Assignment 3 - r/place

Architecture:
We use the following AWS services for our implementation of r/place:

- Amplify
- Lambda
- Elasticache
- DynamoDB
- API Gateway
- CloudFormation
- IAM

Summary:
The clients connect to a simple web application served on a serverless Amplify instance. The client will connect to our services through our API Gateway websocket which routes requests to our Lambda functions. Every operation to and from the client and between each AWS service is executed through Lambda functions. Once the client connects, the board is fetched through an APIGateway which gets the full board from a Redis cluster served using elasticache. The user's IP address is also stored inside a DynamoDB table for the purpose of preventing spam. Each placement of a pixel onto the board will place the pixel information inside of the Redis cache and then broadcast to all connected users. The pixel information is stored in Redis using a bitmap which represents 1000x1000 pixels. Every 4 bits represents 1 pixel where the value of the 4 bits correspond to up to 15 possible colours.

Amplify:
AWS amplify is an easy to use platform that allows users to host full stack applications. In our case, we are using it to host our html page. One of the benefits of amplify is that we are taking advantage of connecting it to a github repository. This makes it very convenient to push changes. AWS Amplify will scale automatically as the number of users accessing our app increases. Our html page is an 1000x1000 canvas. It has a script to fetch the board, establish a websocket, and send and receive updates through the socket.

Api Gateway:
API Gateway is a tool that allows us to create an API server on AWS. It has many features such as cost saving by only paying for the number of incoming request, easy monitoring via Cloud Watch, and various API options. We use a Rest Api and a Websocket Api in our app. We use sockets because they allow us to push updates to

our users as they come. Our websocket API has 3 main routes. Each route is connected to a lambda function which is executed when a message is received on the route. The 3 routes are: connect, disconnect, and update. We also have a Rest api with a single endpoint that is only used to get the board. It is also connected to a lambda function. The board has to be sent through a Rest Api because the maximum message size for a socket payload is only 128 KB whereas it is 6 MB for Rest apis. We can scale up to approximately 3464x3464 pixels before we reach the maximum payload size for the Rest api endpoint. AWS will automatically scale our API by hosting it on more edge locations. They also optimize latency by hosting it on edge locations nearest to the client base.

Lambda:
AWS Lambda is the foundation for serverless applications on AWS. It lets you run code without running a server. This can be beneficial in many ways, especially cost. AWS only charges for the hours that your function is running. We have 6 main lambda functions. ConnectHandler is integrated with the API Socket route $connect. DisconnectHandler is integrated with the API Socket route $disconnect. getBoard lives inside the VPC and retrieves the board bitmap from Redis. BoardHandler is integrated with the one rest api endpoint. This function calls getBoard and returns the data in the response. MessageHandler is integrated with the ‘update’ route. It passes on the change that a user made to all the clients. UpdateBoard is a function called by messageHandler; it updates the bitmap in redis. AWS will automatically scale our functions by increasing the computing power. This includes vertical scaling by running our functions on more powerful pcs or horizontal by running our functions on more pcs.

DynamoDB:
DynamoDB is a fast NoSQL Key-Value severless database. This is important for our database because we’re planning to host thousands of simultaneous users. This database partitions the data using the partition key of the table. This partition key is used to determine which node the data will be stored on. This sharding method allows for near infinite horizontal scaling. We have 2 tables set up. One stores connectionId of every host connected to our app through amplify and the other stores connectionIp. The connectionId table holds only active connections whereas the connectionIp table holds every Ip that has ever connected. This was done to prevent users from disconnecting and reconnecting to avoid the 5 minute wait time. DynamoDB can scale horizontally by adding more nodes to the cluster.

CloudFormation:
This AWS service allows you to deploy and manage all your resources on a stack. We’ve created a template file that allows us to deploy all the needed resources for our

application. This also makes it easy to make changes because we can simply update our template file and then redeploy using it.

IAM/Security:
IAM is a permission management tool. Every resource is assigned a role and each role has certain policies attached to it. These policies can be reused for multiple roles. They grant permission to perform specified actions on other resources. We have created security groups and a VPC along with setting roles and policies that allow for the AWS services to communicate with each other while blocking all unwanted outside access to our AWS services and functions. Our redis cluster is in our VPC and the only resource that has access to it is lambda. The lambda functions that interact with redis have a specific policy attached to their role which allows them to exist in the VPC. Every Lambda function has its own role which gives it access to the resources needed only by that lambda function. Our connect and disconnect handler functions have access to our dynamoDB tables to create and access entries. The update handler lambda function also has those permissions to update the users update time. This function also has access to API Gateway in order to return updates to all the other clients. We used resource based policies to allow API Gateway access to invoke our lambda functions. The only inbound rule to our VPC was through port 6379 which is necessary for redis.

ElastiCache:
ElastiCache is an in memory cache service offered by AWS. You can create redis clusters or memcache clusters. We used it to create a redis cluster with 3 nodes, 1 primary and 2 replicas. Our redis cluster is running on a subnet in 2 different availability zones. We are currently using the t3.micro size but we can upgrade that if we want to scale vertically. We can also scale horizontally by increasing the number of nodes.
We are using elasticache to hold a bitmap of our board in one row. It is currently holding 4000000 bits of data.

Broadcast Updates:
When the client connects to our websocket, the client's information will be stored in a DynamoDB table called clients. The clients table stores all of the clients currently connected to the service which will be sent updates whenever a pixel is placed. You can have multiple clients from the same IP address in this table and all will receive updates. Using the clients table, whenever a pixel is placed onto the board, a Lambda function will loop through the clients table and send the pixel information through an API Gateway to every client's waiting socket.

Spam Protection:
In order to protect against spam, each user can only place a pixel every 5 minutes. This is done by storing the IP address of each connected user in a DynamoDB table called requests. When a user requests to put a pixel, our lambda will first check whether the client's IP address made a request within the last 5 minutes. If it has been more than 5 minutes, then the request will be successful and the requests table will be updated to the current time for that particular IP address.

Scalability:
In our applications current state, it should be able to run upto 3464x3464 pixels without significant change. We are limited to this number because that is the maximum we can return in a rest api response. We may face some delay caused by the front end receiving and parsing the data. By paying more for our resources, we can get access to more memory for our db, and access to faster computers for our functions and processes. We can scale beyond 3464x3464 pixels by changing how we retrieve the board. One option is to send the board in quarters and each quarter can be up to 3464x3464 pixels.
Amplify: AWS Amplify is a serverless approach to hosting our web app which means that the service will scale dynamically for us.
Redis: The Redis cache can be easily scaled through secondaries that we have configured inside of the Redis cluster.
DynamoDB: DynamoDB has auto-scaling enabled by default, meaning that AWS will automatically scale the necessary resources needed to handle the traffic for a large number of requests and clients.
Lambda: AWS increases the computability of Lambda functions only when it is needed. This means that if more requests come into Lambda, AWS will scale the system automatically to handle them.

Availability:
Our subnet is running in two different availability zones.
The automatic scaling system that AWS provides for Amplify, DynamoDB and Lambda allows for a large amount of requests to be handled by our system.

Strengths and Weaknesses:
- UI could be improved to feature the mouse zooming into a section and allowing pixels to be highlighted when selecting them.
- We fetch the entire board in one GET request and parse the bitmap from the client's side.

Architecture Diagram: