Project Documentation: Onlyfish

Team members: Patrick Singer, Marilena Brink, Lea Pinnow, Teresa Kutzner

1. Project Overview

In this project, we created a web app where visitors can view a live stream of an aquarium called “OnlyFish”. The web app also features the ability for the viewer to call an object recognition model to detect fish in the current frame, as well as the ability to toggle the aquarium lights with the push of a button. This feature is aimed at the owner of the aquarium and can only be used if the correct passcode is given.

The goal of this project was to gain knowledge in the area of cloud computing and setting up a cloud infrastructure for live video streaming with an end-to-end solution.

The final architecture of the project can be seen in the image below.

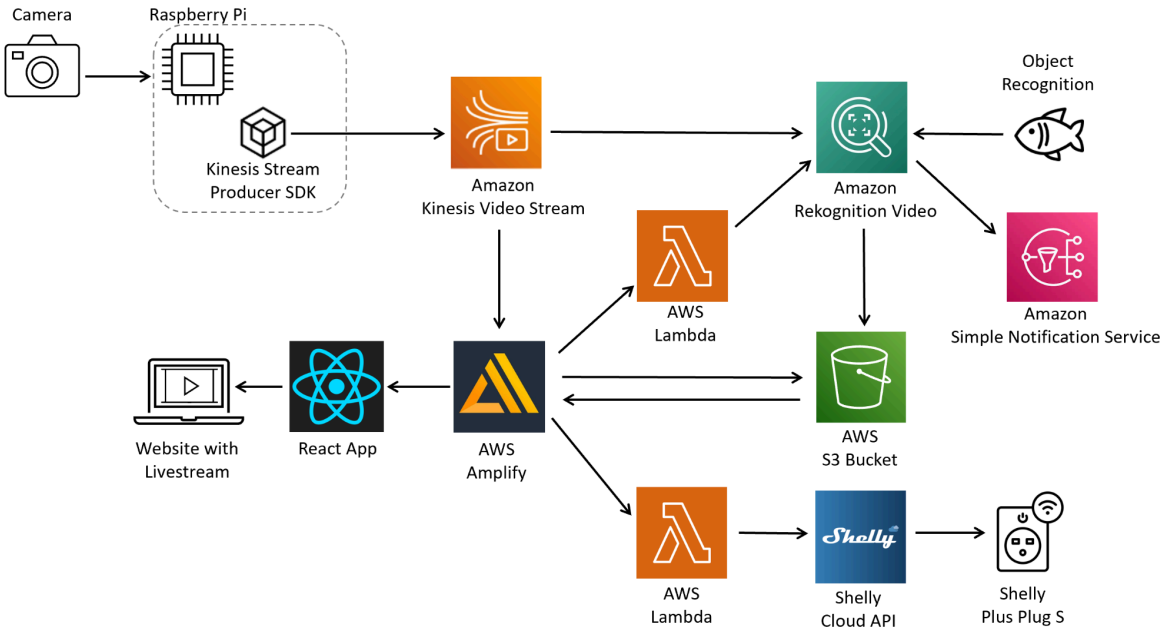


Fig. 1*: Final cloud architecture for the project OnlyFish using AWS Kinesis, Amazon Rekognition, AWS Amplify, AWS Lambda and AWS S3.*

In the following sections, we describe the overall setup and the problems we faced during the implementation for the main components of our project.

2. Hardware & Live Streaming

Setup

In our project we decided to use a Raspberry Pi (Model 3B) with a connected camera as the origin (producer) for our live stream. As a prerequisite, we used the Raspberry Pi Imager[1] to flash the Raspberry Pi with the latest version of the Raspberry Pi OS[2].

First, we created a new Amazon Kinesis Video Streams Instance by running the following command in the AWS CLI.

```
aws kinesisvideo create-stream --stream-name "OnlyFish"
--data-retention-in-hours "1"
```

The data retention was set to one hour, which is the smallest possible setting. This was done to save costs as we don't intend for the user to watch previous streams.

For Raspberry Pi to be able to stream to a Kinesis Video Instance, a new IAM user was created that has write access to kinesis video streams. For this, a custom policy was created. In order to connect the Raspberry Pi to the internet, a *wpa_supplicant.conf* file with the WIFI credentials was created and moved to the boot folder of the Pi.

In the next step, we cloned the *Amazon Video Streams Producer SDK* repository[3] and used `cmake` to build the application on the Raspberry Pi. Before building the SDK, several libraries needed to be installed on the Pi. A list can be found in the Kinesis Video Streams documentation by AWS[4].

As a final step, we needed to configure the SDK to stream to the previously set up Kinesis Video Streams instance as well as provide the IAM user credentials to authorise the Pi to write to the stream. This was done by setting the following parameters before starting the stream:

```
export
GST_PLUGIN_PATH=PATH/amazon-kinesis-video-streams-producer-sdk-cpp/build
export AWS_DEFAULT_REGION=AWS Region i.e. eu-west-1
export AWS_ACCESS_KEY_ID=Access Key ID
export AWS_SECRET_ACCESS_KEY=Secret Access Key
```

The stream can then be started by calling the gstreamer with the Kinesis Stream name.

```
./kvs_gstreamer_sample OnlyFish
```

[1] https://github.com/raspberrypi/rpi-imager
[2] https://www.raspberrypi.com/software/operating-systems/
[3] https://github.com/awslabs/amazon-kinesis-video-streams-producer-sdk-cpp.git
[4] https://docs.aws.amazon.com/kinesisvideostreams/latest/dg/producersdk-cpp-rpi-software.html

Problems & Learnings

The AWS Video Streams SDK was written to use the proprietary Raspberry Pi camera module, which has a special connection on the Pi board. Because we were using an external camera connected to the Pi via USB, the SDK was not able to recognise the camera. We found a workaround by installing the *fswebcam* package. With this library, the USB camera was treated as a standard camera module and the SDK was able to recognise it.

```
sudo apt-get install fswebcam
```

