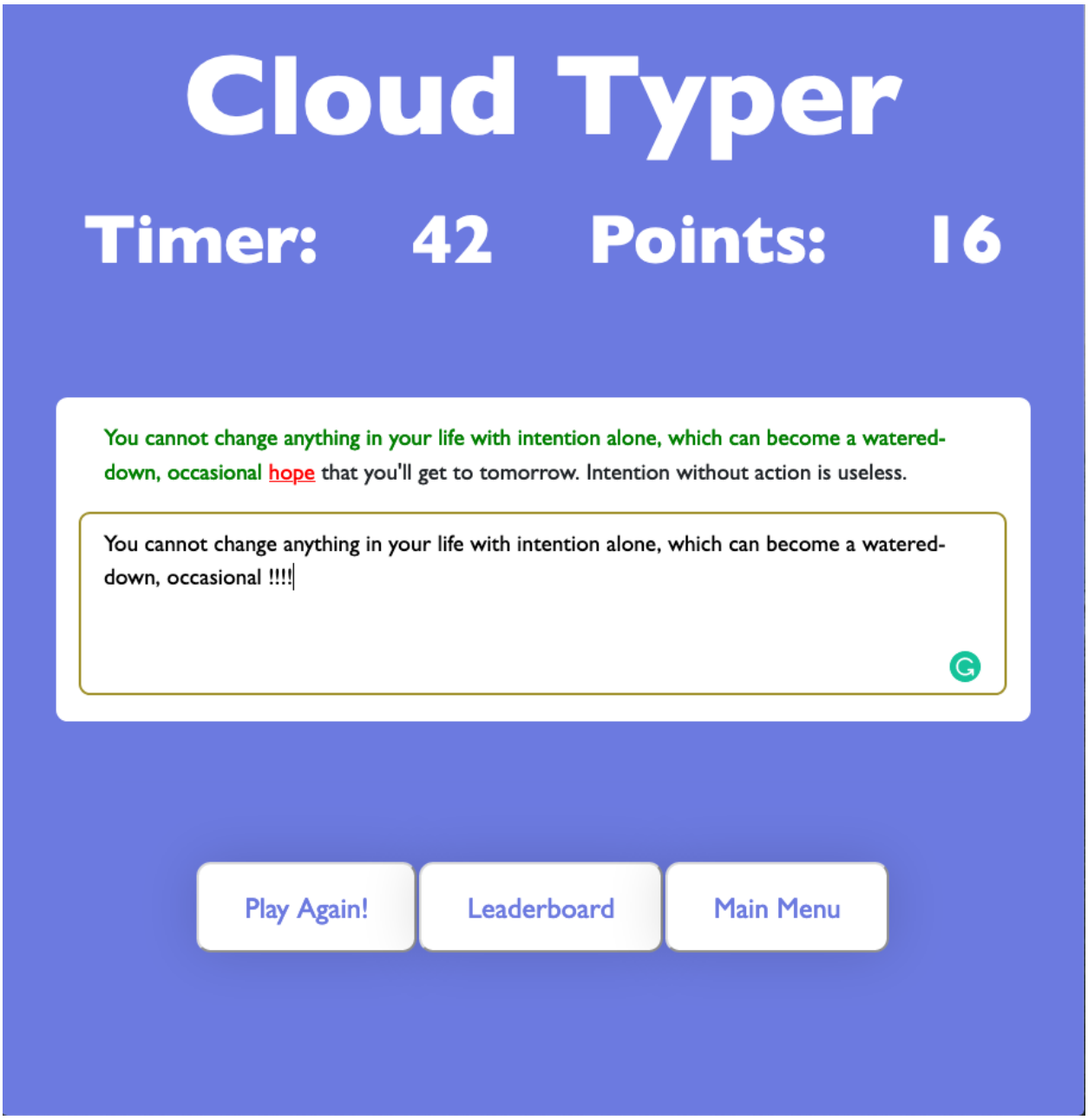
Lexun Chi, Jun Zhen, Pratit Vithalani

Cloud Typer

Program 5 Report



Description

Cloud Typer is a simple typing game that allows users to compete with friends or anyone in the world. Users will have 60 seconds to type and the final score will be all the words the user successfully entered. This score will automatically be passed to our server and get recorded in our database. Besides playing the game, users can view their personal best or visit the leaderboard to see the top 10 faster users. Our application is built with the help of AWS Elastic Beanstalk, Cognito, CodeCommit, CloudFront, CloudWatch, DynamoDB, SNS, S3, Quote API, and our player server. By utilizing these web services, we were able to implement login/registration, authentication, and notification.

