CS 551
Assignment 1: Getting Started with AWS

Link of my web server: http://3.136.154.222

Analysis Section:

1) Cloud computing vendors frequently offer several additional features designed to improve efficiency, security, or any number of usability factors. We saw an example of this in AWS's security groups. Look at the options in the AWS web console. What other useful features does AWS offer?

Security:

i) **Security Groups:** This is something we did in our assignment. These security groups offered by Amazon EC2 are virtual firewalls. They manage outbound and inbound traffic, that can help safeguard and protect our instance. We can add rules that determine which communications can access or exit our instance. When we allowed authorization for HTTP traffic, we modified the incoming rules in such a way that only valid traffic gets in and prevents any other potential malicious traffic. Outbound rules, similarly, regulate traffic that leaves our EC2 instance, ensuring that we have a secure and robust cloud environment.

ii) **Key Pairs:** Amazon EC2 provides key pairs as a security component that ensures safe access to our instances. It has a public and private key pair that serves as a digital signature, authenticating a user's identity. The public key is stored on the instance and the private key is stored directly on the user's machine. This grants safe access to the user, as long as they keep their private key stored securely and eliminate unwanted access.

iii) **Elastic IPs:** Amazon EC2 provides elastic IPs, which is essentially a static IPv4 address adapted for dynamic cloud computing. Unlike standard IPs, which vary when instances and stopped and started, elastic IPs are constant providing trustworthy access points. They stay with the user until he decides to surrender them, personally. In the event of an instance failure, the elastic IP's remap to another instance making sure service disruption is low. Lastly, with elastic IPs and a user's domain DNS record, one can make sure that the domain continuously links to the selected instance providing tractability.

iv) **Network Interfaces:** Amazon offers these as a virtual version of a networking part within a Virtual Private Cloud (VPC). This includes private IPv4 addresses, multiple secondary IPv4 addresses(from the range of VPC), elastic IPs for each private IPv4 address, etc. Apart from the ability of users to create and adjust these network interfaces, AWS also offers requester-managed interfaces, which allow integration with some resources. These cannot be changed directly by the user, making sure the reliability is untouched between interconnected services.

Efficiency:

i) **Auto-scaling:** Amazon EC2 auto-scaling helps in optimizing several instances according to the demand of the application. Users create "Auto Scaling groups" which are groupings of EC2 instances. Within this, they specify the number of minimum and maximum instances for these groups. This helps the number of instances stay within the limits. Also, by deciding the capacity, the service may automatically adapt and launch or terminate instances based on workload. This increases the efficiency of cloud computing vendors.

ii) **Spot Requests:** These are a great way to access the underutilized resources of EC2 in the AWS cloud. These instances are discounted up to 90% compared to the usual On-Demand rates, making them the ideal choice for applications that can tolerate a few interruptions. Companies can better manage costs and scalability since customers can choose when to hibernate, halt, or stop their spot instances, depending on their product life cycle and when EC2 reclaims maximum capacity.

iii) **Reserved Instances:** These reserved instances offered by AWS EC2 provide a big discount on instances for applications that have consistent operations and workload or long-term commitments. Users can reserve capacity for their selected instance by committing to a set period between 1 to 3 years. Also, these reserved instances offer the customer the opportunity to change instance size, etc.

Usability:

i) **Instance Types:** Amazon EC2 has a variety of different instance types that it offers. They are custom-tailor-made to fit the needs of various applications. Different applications need different requirements, so it gives the user to choose from a combination of CPU memory, storage, and network permutations. Also, each kind is split into different sizes, allowing granularity which depends on the project load.

ii) **Launch Templates:** Launch templates are blueprints for launching instances, which comprise AMI ID, key pairs, etc. The main reason we use this is because of its ability to keep using copies of the same template. With version control, we can build configuration options. This option increases efficiency since users can add or remove versions as per project needs.

iii) **AMIs - Amazon Machine Images:** EC2 offers this as a template for setting up virtual servers. Each AMI has custom settings like OS and app sets which are essential for launching the EC2 instance. It also has Amazon EBS (Elastic Block Store) snapshots or patterns for root volumes. It also has a mapping for each associated volume which increases usability by providing streamlined deployment and consistency.

2) In class, we learned about virtualization techniques frequently employed by cloud service providers. How do these relate to the t2.micro instance you just created?