Another problem we encountered when starting the stream from the Raspberry Pi was that in order to start the stream, 5 commands needed to be executed each time. We decided to automate this step in a script. As we did not have prior experience in writing linux command scripts, this was a new experience for us as we learned how to write bash scripts and make them executable from outside the command line. By calling this script in the etc/rc.local file, we were also able to start the stream remotely by giving power to the Pi via a smart plug.

```
PII#!/bin/bash
cd /home/admin/amazon-kinesis-video-streams-producer-sdk-cpp/build
export
GST_PLUGIN_PATH=/home/admin/amazon-kinesis-video-streams-producer-sdk-cpp/build
export AWS_DEFAULT_REGION=eu-west-1
export AWS_ACCESS_KEY_ID=AKIAYEVGMNIEHFUXXRGG
export AWS_SECRET_ACCESS_KEY= ***
./kvs_gstreamer_sample OnlyFish
```

3. Object Detection

3.1 Setup

For the object detection setup with Amazon Rekognition a tutorial from the AWS documentation was taken as a guideline.[5]

3.1.1 Resources and development environment

For the development of label detection on the previously defined Kinesis Livestream, the AWS Toolkit for Visual Studio Code was set up in conjunction with boto3, the AWS SDK for Python.

The following resources were required to implement label detection with AWS Rekognition:

Input:

A **Kinesis Livestream** has been set up as per the previous instructions.

Notifications:

A **SNS Notification Topic** was created to receive notifications when an object of interest is first detected in the video stream. The SNS topic can then be subscribed to an endpoint. We

[5] https://docs.aws.amazon.com/rekognition/latest/dg/streaming-video-detect-labels.html

have defined a mail address as the endpoint for this purpose. When creating the SNS topic, we used the default access policy from AWS, as this was sufficient for our application.

Output:

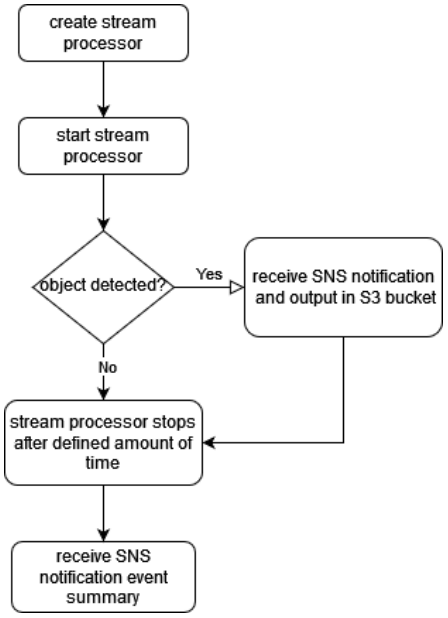
The original plan was to realise the output via a Kinesis Datastream. However, this plan was scrapped due to cost considerations and administrative challenges. Instead, it was decided to use an **S3 bucket** as output. This decision was made as the S3 bucket provides a cost effective and easy to manage solution to store the recognition results.

Giving access to the resources

Since the Amazon Recognition Service requires access to the resources in order to define the stream processor that executes the label detection, a new **IAM policy** must first be created. This defines access to the created Kinesis video stream, the SNS topic and the S3 bucket. An **IAM service** role for Recognition is then created, to which the previously created IAM policy is attached.

3.1.2 Implement the Amazon Rekognition Video label detection stream processor

The Amazon Rekognition stream processor can be used to detect labels in streaming videos. Our label detection process with Amazon Rekognition stream processors is summarised in the adjacent diagram and then described in detail.



Create Stream Processor

To create a new stream processor instance, `CreateStreamProcessor` must first be defined and called. In addition to the definition of the processor name, various settings can be set here.
First, an input must be defined for the stream processor, which, as already described above, is the Amazon Kinesis video stream created. The ARN of the video stream is transferred to the processor. An output for the recognition results must also be provided in the form of our S3 bucket prefix. In addition, a NotificationChannel must be defined, which is defined by passing our SNS topic ARN. In the ConnectedHome settings, you can specify what you want to recognize through recognition in the livestream. You can choose between PERSON, PET, PACKAGE, ALL. As we want to recognize fish, we have opted for the PET package. We also set a MinConfidence, which is the minimum confidence (from 0 to 100) the ML model has to have in order to give feedback that it has recognized something.

Start Stream Processor

To perform a label detection in the live stream, the stream processor must be started with the "StartStreamProcessor" command. After starting the processor, it is in the "STARTING" state. As soon as the analysis is complete or the processor is stopped manually, it switches

to the "STOPPED" state. In the event of errors during processing, the stream processor assumes the "FAILED" state.
To start the stream processor, the name of the previously created stream processor as well as a StartSelector and a StopSelector must be transferred. The StartSelector defines the starting point for the analysis in the stream and can accept either a Unix timestamp or a fragment number as input. In our case, we use the number of the current stream fragment as the starting point. After successful initiation, the state of the processor should be "STARTING".
The StopSelector defines when the Stream Processor should stop analysing the stream. It is specified as a fixed duration in seconds. If there were no errors and the defined duration has elapsed, the stream processor switches to the "STOPPED" state.

Object detection feedback

Event summary Feedback by SNS Notifications

After the streaming processor has stopped, regardless of whether an object has been recognized or not, feedback regarding the stream processor is sent to the user via SNS notification. This contains information such as the ARN of the analysed stream, the duration of the processing and information on the stream processor settings, as seen below.


```
{"inputInformation":{"kinesisVideo":{"streamArn":"arn:aws:kinesisvideo:eu-west-1:559768431112:
stream/OnlyFish/1705501167632","processedVideoDurationMillis":30000.0}},"eventNamespace":{"typ
e":"STREAM_PROCESSING_COMPLETE"},"streamProcessingResults":{"message":"Stream Processing
Success."},"eventId":"4a2fa46c-1631-4346-b0f2-b64dd2ce712b","tags":{},"sessionId":"4a2fa46c-16
31-4346-b0f2-b64dd2ce712b","startStreamProcessorRequest":{"name":"RekognitionStreamProcessor",
"startSelector":{"kvsFragmentNumber":"91343852333181861913309853272845057629589566222"},"stopS
elector":{"maxDurationInSeconds":30}}}
```

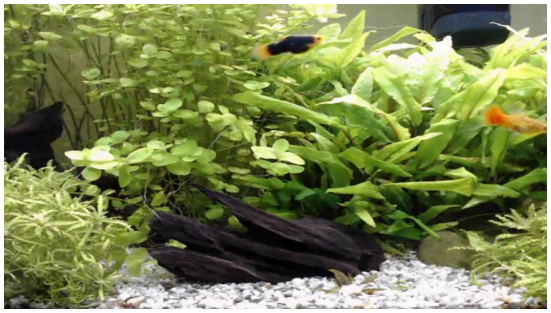

Feedback when object is detected

When an object is recognized, an SNS notification is sent with important details such as the recognized label, the confidence with which it was recognized and the bounding box, i.e. in which part of the image the object was recognized. In addition, the frame of the stream in which an object was recognized and a section of this frame with the recognized object are saved in the output S3 bucket.

