**WhitePaper**

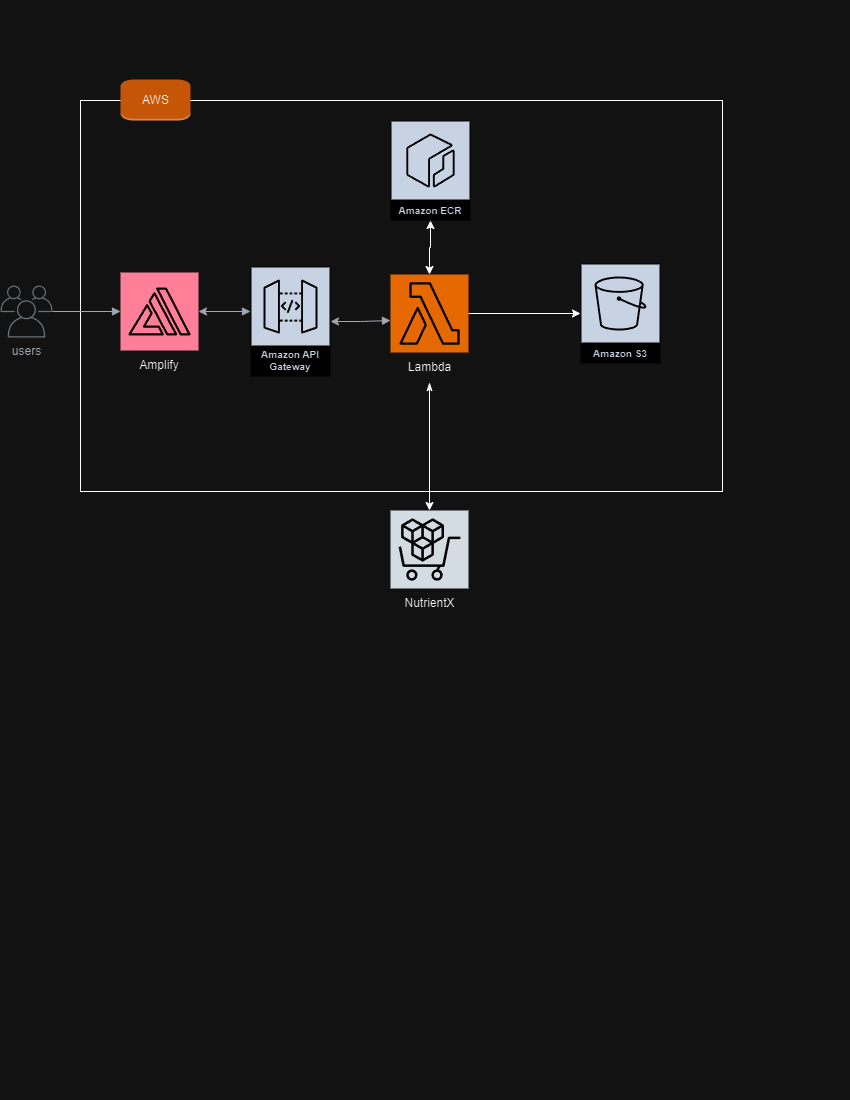
**For  
Foodlens  
Glenn Todd, Liang Chu, Erik Langdon Potts, Craig O'Connor**

**April 2024**

**Overview and Goals:**

Foodlens is a project that makes nutrition easy and fun. Our application allows our users to take a picture of their meal, or upload a picture of their meal, and see all the nutritional information of the meal. To accomplish this goal, we are using artificial intelligence to figure out what meal is being represented in the image, and then we are gathering nutritional information from NutrionX. Once we have the information, it is displayed to the user in a clear and organized manner.

We intended Foodlens to be an application that anyone could use, no matter the end user's previous exposure to technology, age, or even economic status. Our application is a completely free website that anyone can access and quickly gather the information they desire. There is no need to input personal information, or pay for access. Just access our domain, upload or take a picture of your food, and read about the nutritional information!