In class, we spoke about virtualization where we can make servers that appear to be the same as a physical server. As mentioned in class, "it means that multiple tenants can use the same "physical" server without "leaking" data."

We learned about **Virtual Machines and Cloud instances**, which help in achieving this.

With virtual machines, we can emulate a whole software system as a service, which safely coexists on a physical server. This offers isolation which is very useful. In terms of our t2.micro instance that we created, it is a specific virtual environment that we created that is hosted on physical AWS servers. It offers isolation since it's independent and gives a dedicated virtual environment to a user. The physical computer is called the "host" and virtual machines are called "guest machines"

How this works is through **Hypervisors.** These hypervisors help in managing multiple virtual machines so that they stay independent. There are 2 types: Hypervisors Type 1 and Hypervisors Type 2. Type 1 runs directly on computer hardware making it fast and Type 2 runs as an application, which helps in running multiple instances on a single machine.

In relation to our t2.micro instance, the Type 1 hypervisor manages our instances' virtual environment. It helps fetch the CPU power, memory, storage, etc. from the physical server and allocates it to our instance. It also makes sure that our t2.micro instance is isolated from other instances, providing safety measures.

3) Under the ”Description” tab of your EC2 instance on the AWS web console, there are two fields: “IPv4 Public IP” and ”Private IPs”. What is the difference between these two IP addresses? Which one shows up when you run iconfig on your instance? Why is that?

IPv4 Public IP:
This is the IP that our instance uses to communicate with the internet. It is routable to the public internet and if we need to access our instance outside of our VPC, we use this IP address. Essentially it changes every time an instance is stopped or started, unless we use an Elastic IP, as we mentioned before. We can also, like in our assignment, use a screen tool, such that our virtual terminal environment doesn’t cease to exist when we stop our SSH client. This also helps in having a persistent public IPv4 address.

Private IP’s:
This is the IP address for our EC2 instance made within the VPC. This is primarily used for communication between peered VPC’s or resources within the same VPC. This IP cannot be accessed from the public internet since it’s private to the AWS environment. One has to go through a NAT gateway or something similar, so the private IP can communicate with the NAT, and following that, the NAT will make its public IP with which we can communicate to the internet. Also, this private IP will stay constant for the whole lifetime of the EC2 instance, unless we decide to change it manually.

When we run “iconfig”, we will see the “**private IP**” of our EC2 instance. This is because we are essentially communicating with the instances’ operating system. It shows the OS's configuration and details and network statistics. In the case, of our EC2 instance, we are associated with the private IP that the OS recognizes, hence we see that. The public IP on the other hand is managed by AWS infrastructure and is outside of our instances’ operating system.

4) You’ve launched a simple web server and updated the code to the EC2 instance. In a production environment, how would you deploy updates to the server while maintaining high availability? If the deployment failed, how would you roll back?

In a production environment, I would deploy updates in a calculated manner so that I maintain high availability. Below are some of the most important steps I would follow:

i) Use Load Balancing for updates:
Load Balancers distribute traffic across multiple instances to maintain high availability. In a production environment, load balancers can be very useful for rolling out updates since they can identify healthy instances and map traffic to them rather than unstable or highly resource-consuming instances.

ii) **Use Auto-Scaling for updates:**
Auto-scaling automatically modifies several instances based on usage, as we saw before. When we roll out updates, it might take more resource usage and some instances might get unstable or have network congestion. With auto-scaling, we can compensate for this, by automatically assigning extra instances.

iii) **Rolling Updates:**
With this, a certain number of tasks or instances are pulled out, updated, and restored, until all are updated. This helps while deploying updates since it does not expose all changes at once. This optimizes the availability of a server and makes it less vulnerable to deployment crashes.

iv) **Blue/Green Deployments:**
This is a powerful method to have efficient updates. "AWS CodeDeploy" lets us use these blue/green deployments to roll out safe updates. The blue environment retains the current version while all the updates are deployed and tested on the green environment, which is a copy of the functioning blue environment. Once the traffic and the instances are stable, traffic is shifted from a blue to a green environment, hence ensuring safe transfers while updating.

We can use a combination of these methods in our production environment, to make sure we deploy safe updates to the server while maintaining high availability.