SNS Notification:


```
{"inputInformation":{"kinesisVideo":{"streamArn":"arn:aws:kinesisvideo:eu-west-1:559768431112:
stream/OnlyFish/1705501167632"}},"eventNamespace":{"type":"LABEL_DETECTED"},"labels":[{"id":"9
69f2047-b2f6-4b18-8bee-e74088d69ad2","confidence":23.38142,"name":"PET","frameImageUri":"s3://
rekognitionoutputbucket2/data/RekognitionStreamProcessor/0d1604d5-99b1-41e4-8cba-7bf299ee12a3/
notifications/27_5.400000095367432.jpg","croppedImageUri":"s3://rekognitionoutputbucket2/data/
RekognitionStreamProcessor/0d1604d5-99b1-41e4-8cba-7bf299ee12a3/notifications/27_5.40000009536
7432_heroimage.jpg","videoMapping":{"kinesisVideoMapping":{"streamArn":"0d1604d5-99b1-41e4-8cb
a-7bf299ee12a3","fragmentNumber":"91343852333181860635755732730332509894523851797","serverTime
stamp":1707129343087,"producerTimestamp":1707129260439,"frameOffsetMillis":5400}},"boundingBox
":{"left":0.47311956,"top":0.097405836,"height":0.07886997,"width":0.11430776}}],"eventId":"36
15f4ce-c374-32e2-8612-bd3242cc4b08","tags":{},"sessionId":"0d1604d5-99b1-41e4-8cba-7bf299ee12a
3","startStreamProcessorRequest":{"name":"RekognitionStreamProcessor","startSelector":{"kvsFra
gmentNumber":"91343852333181860615948692101766423878649111890"},"stopSelector":{"maxDurationIn
Seconds":30}}}
```

Current video stream frame:



Output from Rekognition for a detected pet:

3.2 Problems & Learnings

1. Amazon Rekognition Label Detection not usable in Frankfurt Region

For Amazon Rekognition, the project faced the challenge that the Rekognition Label Detection function was not available in the Frankfurt region (eu-central-1). To solve this problem, the project was moved to the Ireland region (eu-west-1), where Rekognition is fully supported.
This experience has taught us important lessons. Firstly, it became clear that not all features of an AWS service are available in all AWS regions. In this particular case, the label detection feature of Amazon Rekognition was not available in the Frankfurt region, although other features of the service could be used. This finding emphasises the importance of thoroughly researching and checking the availability of services and features in the respective AWS regions before deciding on a specific region.

2. Using Kinesis Datastreams quickly becomes expensive

Another problem the project faced was the realisation that the definition of Kinesis DataStreams was too expensive. The assumption was that as long as no output was passed from Rekognition (i.e. no data was transferred), no costs would be incurred. The actual situation, however, was that a Kinesis DataStream incurs costs as long as it is active and cannot be stopped. Instead, it must be deleted and restarted each time, otherwise costs are incurred permanently.
To solve this problem, it was decided to define an S3 bucket as a recognition output.
This experience also taught us important lessons. Firstly, it became clear that when using AWS services, it is crucial to take a closer look at the cost models of the services used. Although the assumption was made that no costs would be incurred as long as no output was transferred, it turned out that this was not the case for Kinesis DataStreams. A better understanding of the cost structure and models can help to avoid unexpected costs.
However, it is important to note that the description of costs is often not very clear and concise, which can make it difficult to predict and plan costs.

3. Analysing the most recent frame of the livestream must be set specifically

There was also the problem that despite the transfer of the current Unix time in the StartSelector of the stream processor, the first fragment of the Kinesis livestream was always analysed, which meant that the same fragment was always processed.
The solution to this problem was to extract and pass the fragment number of the current fragment of the Kinesis DataStream. This adjustment ensured that the current fragment was analysed each time and not always the same one.
It became clear that the settings and methods defined in the tutorial are not always suitable for every project and often need to be adapted to the specific requirements. In the specific case, it turned out that the transfer of the current Unix time was not sufficient to achieve the desired behaviour and an adaptation of the logic was necessary.
Another important finding was that there are few sources for label detection with Rekognition on a Kinesis livestream besides the official Amazon Rekognition tutorials. This often makes it a challenge to find and customise the appropriate settings and methods. It is therefore important to be flexible and to develop and adapt your own solutions if necessary.

4. Extract FragmentNumber from StreamingBody

One problem that arose from solving the previous problem was extracting the fragment number of the current frame from the Kinesis livestream. The information about the current frame was passed in a `botocore.response.StreamingBody`. However, we could not decode this directly to extract all the information in a meaningful way, which led to difficulties.
The solution to this problem was a workaround: First, the StreamingBody was decoded with the character set `iso-8859-1 (Latin-1)` to make it readable. The fragment number was then extracted from the returned string using regular expressions (regex). This workaround allowed the fragment number to be successfully isolated and used.
A lesson learned from this experience is that sometimes it is necessary to find creative solutions when the standard methods do not work. Although we originally attempted to decode the StreamingBody directly, this proved to be unfeasible. However, by applying a workaround, we were able to achieve the desired result.