URLs

Web application - http://cloudtyper.s3-website-us-west-2.amazonaws.com/
Our server - http://cloudtype.us-west-2.elasticbeanstalk.com/

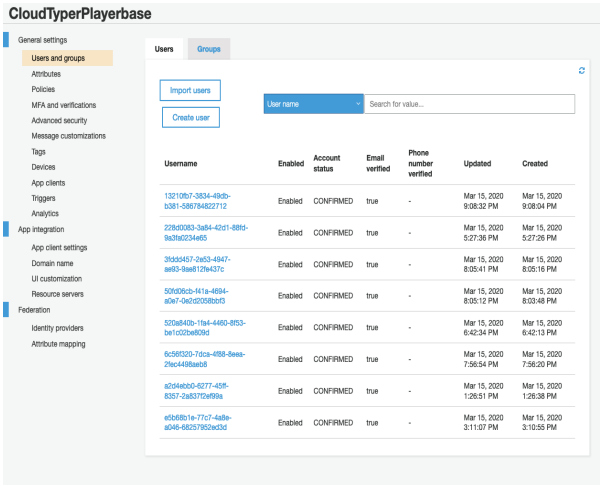
- Leaderboard- http://cloudtype.us-west-2.elasticbeanstalk.com/leaderboard/top10 -
- Profile - http://cloudtype.us-west-2.elasticbeanstalk.com/profile/?email=

How to Play!

1. Visit http://cloudtyper.s3-website-us-west-2.amazonaws.com/
2. Register with an email and password
3. A verification email will be sent, click the link to be verified
4. Login with registered email and password
5. At the main menu page, click on the “play game” button to start playing!

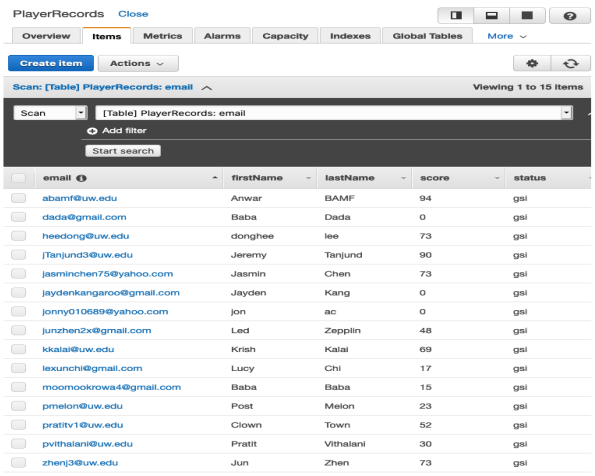
List of Services Used (10)

AWS Cognito



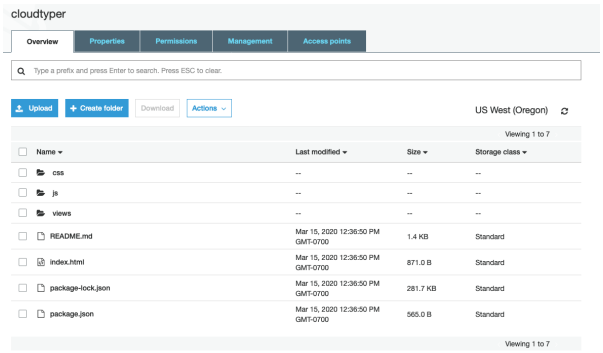
Allow users to register and login. Users must be authenticated by AWS Cognito and obtain an access token to be able to play our game.

AWS DynamoDB



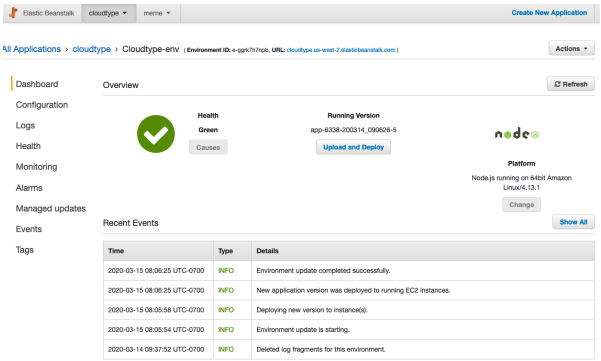
Users registered with us will be stored in our database with a base score of 0. Whenever they play the game, their score will be updated here.