If the deployment fails, or any of the steps above don't work as expected we would need to roll back the functioning update back online. For that, I would use:

i) **AMI's:**
Amazon Machine Images are a powerful tool that helps us roll back an update if something doesn't function as expected. Before deploying an update, I would create an AMI of the established state. In case of a deployment failure, I would launch new instances using these AMIs to revert to the previous version.

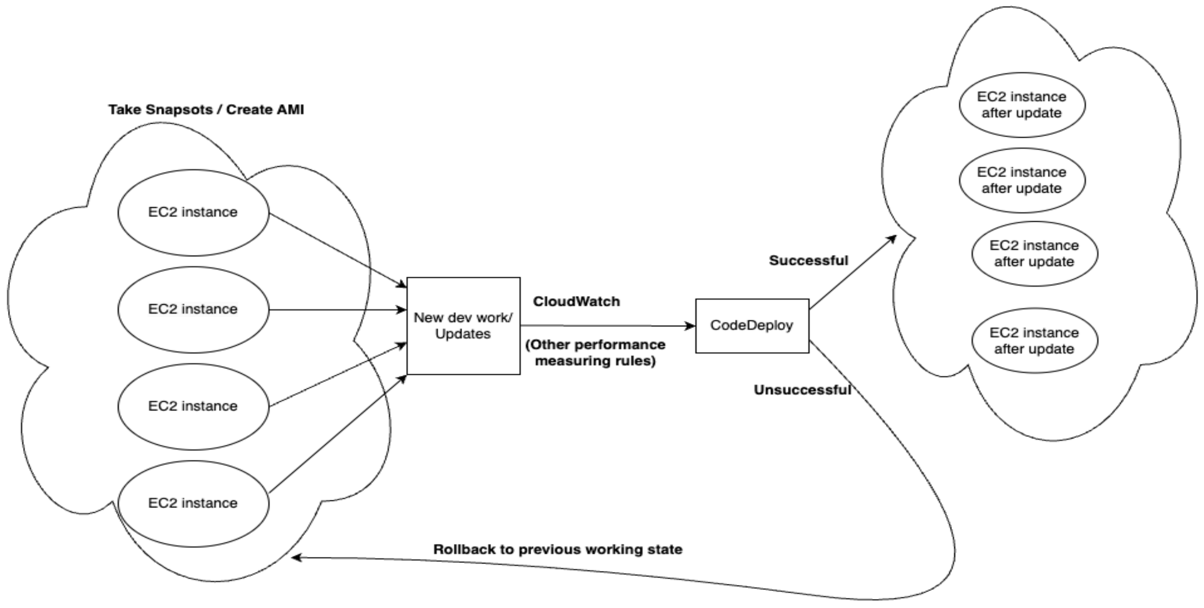
ii) **Load Balancer:**
The Load Balancer can detect when the program or instance is being congested or unresponsive and reroute traffic to safer instances. As we saw above, with the blue/green zone; if the green environment fails, the load balancer switches traffic to the blue (functioning) environment making sure we have high availability.

iii) Auto-Scaling:

Similarly, with auto-scaling, when updated instances get unhealthy, auto-switching groups can replace the updated instances with the previous AMI's of the original instance. This can also be done automatically, and we don't need manual presence for this. Automating this process ensures low downtime and high availability.

iv) Snapshots

We can take snapshots of our instances in the working state. These snapshots are essentially point-in-time backups. When we deploy an update and it doesn't go as expected, we can roll back our updates using these snapshots. This also ensures the integrity of the data and safe retrieval of the previous(functioning) instance.


Take Snapsots / Create AMI
EC2 instance
EC2 instance
EC2 instance
EC2 instance
New dev work/ Updates
CloudWatch
(Other performance measuring rules)
CodeDeploy
Successful
Unsuccessful
EC2 instance after update
EC2 instance after update
EC2 instance after update
EC2 instance after update
Rollback to previous working state

This is an overview of the strategy that I will use to deploy/roll back updates. Initially, if I have multiple instances or a cluster of instances for an application, I will take snapshots or create AMI's of the working condition. Then I will use CloudWatch or any other monitoring tools to supervise performance metrics and verify if the update or any dev work done on the existing application is compliant with the standards. If the deployment is successful, I will then use AWS CloudDeploy to create a new quartet of updated EC2 instances. If we have any failure while making this update, I will have a system-wide rollback to restore the pre-update state of the EC2 instances.
Note: I can use the green-blue deployment strategy in this as well. I will test on the green environment and measure the performance. If it works out, I will deploy the green environment. If not, I will restore the initial blue working environment to roll back the update.