5. Low confidence for detected labels

Another problem that arose during the project was the low confidence in label detection with Amazon Rekognition. In most cases, the confidence was between 10 and 20 percent. This could be due to a number of reasons, including poor image quality of the video stream, the incorrect position of the camera, insufficient lighting and the fact that the object detection model used by Rekognition was not specifically trained on fish, but on pets. In particular, bright fish are difficult to recognize on the bright background with plants, which was also reflected in Rekognition's output where mostly dark fish were recognized.
Various approaches were considered to tackle this problem. Initially, different camera positions were tried, but none of these approaches led to a significant improvement in the results.
Another solution was to use Amazon Rekognition Custom Labels. For this purpose, a custom model was trained with fish images. However, it turned out that Rekognition Custom Labels does not support the application of the models to Kinesis livestreams, but only to

individual frames. This meant that individual fragments of the video stream would have to be extracted and iterated over, potentially causing performance issues and not fitting into the project schedule.
We learned from this part of the project that sometimes it is necessary to compromise and find alternative solutions when the standard services cannot deliver the desired results. It is important to understand the limitations of the available tools.

6. Limited label detection analysis duration

A problem that arose from the use of Rekognition label detection was that Amazon Rekognition only analysed the livestream for a duration defined by the StopSelector. This meant that the stream processor had to be restarted frequently in order to perform object detection.
The proposed solution was to integrate a button for manually starting object detection in the web application. By pressing this button, the user could start the analysis of the live stream at any time, which enabled more flexible control of object detection.
An important lesson from this part of the project was that integrating user interactions into the application can often be an effective way to counteract limitations of the services used.

4. IOT Device Control

Setup

In order to be able to control the lights of the aquarium, we used a smart plug from the company Shelly. We set up an access point for the plug to be controlled via HTTP requests by creating an authorisation key for the Shelly cloud API and a device ID for the smart plug.

In the next step, we wrote a python function that first uses a GET request to the plug to get the current state (on or off) and then, based on this information, sends a POST request to toggle the smart plug to the opposite state.

This python script was then integrated into a lambda function. Because the lambda function code will be accessible once called, we created environment variables for the authorization key and endpoint server by using the following CLI command:

```
aws lambda update-function-configuration --function-name toggle_lights
--environment "Variables={auth_key=***,
uri=https://shelly-94-eu.shelly.cloud}"
```

Additionally we needed to set up CORS for the lambda function so we can call it from the web app. This was done in the AWS Console under the “Function URL” setting by defining our web app URL as a valid origin.

Finally, we added a button and entry widget to the frontend of our web app and wrote a function that creates a POST request to the lambda function with the current text of the entry widget as the passcode argument. This way, the user first needs to input the correct

passcode and then press the button to toggle the lights. The lambda function validates the passcode and executes the HTTP requests to the smart plug, if it is correct.

Problems & Learnings

At the beginning, we struggled to find a way to control a smart plug via an API. Most companies that offer smart plugs don't have an open API but must be controlled through their service. We soon learned that we need a smart plug that is accessible through a standard http request. After some searching we found that the company Shelly provided the option to create an http endpoint for its devices.

A task that was much more difficult than expected was calling an AWS Lambda function from our amplify hosted web app. We were surprised to find that there was no straightforward way by AWS to connect both services. After some research we decided to use a standard HTTP request to call the lambda function. Here we had difficulties setting up CORS the right way. We learned that some headers were blocked by lambda and needed to be removed from the HTTP request made from the web app. In the end we only included the “Accept” header type as we found other headers were not necessary and could potentially lead to problems with lambda.
For a long time, we only managed to get GET requests working. Wenn calling POST requests for the lambda function, we would always get a 502 error without further information. In our case we needed to be able to send POST requests because we wanted to authenticate the use of the light toggle function with a password.
After debugging the code step by step, we found that lambda converted the input body from a json to a type of string. However even with python's `json.loads()` function, we were not able to convert the body back to a json structure. We finally found a workaround by using the `eval()` function in python, which was able to convert the body back to a readable format.

In accomplishing this task we learned a lot about how to correctly set up a lambda function and how to structure an HTTP request that executes a lambda function while also including arguments, like the passcode value in our case.

5. Web Application

Setup

First we created a new React application using the create-react-app command. This created a new React application with all the necessary files and dependencies.
Next we initialised a new Git repository[6].
This allowed us to track changes to our code and deploy our application to AWS Amplify.
After initialising our Git repository we connected it to AWS Amplify.
Once we had connected our Git repository to AWS Amplify by giving it the link to the repository, we configured the build settings for our React application. This allowed us to specify build settings such as the environment variables and the build commands.

[6] https://github.com/marilena-brink/amplify-react-app.git

Finally we deployed our React application to the cloud using AWS Amplify Hosting. This allowed us to host our application on a global content delivery network (CDN) and make it available to users around the world.

Problems & Learnings

The initial setup of the framework for a React app hosted via Amplify was well-documented by AWS and easily achievable in a few steps. The connection to the GitHub repository works well and Pushing to the GitHub repository triggers the pipeline in Amplify. This automatically generates a new build and deploys it. Changes are available after a short waiting period.

5.1 Connection of Webapp to Kinesis

To display the Kinesis VideoStream on the website, we had to create a connection from the web app to KinesisVideo. To do this, we used the AWS SDK for React Native in Javascript.

First, we created an instance of KinesisVideo to display our stream. To get the link of the stream, we were able to use a function called `getDataEndpoint()` of the AWS SDK, which returned the necessary endpoint using the stream name.
We encountered authentication problems, which meant that we had to pass an *AccessKeyID* and the *SecretAccessKeyID* of Amplify's IAM role to the requests in order to authenticate the request.

This caused problems because we had initially hardcoded the credentials for the AccessKey and the SecretAccessKey in the code. This led to AWS disabling our AccessKey, as the secret key was publicly available in our Github repository. Due to this measure, we had to store the credentials in the .env file of our project and then import them into our code. This was possible with Amplify and the environment variables, as we automatically read in the keys from an IAM user in the build script. As a result, the credentials were no longer hard-coded and we were able to authenticate our requests without any problems.
Furthermore we had to create IAM policies to allow Amplify access to Kinesis and KinesisVideoStreams.

