

The Magic Wand Super Project

ECEN 5783: Embedded Interface Design

Brian Ibeling & Connor Shapiro

12/11/2019

Final System Architecture Diagram

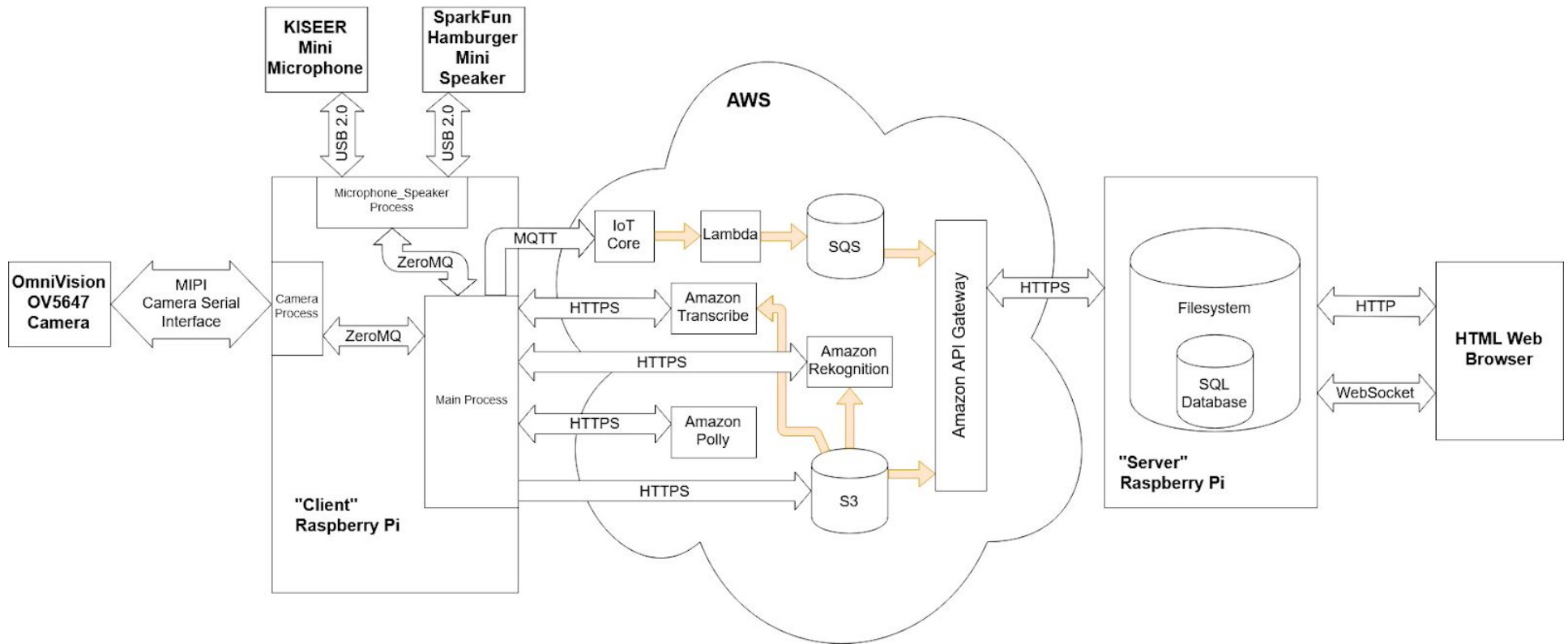


Figure 1: Magic Wand System Architecture Diagram

Note: Yellow arrows indicate AWS inter-service transport.

