web Services that allows the user to manage cloud containers.
9. **FastAPI –** The Python framework used for this project to develop the back-end of the website that allows for asynchronous processes.
10. **Heuristic - A**nything that allows a function in code to be guided in finding good solutions.
11. **List –** In the context of Python, a list is a mutable array that serves as a dynamic data structure.
12. **Material-UI –** A React UI Library that contains an array of user interface components for React web application development.
13. **Presorting –** In the context of this project, Presorting is to sort data structures before they need to be accessed.
14. **React –** The JavaScript framework used for this project to develop the user interface of the web page. It works as a tool to efficiently create visuals and provide assets for the application.
15. **Stack –** A last-in-first-out (LIFO) data structure where only the most recently added data can be accessed at once.
16. **Static –** A category of data structure in which the size is fixed. Once initialized, Static data structures cannot change the number of entries they can hold.
17. **Tuple –** In the context of Python, a Tuple is an immutable array that serves as a static data structure.
18. **Virtual Machine –** A virtual environment that is allocated the processing power of real hardware to function independently as its own computer.

Appendix

Algorithm Pseudocode

The original Pseudocode as documented by the previous team of developers is as follows:

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

However, given the further development made to the program, the Pseudocode needs to be updated to provide an accurate description of how it currently functions.

The updated Pseudocode describing the current structure of the program is as follows:

```
FUNCTION allocate_vms(vms, workloads):
    # Start timer
    sTime = current time in seconds
    # Sort vms by cost
    vms.sort(key=lambda x: x[3])
```

```
# Create a dictionary to store which vm is allocated to which workload
allocation = {}
# Initialize the total cost to 0
total_cost = 0
# Create an empty list to store the missed workloads
missed = []
# Iterate over each workload
FOR workload IN workloads:
    # Find the least expensive vm that can accommodate the workload
    best_vm = None
    FOR vm IN vms:
        IF vm[1] >= workload[1] AND vm[2] >= workload[2]:
            IF best_vm is None OR vm[3] < best_vm[3]:
                best_vm = vm
    # If a suitable vm is found, allocate the workload to it
    IF best_vm is not None:
        # Make a copy of the vm so we can find it later
        copy = best_vm
        # Update the allocation dictionary with the vm id and workload id
        allocation[best_vm[0]] = workload[0]
        # Update the total cost
        total_cost += best_vm[3]
        # Decrement the vm's resources
        best_vm = (best_vm[0], best_vm[1] - workload[1], best_vm[2] - workload[2], best_vm[3])
        # Replace the original vm with the updated vm
        vms[vms.index(copy)] = best_vm
    # If no suitable vm is found, add the workload id to the missed list
    ELSE:
        missed.append(workload[0])
# Return the allocation and the total cost
# This will return a dictionary of vm id = workload id
```

eTime = current time in seconds

fTime = eTime - sTime

allocation = {k: v for v, k in allocation.items()}

RETURN allocation, total_cost, fTime, missed

Setup Instructions

Experimental Tool:

1. Clone repository locally: (Repository withheld in report at request of 7Factor)
2. In power shell: “pip install flask”
    a. Python 3.9.13
    b. Flask 1.1.2
    c. Werkzeug 2.0.3
3. Run App: “python routes.py “

Full-Tool:

*Note: For more in-depth starting instructions please refer to the following link:

https://digital.wpi.edu/downloads/9593tz49c

1) Clone the repository from https://github.com/tiangolo/full-stack-fastapi-postgresql
2) Create a brand new repository.
3) Download Git.
4) Download GitHub Desktop.
5) Set up the repository for the other members by cloning the repo to a local repository.
6) Install Docker from 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.
8) If there is an issue with the backend being able to properly start up, change the option from CRLF to LF in the prestart.sh file.
9) If the backend still fails to compile, comment out lines #3 & #12.
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 following 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 "aws configure" in the backend terminal, where you will be requested to supply the credentials. d. Currently, for our region, we are us-east-1 and our output is in JSON format.
11) Install Node.js.

12) To run the frontend: a. Change the directory to frontend. b. Run "npm install" (NOTE: first run). c. Run "npm start".