AWS Simple Storage Service (S3)



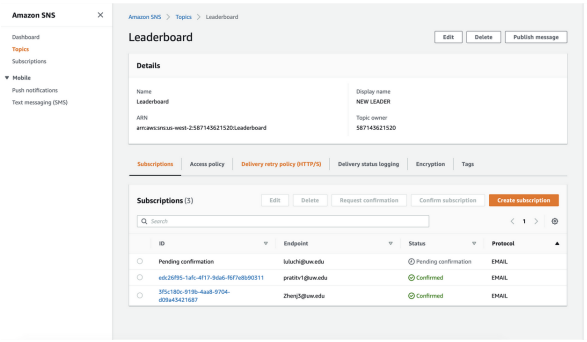
Uses S3 to store our data and host our web pages. The static webpage will communicate with our server and other API to become a dynamic web application.

AWS Elastic Beanstalk



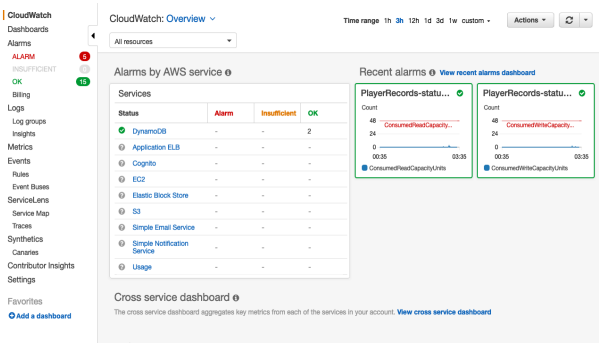
Utilize Elastic Beanstalk to host our server so that we can turn our server into an API. People hit our endpoints and return information about our player base.

AWS Simple Notification Service (SNS)



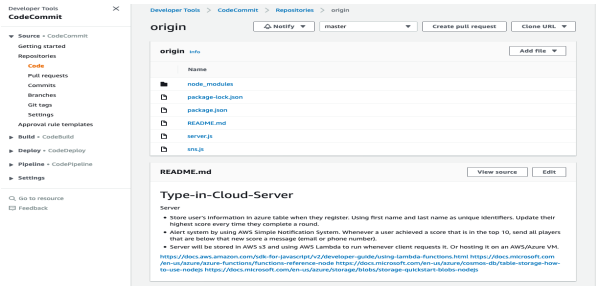
Add a notification system to our application so that whenever there is a new champion, our player base will be notified through emails.

AWS CloudWatch



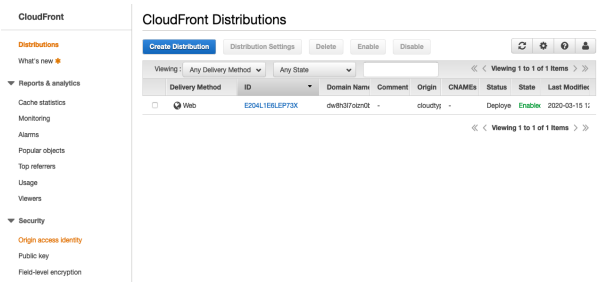
Monitor database and server activities. Add alarms that will notify us if any anomaly occurs..

AWS CodeCommit



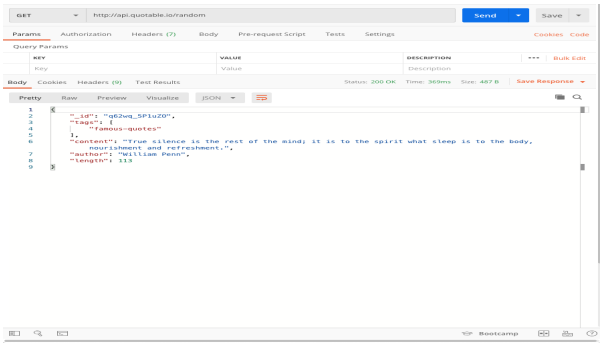
Repository for our project. CodeCommit allows us to store our project in S3 and deploy to Elastic Beanstalk. Achieved continuous integration and continuous development.