5) Let’s say you work at a large investment firm that your TA worked where clients (6 million retail customers) can view their stock portfolios through a web app interface. During periods of market volatility i.e., the start of COVID-19, your clients all log in at the same time. How would you scale from one simple web server to many? What are the possible trade-offs in the technique you proposed?

I would use these techniques to combat the sudden rise in network congestion when scaling from 1 web server to many:

i) Auto Scaling and Load Balancing:

As we saw before, auto-scaling helps with increasing or decreasing resources based on preset parameters or real demand. In the context of a stock portfolio web interface, we would combine this with a load balancer to spread the traffic over an automatically adjusted number of instances based on demand. If there are a lot of people logging in, we would dynamically allocate more instances and reroute traffic equally among the instances so none of them get congested causing delays or service interruptions. This can also help with saving money as when the demand is low, it drops some instances and de-provisions them, until needed.

ii) Caching:

Caching helps in scaling from one to many servers at peak times since it makes sure we have quicker response times for users. If many users want to view their stock portfolio or examine a certain stock, the database queries every time which increases time and causes delays. Repeated requests are cached so that we reduce the load on the server and the database. This also helps new instances in a scaled environment provide faster responses to queries and since stocks change every minute, computing it in real-time is a very computationally heavy task. With caching, we can make it a tad easier and give users a better experience even at peak times.

Trade-Offs:

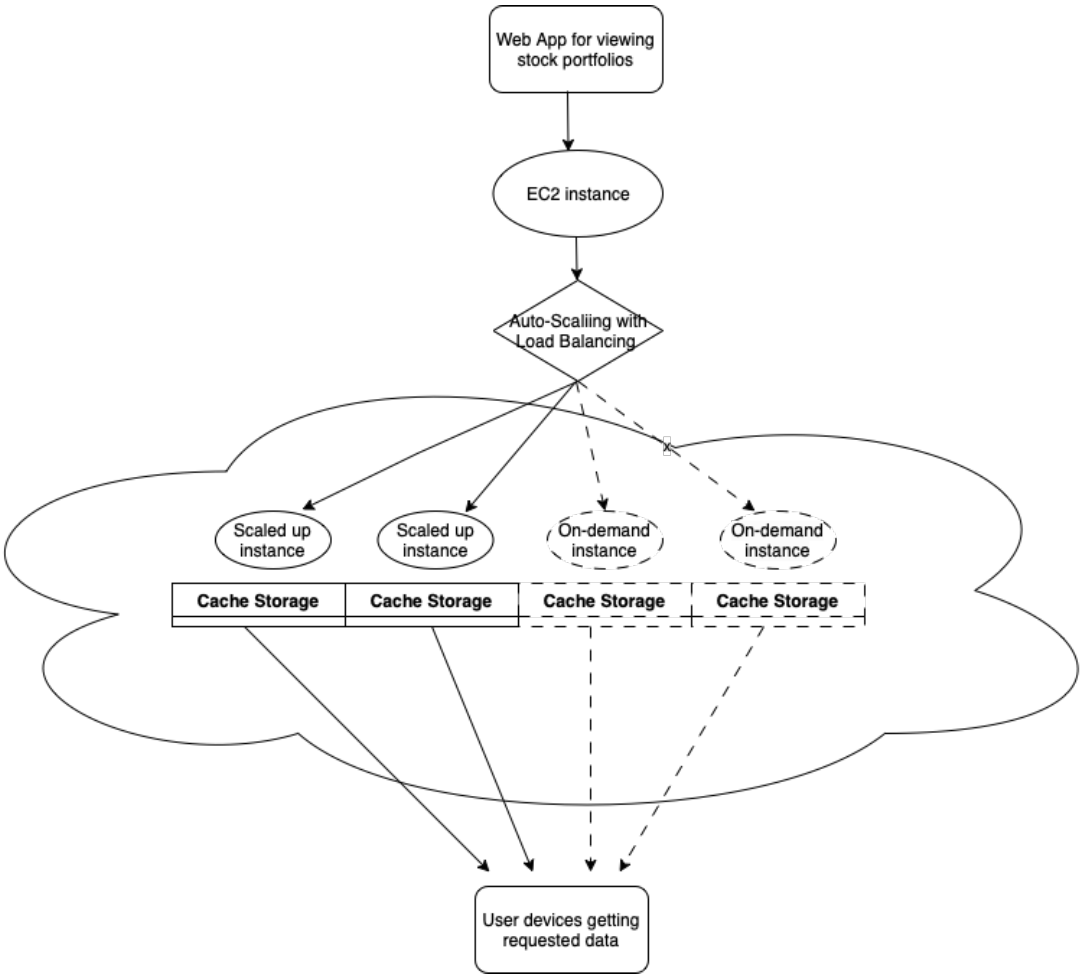
With auto-scaling and Load Balancing:

- We can inflate costs if we don’t monitor our instances and use more instances than needed.
- New instances might take time to start and initialize causing some delays in the beginning.
- In some cases, the load balancer itself might need a scaling strategy if there is a bottleneck in routing traffic

With caching:

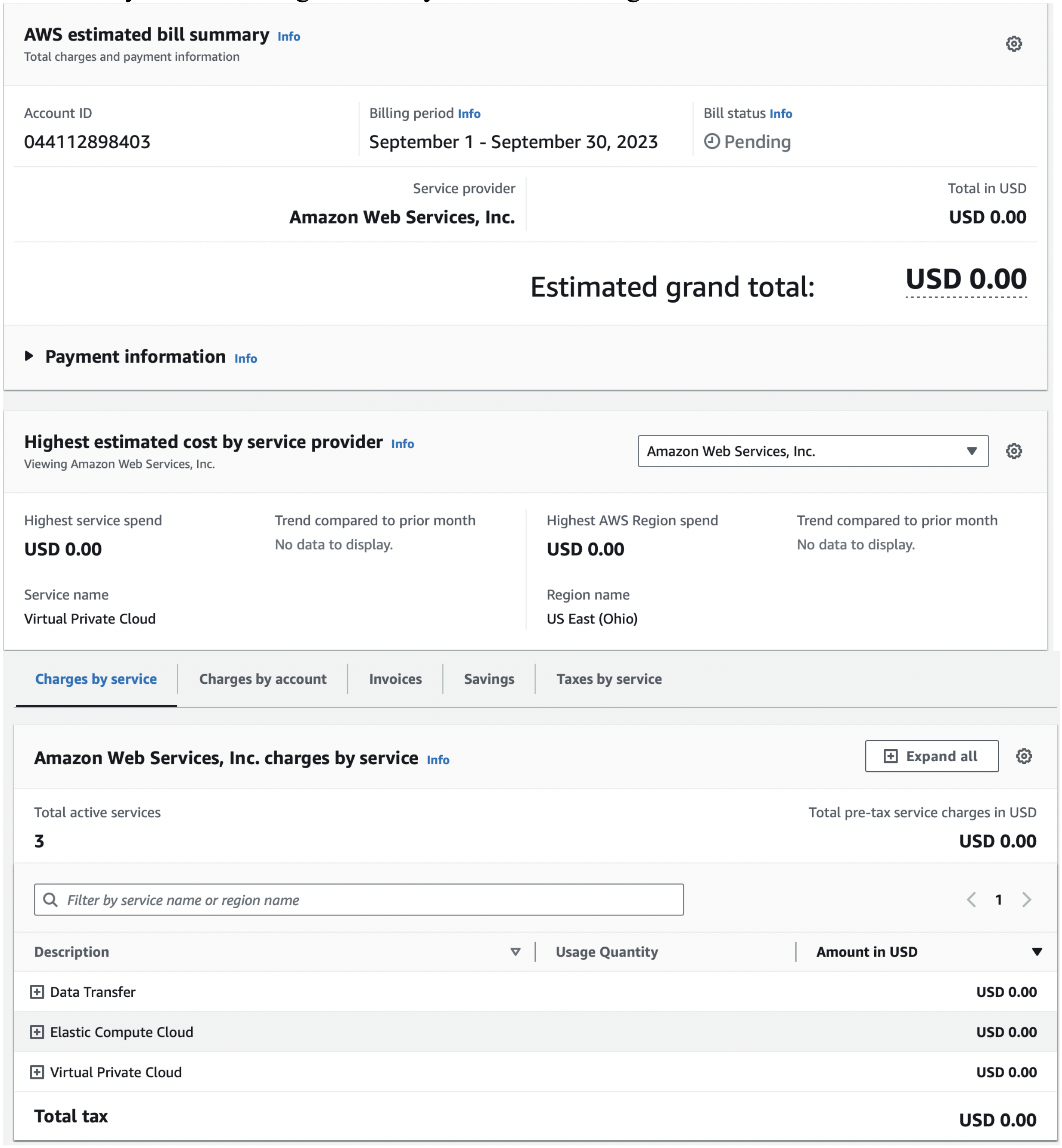
- Caches might reproduce outdated information, if not refreshed regularly. This can be an issue for stock trading platforms since every minute matters.