By accessing the endpoint, we were able to retrieve the stream as live video using the Dash.js library. It was important to let the individual lines of code wait for each other, as otherwise the authentication was not completed, but the code was still executed and errors occurred. The stream itself should therefore only be loaded after the stream URL has been successfully loaded.
With the help of Dash.js, we implement a media player in the code, which then displays the stream data.

Dash's `getMP4MediaFragment()` method shows video fragments on the website one after the other, which form the live stream. As the data requested is very large, we have generally scheduled the authentication for the stream via KinesisVideoStream to 5 minutes, as we

wanted to save resources and therefore costs. The user must therefore manually reload the stream if it expires due to authentication.

We have implemented a reload button for this and inform the user that they must reload the stream in order to see the current transmission.
If no stream is running, a 403 error occurs due to the code that calls the stream via URL. We intercept this and show the user information that the stream is currently offline.

Learnings

Setting up the stream taught us a few things, such as how a stream works in general. At first, we didn't realise that a stream only consists of individual fragments that are played one after the other. We also realised how much data is sent and how quickly such a stream can become very costly. We also realised how important it is for AWS to always authenticate all requests.

5.2 Connection of Webapp to Rekognition

After setting up our React web app with Amplify, the Rekognition service, the S3 bucket and the SNS service, we looked at how to connect the web app to Rekognition. When Rekognition detects a fish, it should send a notification to the website to indicate to the user that a fish has been detected in the stream. In addition, the image with the recognized fish should be displayed to the user. We tried different approaches to achieve this.

First, we used the existing SNS service, which sends an email to our mail onlyfish@web.de when a fish is detected. This email contained JSON data and the images that Rekognition labelled as "pets". With SNS, we wanted to set our website as an additional endpoint so that this data could be sent directly to our website and be accessed.
For this purpose, an SNS subscription was created via an https protocol, which gave us a link that we could address with the website. However, when creating this subscription, SNS wants the subscription to be confirmed, for which a SubscriptionConfirmation should be received.
Unfortunately, it was unclear to us how the POST/GET requests have to be structured in order to receive the SubscriptionConfirmation, and there were only a few examples of this online. In addition, it was claimed in a forum that it could be a problem that the SNS service is located at AWS Ireland and our Amplify web app at AWS Frankfurt and therefore they cannot communicate properly. To save time, we then decided on a different solution after some trial and error.

As a new approach, we then decided to access the S3 bucket directly, in which the images of the detected fish are stored. To do this, we declared the S3 bucket as the backend of our web app by setting up a backend for the web app with the help of Amplify Studio.
The S3 bucket was then specified in the backend as the main storage of our website, which allowed us to address it on our website using its web address "https://rekognitionoutputbucket2.s3.amazonaws.com/" via the `AWS.S3()` instance.

After this was set up, there were some problems, firstly because no settings were made for CORS headers and secondly because policies were missing so that the web app could read the contents of S3 and the bucket could make its contents public.
To solve these problems, we had to specify AllAccess for S3 buckets in the IAM role of the Amplify web app. We also set up a bucket policy for the S3 bucket that makes the content of the bucket public and added CORS rules that regulate the permitted headers and the permitted requests from external websites.

After successfully connecting the Amplify web app to Rekognition, the Rekognition service could be triggered by pressing a button and the bucket could be checked for new documents. If new documents are recognized in the bucket, this means that fish have been recognized and their images have been stored in the bucket. These images are then sent to the website and displayed to the user as recognized fish. Since we only get the coordinates of the recognized bounding boxes through the SNS notification and could not transfer the SNS notification to the web application, it was therefore not possible to display them in the livestream.

Implement Button to start object detection from the web application

As already explained in point 6 in section 3.2, we have decided to make object detection accessible to the user via a button in the web application due to the limited processing time of recognition object detection.
A problem with the button logic implementation was the discrepancy between the programming language of the web application, which was written in JavaScript, and the existing recognition code, which was in Python.
To solve this problem and make the recognition code executable within the web application, we decided to implement a lambda function. This way, the web application could call the lambda function to perform the desired analysis.
This experience illustrates the importance of carefully planning and coordinating the implementation of each component during the development process to avoid compatibility issues and ensure a smooth process.
In order to be able to use Rekognition from the Lambda function, a new IAM user had to be created for the Lambda function, which has an AmazonRekognitionServiceRole, access to the Kinesis video stream, the S3 output bucket and the SNS service, as well as a PassRolePolicy.
A function URL was also created to call the Lambda function from the Amplify application. In addition, the timeout time, which defines how long the Lambda function runs, had to be increased, as the processing was aborted before the end of the label detection.
In addition, cross-origin resource sharing (CORS), which gives another resource access to the created function URL, had to be activated for the function and our amplify application had to be defined as a permitted origin.

6. Conclusion

Working on this project was a challenging but very rewarding experience for us. We were able to learn a lot about how a cloud infrastructure can be set up and gained experience in working with many different services from AWS. In the end, we were able to develop all features that we defined in our research phase. We even managed to include the optional feature to toggle the aquarium lights with the press of a button. However, we had to change some aspects during development as we learned more about the limits of the services we used. For example, because of monetary reasons, we had to change the planned continuous, real time object recognition to an action that can be triggered by the user to detect a fish.

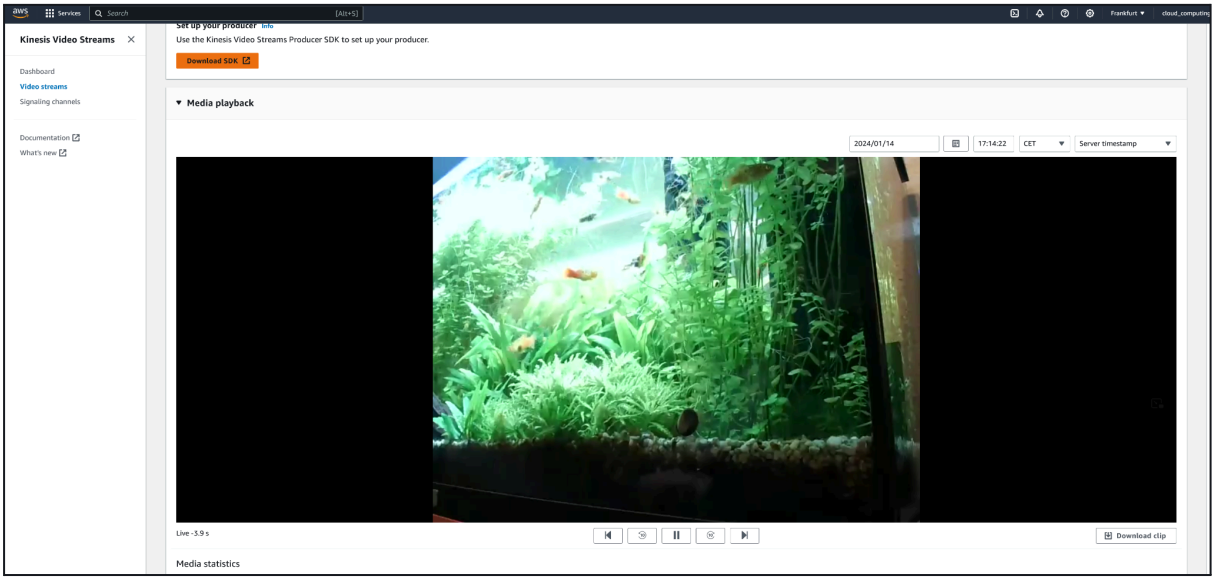
Besides gaining insight into the development of a cloud architecture, our main takeaways from working with AWS are the following:

- AWS has a very **steep learning curve**. The huge amount of offered tools and different services made it difficult for us to get a foothold in this environment and required us to do a considerable amount of research and trial and error.
- The **cost** of services was **difficult to predict** as they only become apparent once the service is running. In our case, we were surprised by the large costs from Kinesis Data Streams and in turn needed to adjust our architecture to use a S3 Bucket for data transmission between Rekognition and Amplify.
- Every part of the pipeline requires **authentication**. This becomes difficult at times because authentication credentials can't be openly accessible. We learned that working with environment variables in the AWS services enabled us to keep IAM user credentials secret.
- **Connecting different AWS services** is not as straightforward as we thought. We learned that there is often no easy prebuild connection option by AWS and we instead need to employ basic HTTP requests for many connections. For example the connection between Amplify, Lambda functions, and the S3 bucket.
- After implementing part of our project in one region, we realised that **certain features** such as label detection by Rekognition are **not available in** our **region** (eu-central-1). Before deploying a project, it should therefore be checked, if all service features are available in the desired region to avoid having to relocate the project to another server during development.
- AWS Services are generally well documented but for most AWS applications the **documentation is often insufficient** or outdated. We learned that a significant amount of time needs to be spent on research and trial and error style testing.