AWS CloudFront



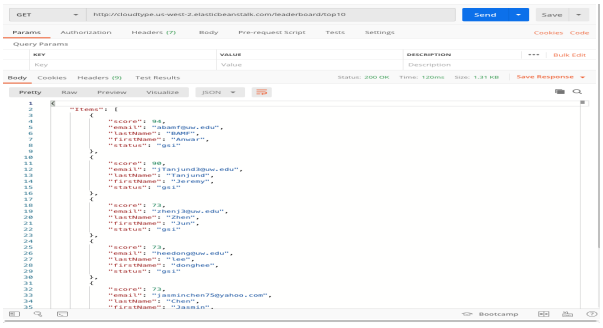
Allow us to copy, backup and scatter the client web page across multiple regions. Increase the durability and availability of our web application.

Quote API



Generates a new and random quote each time. Use for the user to type against.

Server/Playerbase API



Server endpoints that generate a JSON formatted that contains information about users.

Design

The system as a whole can be broken down into three different views. Figure 1 shows the view of the system from the user's point of view. Figure 2 shows what the system does when a new user logs in. Figure 3 goes over the logic done with holding player high scores in the AWS DynamoDB.

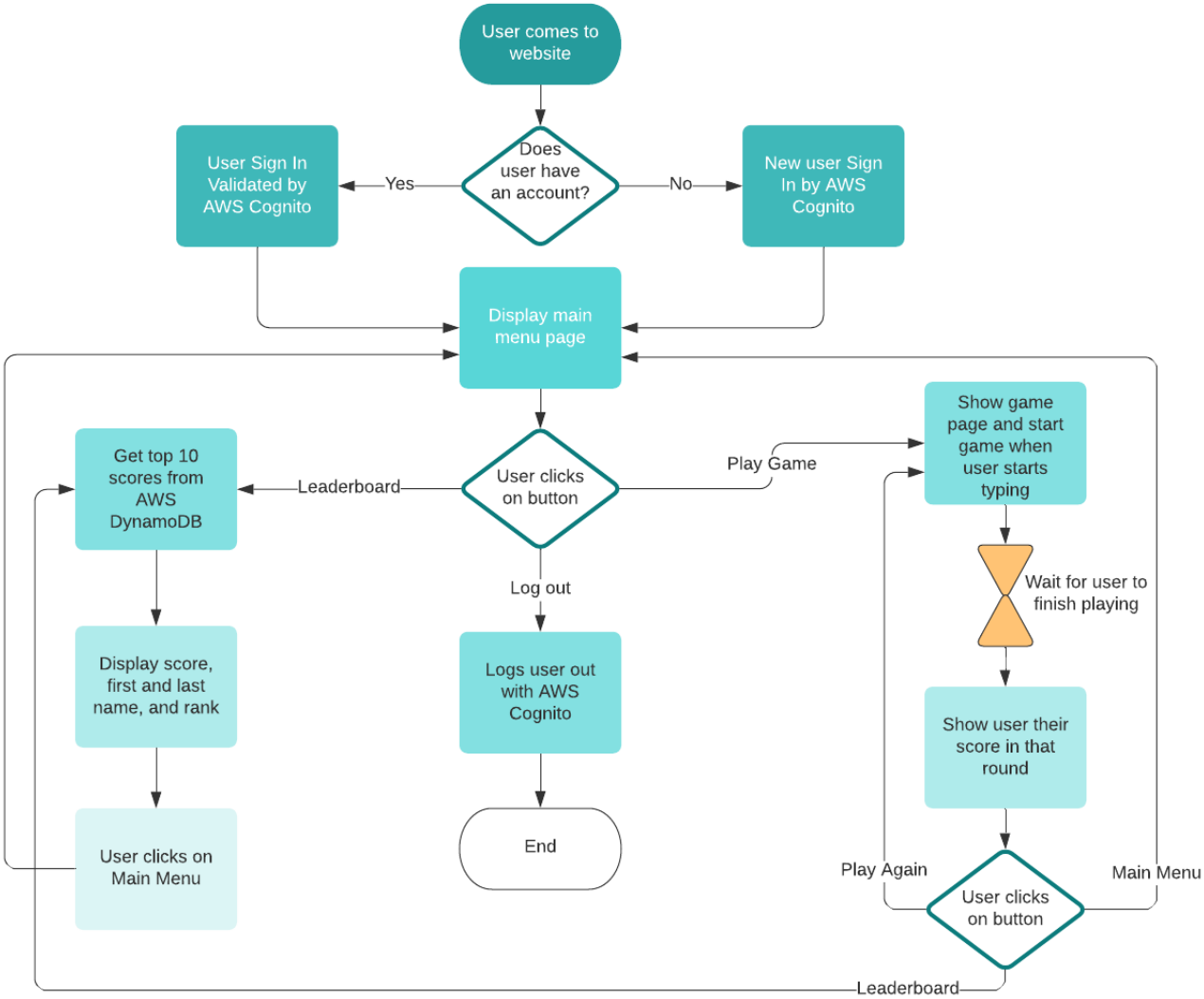


Figure 1: Flow chart of user view

The process of the user in the game starts with them signing in or making a new account. When signing in, all they have to give is their email address and their password. When signing up as a new user, they have to enter their first and last name, email address, and password, which they enter twice to confirm. Then, the user is shown on the main menu screen.

On the main menu, there are three buttons: play game, leaderboard, and log out. If the log out button is pressed, the user is signed out and given the logout page. If the leaderboard button is pressed, the top 10 scores from the AWS DynamoDB are retrieved and displayed, along with a button to go back to the main menu. If the play game button is pressed, the game page is shown after a quote is received from the quote API. Every time the user enters a quote correctly, a new quote is fetched. After that, the user's score is displayed and three options are given: play again leaderboard, and main menu.

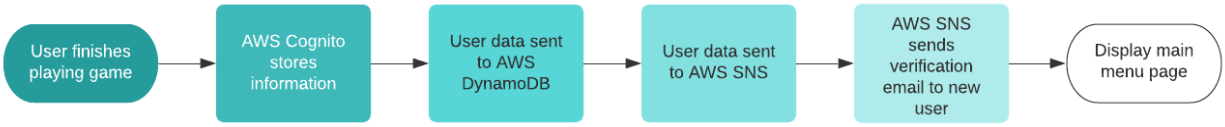


Figure 2: Flow chart of back end when a new user signs up

When a user is signing up for our system for the first time, AWS Cognito adds them to its data store. However, that information is also needed for the game because we have to also store the best scores of the players. So, the data is gathered from AWS Cognito and added to the AWS DynamoDB table. It is also added to AWS SNS, for sending notifications to users about the record being broken. For both AWS Cognito and AWS SNS, user verification is needed, so an email is sent from each service. After they verify and log in, the main menu is displayed.

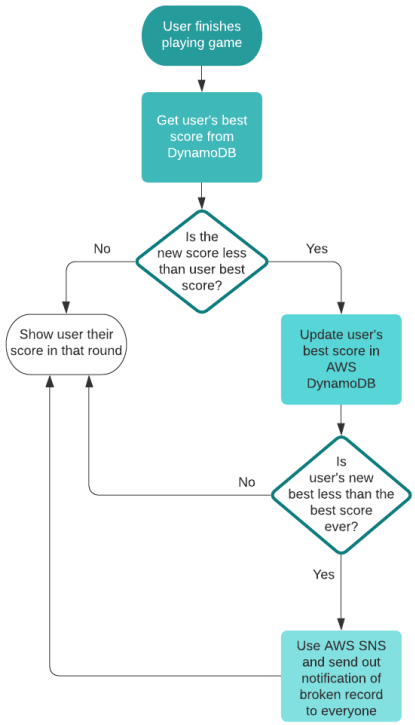


Figure 3: Flow chart of back end when a user completes a game

After the user finishes their game, their best score may need to be updated. The system first gets the user's best score and compares it to the new score. If it is less than the previous score, the new score is updated to become the best score. Then, another check is done to see if the overall highest score was changed. If it was, AWS SNS sends out a notification to all of the players saying that the record was broken. After all of this, the user's score is displayed to them.

Why We Chose Amazon Web Services

The decision to use Amazon over other cloud services such as Azure or Google Cloud is the ease of use of Amazon Web Service. AWS was built with a mind for the users. This shows with their management console being very user-friendly. The documentation for their service is very rich in content. They support many programming languages/frameworks such as java, javascript, python and more. Other providers such as Azure are more business-friendly and built for corporations.

The main decision for us to choose AWS is that their services work very well together. AWS's version control CodeCommit allows us to back up our codebase to S3 and then utilize S3 to host our web page. Also, through CodeCommit, we were able to deploy our server directly to Elastic Beanstalk with just a couple lines of commands in the CLI. Not only does AWS allow us to easily deploy our application, but it can also help us establish alerts and notifications through their CloudWatch service. This allows us to be able to maintain by monitoring all of our application's components within one place. By strictly using AWS, we do not need to hide any credentials such as secret keys. Since we are only using Amazon's services, our authentication is already done through just one simple sign-in check.

Monitoring System

Since our system is run through AWS, we can see metrics on the services that we are using. Figures 4, 5, and 6 show the console view of how AWS Elastic Beanstalk and AWS DynamoDB appear to us. Figures 5 and 6 show the view from AWS CloudWatch, which is another service provided by AWS to monitor other services.

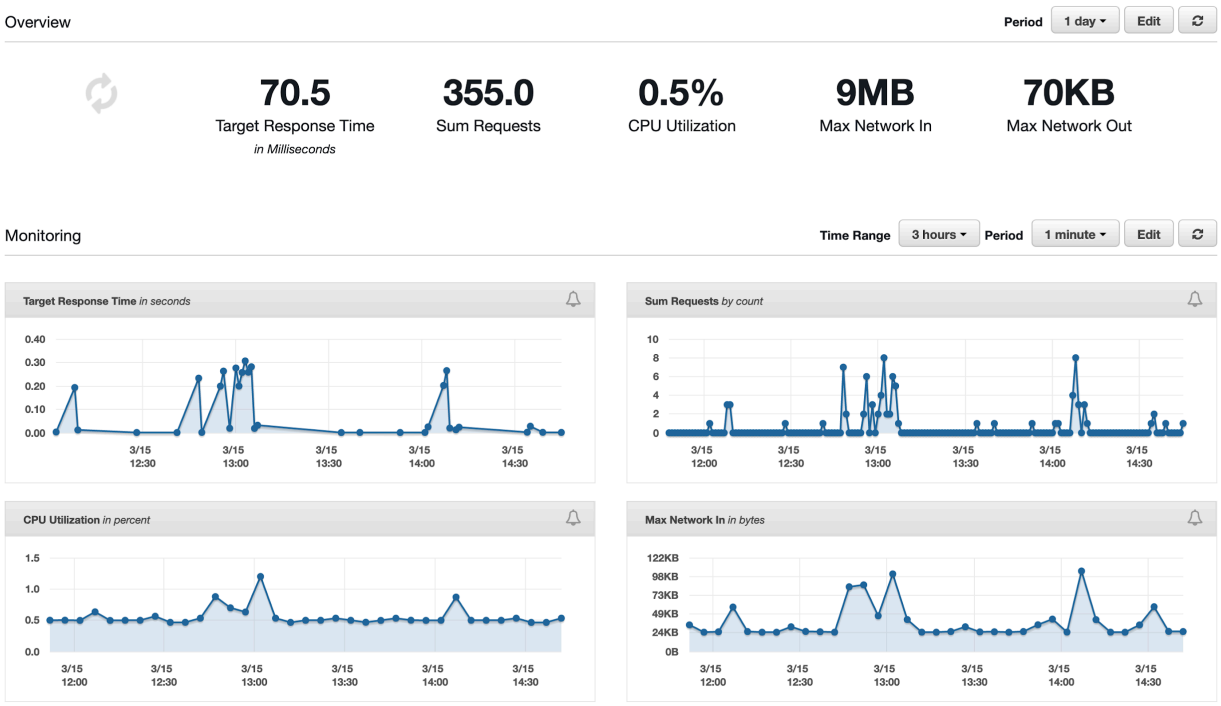


Figure 4: Elastic Beanstalk

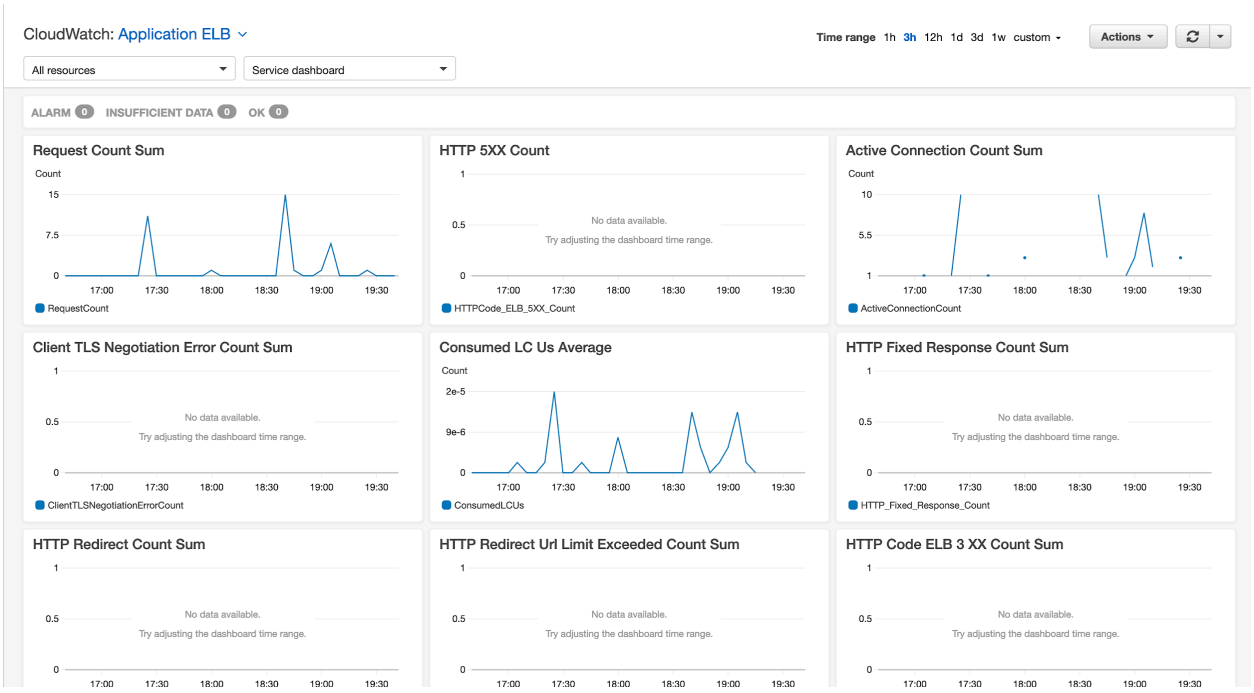


Figure 5: Elastic Beanstalk on CloudWatch

Figures 4 and 5 showed the health and usage of our AWS Elastic Beanstalk, which is used to host the website. This shows us how many times the website is being accessed and how it is working with other services we are using.

Availability is monitored by setting alarms on specific attributes related to the website. One of the values that we look at is the target response time for the AWS Elastic Beanstalk. This is important to keep track of because it shows potential problems, such as a slow network or a bottleneck elsewhere in the system. Another value that we keep track of is CPU utilization under the AWS EC2 instance that the website is running on. If the utilization is at zero, the website is down, and AWS should automatically be moving the website over to another AWS EC2 instance. A third alarm that is set is for the server/API, as that is a service we made and rely on, to see how many reads and writes are being made. If the value exceeds the threshold of 48, we are notified and we will allocate more resources to make up for the increase in traffic.