- Since caches have memory limits, we need to make sure the important information isn't deleted every time to make space for new information that might not matter as much.
- The initial query might take longer since the cache is empty, so it has to map to the original database instead.



This is my initial approach to scaling up servers. Initially, we have 1 EC2 instance for hosting an application that lets users view and edit their stock portfolios. We use auto-scaling with load balancing as mentioned above, for creating multiple instances on demand. If there is more demand, we add more instances (dotted ones) and remove instances when demand is not that high. Load balancing helps distribute network traffic equally, so no instance gets congested. Furthermore, we introduce cache storage to increase the response time of requests. The requests get stored in a cache, and when a repeated request comes, instead of querying the database, we query the smaller cache. When we fetch the results, we send it back to the user. This helps in scaling effectively while keeping time and money as constraints.

6) Include a brief description and picture of how much your web server costs. This is so you remain cognizant of your free tier usage.



AWS estimated bill summary Info

Total charges and payment information

Account ID	Billing period	Bill status
044112898403	September 1 - September 30, 2023	Pending

Service provider	Total in USD
Amazon Web Services, Inc.	USD 0.00

Estimated grand total: **USD 0.00**

Payment information Info

Highest estimated cost by service provider Info

Viewing Amazon Web Services, Inc.

Amazon Web Services, Inc.

Highest service spend	Trend compared to prior month	Highest AWS Region spend	Trend compared to prior month
USD 0.00	No data to display.	USD 0.00	No data to display.
Service name: Virtual Private Cloud		Region name: US East (Ohio)	

Amazon Web Services, Inc. charges by service Info

Total active services: 3

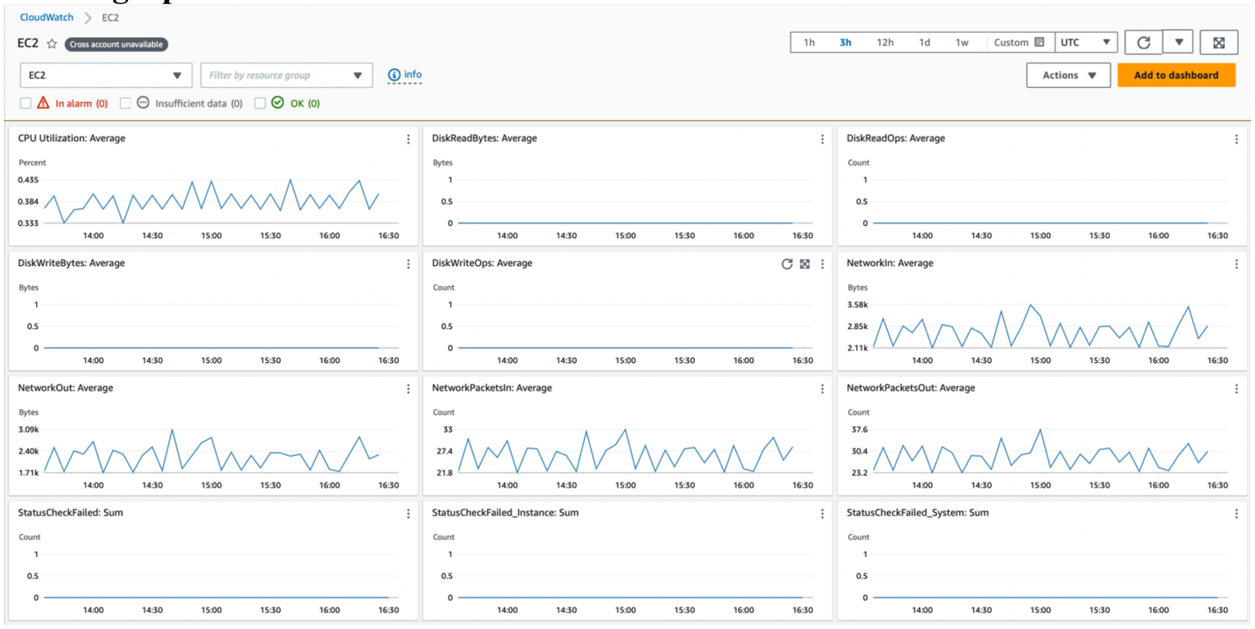
Total pre-tax service charges in USD: USD 0.00