**Architecture Diagram:  
**

Our architecture diagram was kept very simple, as to aid in the development process. In our architecture, we are using some AWS technologies, such as – AWS Amplify, Amazon API Gateway, AWS Lambda, Amazon ECR, and Amazon S3. The way data moves in our application is very streamlined. The end user accesses AWS Amplify which hosts our front end, where they are able to upload or take an image of food. From there, the image information is passed through Amazon API Gateway into the AWS Lambda functions we created. From there, multiple operations are performed so the nutritional information populates the front end.

**AWS Technologies Used:**

The technologies used in Foodlens include the previously mentioned: AWS Amplify, Amazon API Gateway, AWS Lambda, Amazon ECR, and Amazon S3. Each of these technologies has a key role in the functionality of our product. AWS Amplify hosts our front end, which is a React web page written with JavaScript. The use of React and JavaScript allows us to have a fully responsive web page for end users to access. Amazon API Gateway is how our front end connects to our AWS Lambdas. This is done through the use of API endpoints. So when a certain function is requested by the front end through a button click, a request will be sent to the corresponding endpoint, which will trigger the AWS Lambda. Within the AWS Lambdas there is Python code that we wrote to handle each request. Alongside that we are also utilizing Amazon ECR which enables us to host our image classification Docker image for our image classification Lambda function. This image contains our trained image classification PyTorch model and our code for interacting with it. The model was trained using a 101 class dataset (https://www.kaggle.com/datasets/dansbecker/food-101). Finally we have the Amazon S3 buckets which we use to store the images that the users upload. We can use the image information to get the name of the food through our artificial intelligence model, which is then used to get the nutritional information.

There were some technologies that were originally planned to be used, but due to multiple constraints they were not included. These technologies are AWS Incognito, AWS Rekognition, AWS SageMaker, and MariaDB. There were multiple reasons these technologies could not be included. AWS Rekognition and SageMaker turned out to be incredibly costly, and it was not feasible for us to work with them given our budget. MariaDB and AWS Incognito were not utilized due to time constraints. We decided to move away from an account system that saved user information and previous food entries. Since we were not going to be adding these features, we no longer needed a user login page, which removed the need for AWS Incognito. Alongside that, since we are not storing user entries we also did not need to utilize MariaDB.

Since we moved away from AWS Rekognition and AWS SageMaker, we were able to stay under an incredibly strict budget. Using AWS Pricing Calculator, we estimate that Foodlens cost around $2 to run per month. Totaling to just over $25 dollars per year.

AWS Lambda free tier includes 1,000,000 requests per month and 400,000 GB-seconds of compute time per month. On average our image classification Lambda takes 3.128 seconds with 1024 MB of memory max. Our image upload Lambda takes 0.066 seconds with 128 MB of memory max. Lastly, our nutrition information retriever Lambda takes 0.116 seconds with 128 MB of memory max. Each full image upload, classify, and retrieve thereby takes 3.15 GB-seconds. That enables us to make approximately 127,000 full pipeline (upload, classify, and retrieve all together) requests per month for free and $0.000053 for every full request of all Lambda functions beyond that. AWS ECR private repository costs $0.10 per GB per month. Our image is approximately 6 GB making that cost be $0.60 per month. Amazon S3 free tier includes 5GB per month, 20,000 GET requests per month, and 2,000 POST requests per month. Storage beyond 5GB costs $0.023 per GB. So, if all images were held for a month, it would cost $5.842 for the month. Each 1000 POSTs beyond that would be $0.005 and each GET beyond that would be $0.0004. So, if we were to use up the entire AWS Lambda free tier with 127,000 requests per month, the GETs would cost $0.0428 and the POSTs would cost $0.625 which totals S3 costs to $6.5098. Amazon API Gateway free tier includes 1,000,000 API calls per month and $3.50 per million beyond. So, for our usage of 127,000 full calls we would remain in free territory. For the AWS Amplify free tier, we get 1,000 build minutes per month, 5 GB of data stored per month, and 500,000 requests per month. Beyond those figures, it would cost $0.01 per build minute, $0.023 per GB, and $0.30 per million requests per month. Thereby we would remain well within free tier territory. So, our application would in total cost $7.1098 per month at the scale of 127,000 full requests.