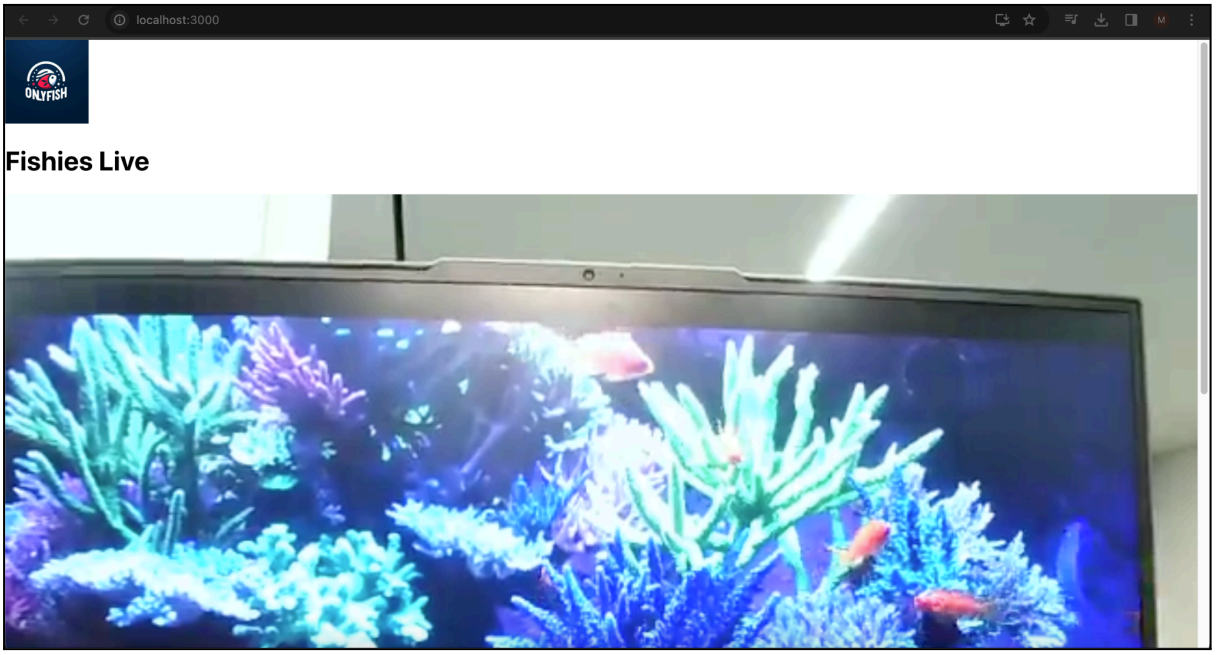
7. Appendix

Project Progress Milestones

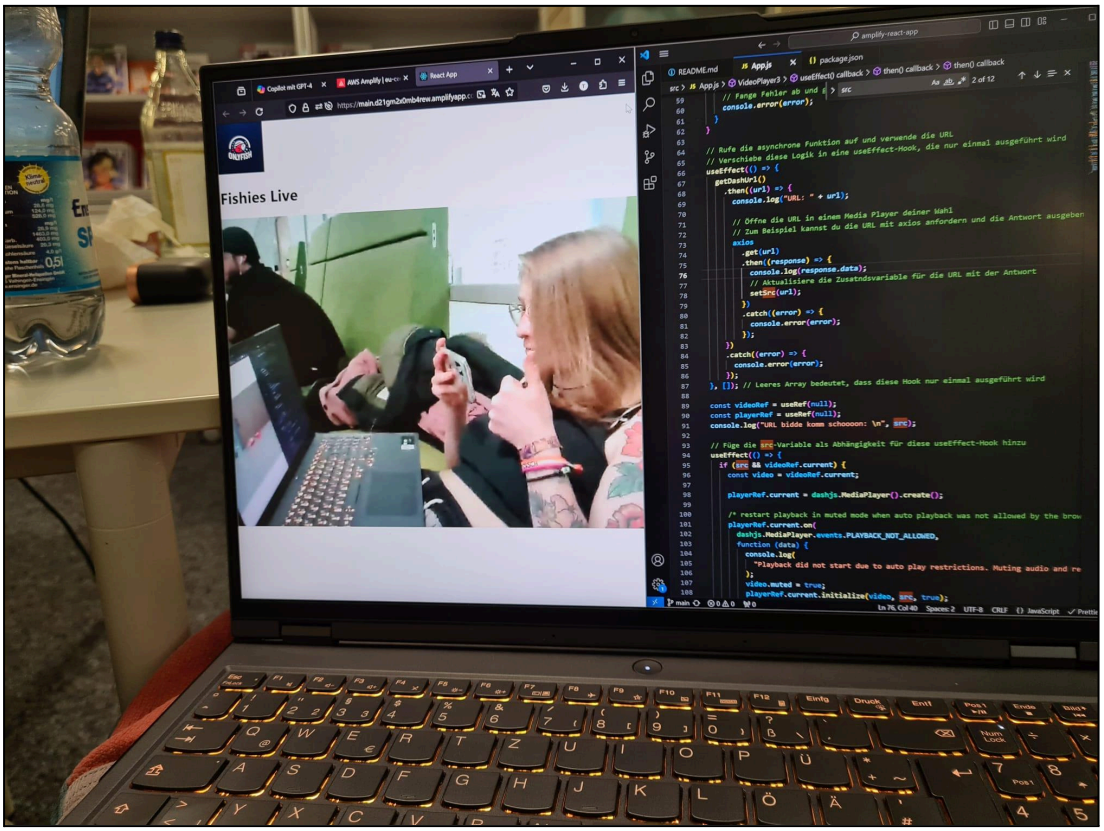
1. First successful stream from the Raspberry Pi to AWS Kinesis.



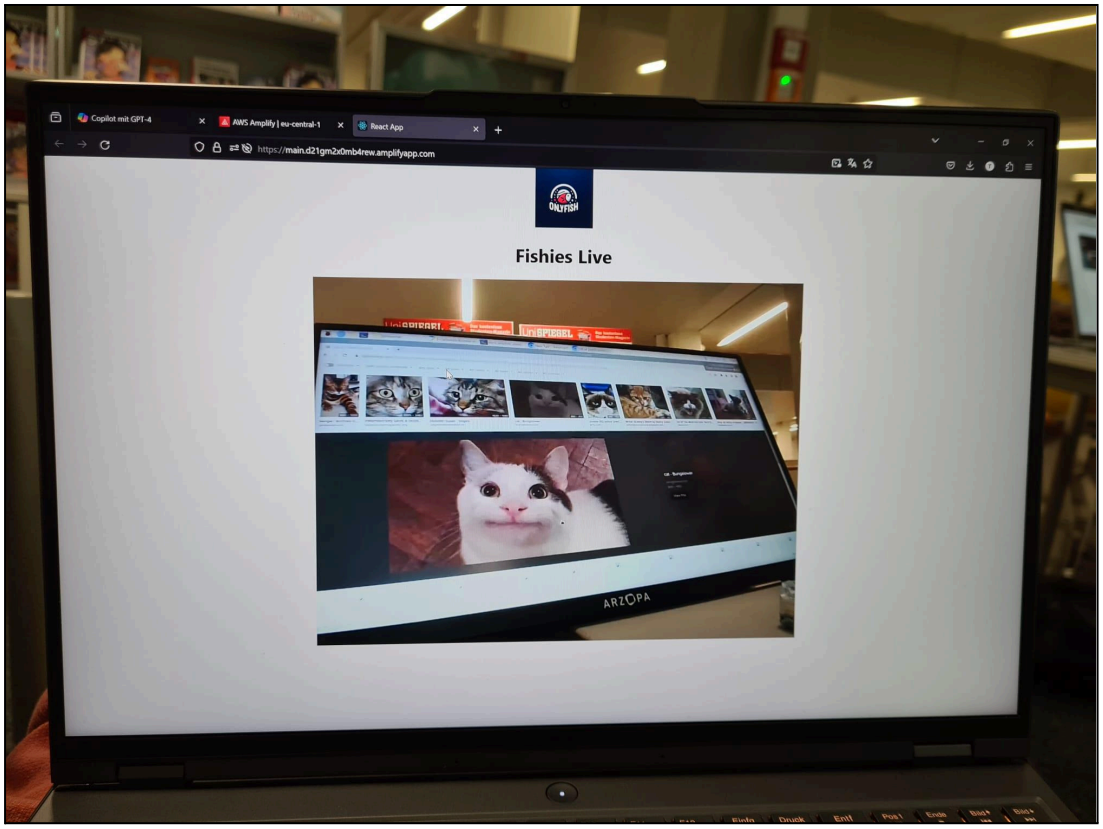
2. First working stream on a locally hosted website.