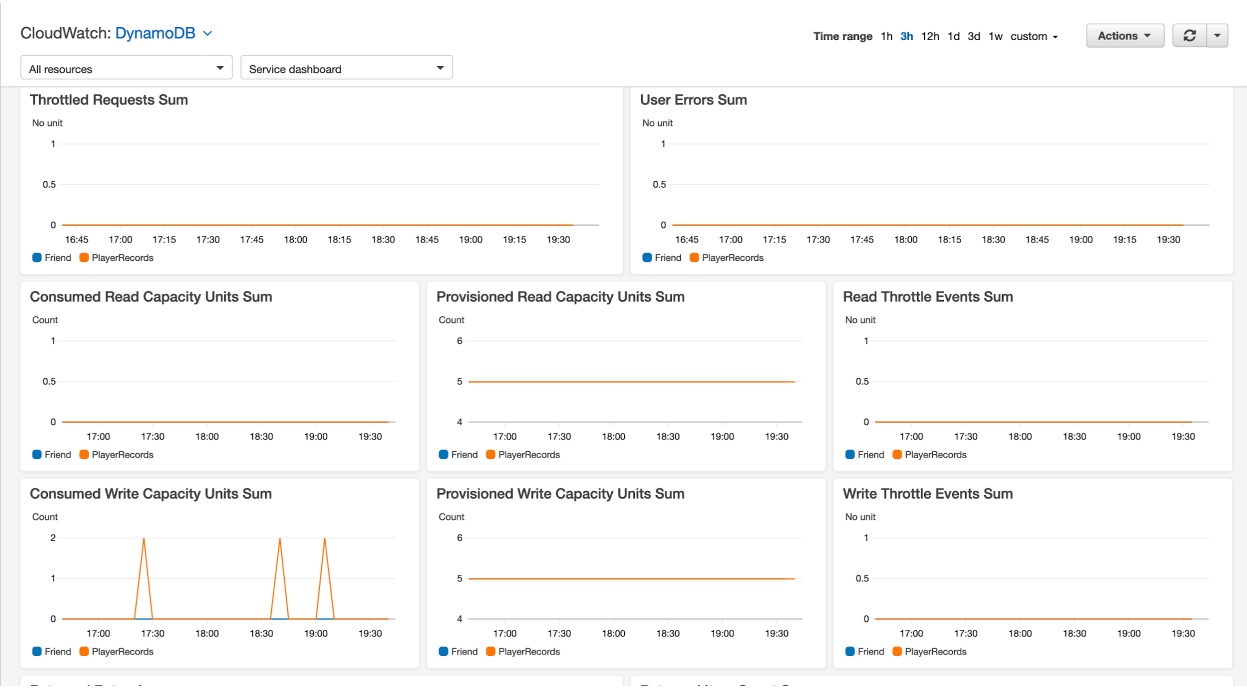


Figure 6: DynamoDB on CloudWatch

Figure 6 shows the AWS DynamoDB view from AWS CloudWatch. Here, our thresholds are on the User Errors Sum and Throttled Requests Sum. User Errors Sum shows any errors that occur from our code writing to the database, which we have in place to catch edge cases that we didn't encounter during testing. In other words, it is a last-resort safeguard against weird client inputs. Throttled Requests Sum raises an alarm if there is too much traffic going to the database for the database to handle. Since it is becoming the bottleneck of the system, we may slow down other parts of the system to allow the database to be written to without overloading it.

All of these alarms will send us an email stating the issue that is going on, giving us the ability to look into it and fix it ourselves. The reason why we want to do the fix by ourselves is that the fix may be code or other measures we placed on our AWS usage. If the problem is related to code,

we have to fix it anyway. However, if the problem is being caused by the limits we put on our AWS account, we have to judge how making any change will affect how much we pay AWS.

SLA

Although our application utilizes a total of 10 unique services, not all of them are hard dependent. Services such as CloudWatch, CloudFront, and CodeCommit are more like tools that assist us with monitoring and deployment. If they go down, our application can still be 100% functional. The services that our application requires are Cognito, DynamoDB, S3, and Elastic Beanstalk/Server. Without these services, users will not be able to access the application at all. The set of services our application relies on but not required to be up are SNS and Quote API. These services enable the application to have more features but are not required to be functional.

H - hard, M - medium, and S - small dependant

Standard SLA:

- Cognito - 99.9% (H)
- Dynamodb - 99.99% (H)
- S3- 99.9% (H)
- EB/server - Not given, assuming: 99.99% (H)
- SNS - Not given, assuming: 99.9% (M)
- Quote API - Not given, assuming: 99.9% (M)
- ~~CloudWatch - (S)~~
- ~~CodeCommit - (S)~~
- ~~CloudFront - (S)~~

(Cognito - 99.9%) * (Dynamodb - 99.99%) * (S3 - 99.9%) * (EB - 99.99%) * (SNS - 99.9%) * (Quote API - 99.9%)

Total SLA of Cloud Typer: **99.58%**
We are omitting the services that are small dependant because our applications do not require those services to be up for our application to be available and functional.

Why the SLA is Achievable

This SLA is achievable for our system because the services that we have a hard or medium dependency on are available for that SLA. We are not doing anything significant on our end in terms of special compute or providing a service and posting it to an API endpoint. We are simply

using the cloud services provided by AWS and the quote API, so our system is only reliant on those services being up.

How the System Will Scale

If there is an increase in demand for our system, we will need to scale it to allow for more users to play our game. One service that we will have to scale is AWS Elastic Beanstalk. This can be done by setting the minimum and maximum number of instances that can be used. AWS will automatically add and subtract instances depending on the load that the current amount of instances is under. A similar setting can be set on AWS DynamoDB, except for this, the minimum and maximum indicate the number of reads and writes to the database. For AWS Cognito, the maximum number of users in one user pool is 20,000,000. When we get to those many users of our game, we will have to split the users up and put them into user pools related to their region. This way, all of them can continue to play the game and we can stay within the maximum user threshold. Some services may not need to have extra configuration done to them because they are already built for scaling. S3, for example, can handle 5,500 retrievals per second. Since we are using CloudFront, our users around the world will be able to access the S3 buckets from a closer location, meaning all of the retrievals won't be from just one location.