**Key Aspects:**

**Why the Cloud?** - Initially we had thought the cloud to be essential for our application in terms of having a database of nutrition information be regularly updated when new information is released and new foods added. We had thought a server would be ideal for hosting our image classification system. After we experienced the costs associated with hosting servers powerful enough for hosting our model, we had to change our plans significantly. That is when we realized the vitality of a serverless implementation for our application. Keeping servers with the ability to classify images is very expensive. Hosting a serverless AWS Lambda function is much cheaper in that it only runs when it needs to. Without the serverless capabilities of the cloud, our application would not be feasible.

**Challenges** - We initially planned to use native AWS technologies for our image classification system. We began with Amazon Rekognition as it is presented as the premier computer vision system. Not long into the development one of our team members faced a charge of over $100 for simply hosting the model for half a day. At this point we realized that for our application, Amazon Rekognition was not going to work. So, after some research we determined Amazon SageMaker to be the best alternative. It would require some extra work on our end, but could essentially perform the same function as Amazon Rekognition. Shortly after beginning work on moving to Amazon SageMaker, another team member faced a large charge of over $100. This time it was just for leaving a SageMaker domain up for a day. We had yet to even implement a model to it. This forced us to again rethink our entire plan for the image classification system. After additional research, we concluded that it would be best for our application if we did not use an AWS specific service for our image classification system and instead used a custom trained PyTorch model hosted as a serverless AWS Lambda function. This saves us a considerable amount of money.

Towards the end of our work on the project we faced an issue when connecting our Amplify hosted frontend to our backend API Gateways. Specifically, our API requests that worked when tested from Postman did not work when called from Amplify due to Cross-Origin Resource Sharing (CORS) issues. After significant efforts on the team’s part and with the help of others, we were able to resolve the CORS issue. It was a multi-pronged issue with fault lying at every level of the application. Most significantly, the API had configuration issues. The OPTIONS method request for the preflight was configured incorrectly causing requests to be rejected. Additionally, the API Gateway’s ties into our Lambda functions were not routed correctly to the right URL for our POST requests. Following this resolution, our upload image system fully worked from the frontend. The other two systems (classify and retrieve nutrition information) still threw CORS errors. We found the first issue in the classify system to be a lack of headers in the return headers of the classification lambda. After this, a CORS issue was still being thrown from a different place in the classification system. We found this issue in an extremely small section of the request being made from our frontend for both the classify and retrieve nutrition information systems. Upon resolution of these frontend issues, no more CORS issues were observed. During our research into the specific CORS error we had been receiving we found that the exact error can occur for any number of reasons. It was apparently very generic (as we saw with every level of our system causing the issue independently) and thereby difficult to find the solution. But, by working together as a team and collaborating with others we were able to find resolutions.

**Stretch Features** - We would have liked to implement user accounts and the ability of users to keep track of their nutritional data in a persistable format. Users would be able to log into their accounts, set nutrition goals, scan their food items, view their progress towards their goals, search for foods, add foods to their favorites, and receive notifications weekly on their progress. For this, MariaDB would be used for persistability of data, Amazon SQS would be used to schedule notifications, and Amazon SNS would be used to deliver those notifications. Despite not having enough time to complete these aspects, our application in its current state still delivers significant value to the user by fulfilling the core functionalities needed to get nutritional information from a photo of a food.

With additional time, we would have also liked to have broadened the scope of our trained model for image classification as well as increased the accuracy. The model is currently trained on 101 different food types with an accuracy of just over 96%. The dataset used to train the model contained 1000 images of each of the 101 different food types and the final iteration of the model took nearly a full day to train. Given additional time we would have been able to increase these fields, but the cost limitations of our scenario limits the degree to which we can take the model. A list of all the available foods can be seen below.