Description	Usage Quantity	Amount in USD
Data Transfer		USD 0.00
Elastic Compute Cloud		USD 0.00
Virtual Private Cloud		USD 0.00
Total tax		USD 0.00

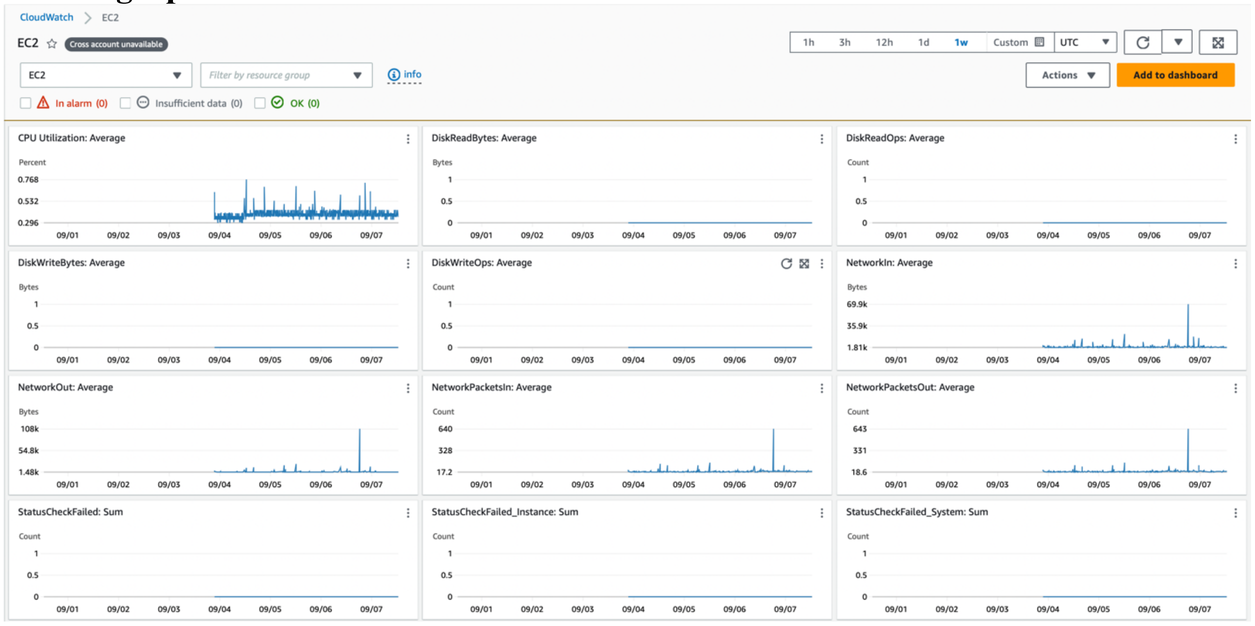
My total cost of running the EC2 instance is 0 dollars. This is because I'm running on a free tier provided by AWS. This includes 750 hours per month of running t2.micro instances(dependent on region). At the same time, it's crucial to monitor this if we are exceeding the limits. But since there are an average of 730 hours in a month, we should not be billed since the free tier offers 750 hours per month. This means I can run my t2. micro instance and let it be active for a month, with my costs being $0. It also offers 5GB of Amazon S3 standard storage.

Observability:

3-hour graphs:



1-week graphs:



These graphs are pivotal to see our web traffic and latency. I have also included the graph for average CPU utilization to make sure we are not going out of bounds. These help us identify what the average network package time is to get in and out of our EC2 instance. This can be seen in NetworkPackageOut and NetworkPackageIn graphs.