3. First working stream deployed on an Amplify hosted website.

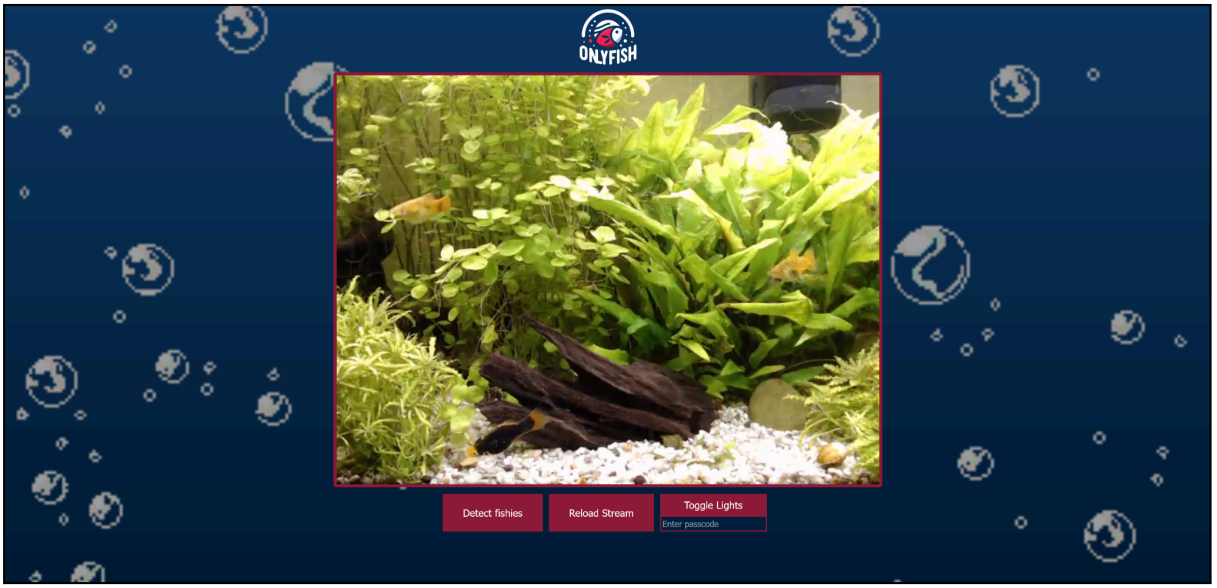


4. Trying to detect pets in the stream with AWS Rekognition.



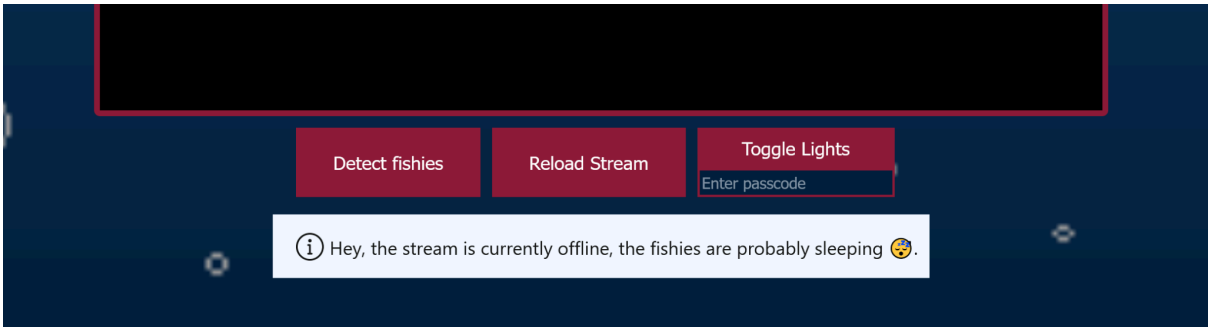
5. Final camera and Raspberry Pi setup in front of the aquarium.

6. Host of Aquarium stream on deployed website.

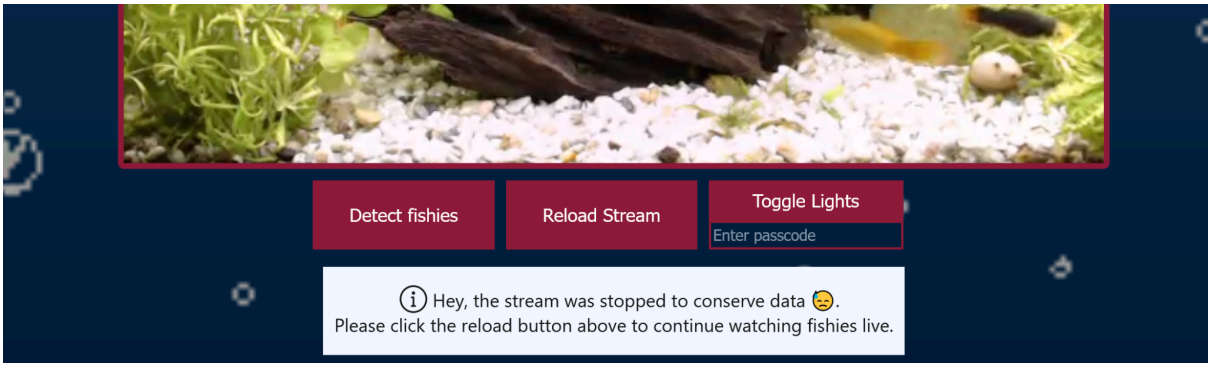


Notification Types

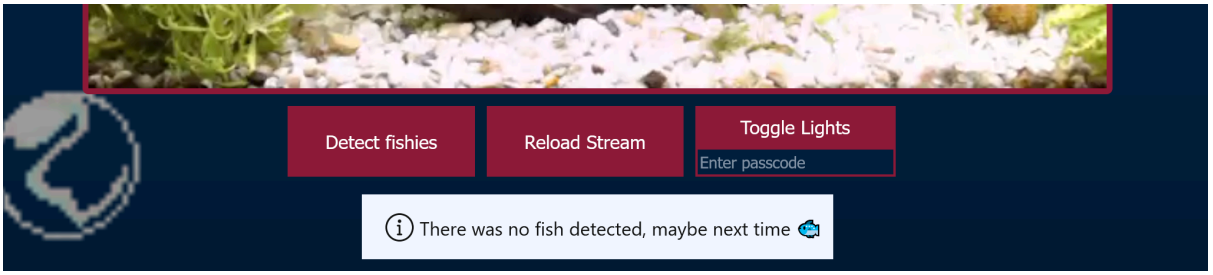
1. Stream is offline.



2. Stream stopped after viewing it for more than 5 minutes.



3. No fish was detected during object recognition.



4. A fish was detected during object recognition.