The embedded application side of the Magic Wand is handled by the Client Raspberry Pi (Client Pi). This runs three separate python3 processes to allow system processing to completed in parallel, which is particularly valuable as audio is being constantly recorded to detect user provided voice commands. The main process block shown on the left side of the Figure 1 acts as the primary processing loop for the Client Pi application. This

would receive audio filenames from the microphone process at a continuous rate of once every 10 seconds. Each audio filename is then maintained in a processing queue within the main process, and sent to AWS Transcribe to be converted to text and return to the Client Pi. This allow parallel processing of these audio files and improve system responsiveness. Based on the audio returned, voice commands are triggered by the “Identifio”, “correcto”, and “wringo” keywords to capture a picture or provide an “image tag” for AWS rekognition processed images. Images are then sent and processed by AWS Rekognition to identify what the image was, and the highest confidence label is then output from the Client Pi speaker via an audio file from AWS Polly. Inter-Process Communication between the Main Process and the other peripheral hardware processes (camera and microphone_speaker) was handled via Zero Message Queue in a Pub-Sub configuration. This allowed data to be passed without needing the receiving process to provide a response for each message and allow each process to run more autonomously. Additionally, the use of S3 buckets was needed to allow several AWS components to access needed data files, and to allow the Server Pi application to access image files for the user GUI.

The API Gateway provides an interface through which the Server Pi can pull images (from S3) and image metadata (from SQS). The Server Pi - running a Node.js application which runs at bootup - polls the gateway periodically and anytime there are new images & metadata they are downloaded and stored on the Server Pi filesystem. Images are stored as normal files in the filesystem while all metadata is held in a local SQL database. Data on the Server Pi is exposed to a Server user via an HTML web browser (accessible anywhere on the same LAN as the Server PI), which uses a WebSocket to send commands and receive metadata from the Server Pi. Images stored on the Server Pi filesystem are served to the web browser via a lightweight Node.js HTTP server. The HTML page is stored locally on the filesystem of the computer running the web browser and is not itself served by the Server Pi.

Project Deviation Statement


- It was intended that voice commands would be handled and processed in real-time. Unfortunately, AWS Transcribe was not able to provide the real-time audio to text translation that was expected with their non-streaming interface. Attempting to migration and implement AWS Transcribe streaming proved unsuccessful and ran out of development time for the project. Ultimately implemented parallel processing of Transcribe jobs to help improve voice command feedback from system.
- While the client-actor sequence diagram references metadata for images being processed at the end of transactions, this was actually performed mid-sequence.
- AWS IoT Core and Lambda were added to the Client Pi side of things for metadata transport & logic, respectively.
- Some portions of the Project 4 deliverable (such as wireframes) referenced the HTML web browser showing 25 images at a time, this number has been reduced to 10.

- The Amazon API Gateway uses HTTPS, not HTTP, for the Server Pi interface.
- The HTML Web Browser uses WebSockets and HTTP for data exchange with the Server Pi, while the original plan was to just use HTTP.
- Originally, correctness was going to be marked with a Check, Question or X mark to the left of the images in HTML page. This functionality is present in an annotation to the image label caption instead.
- Originally, there was to be a performance details page with timestamps for each classification type & each recognized or unrecognized command. This functionality was simplified to just the summary pi charts seen on the right-side of the main page.

Third-party Code Used Statement

- Client Pi
 - All references used and their-party software for each Client Pi file is linked at the top of each file. Below are their-party tools used to interface with peripheral hardware connected to the Client Pi system.
 - PiCamera <<https://projects.raspberrypi.org/en/projects/getting-started-with-picamera>>
 - Used to interface with Raspberry Pi Camera.
 - PyAudio <<https://makersportal.com/blog/2018/8/23/recording-audio-on-the-raspberry-pi-with-python-and-a-usb-microphone>>
 - Used to interface with USB attached mini Microphone
 - PyGame
 - <<https://learn.sparkfun.com/tutorials/python-programming-tutorial-getting-started-with-the-raspberry-pi/experiment-2-play-sounds>>
 - Used to interface with USB attached speaker
- Server Pi
 - Node.js aws-api-gateway-client module <<https://www.npmjs.com/package/aws-api-gateway-client>>
 - Handy module for making use of AWS API Gateway auto-generated JavaScript SDKs in a Node.js application
 - AWS API Gateway SQS Proxy tutorial <<https://dzone.com/articles/creating-aws-service-proxy-for-amazon-sqs>>
 - Brief but very useful, this article highlights the important aspects of IAM authentication (between SQS and API Gateway) as well as providing examples of proper Method setup
 - Node.js express.static function (of Express module) <<https://expressjs.com/en/starter/static-files.html>>
 - Perfect lightweight solution for serving up images via HTTP, very easy to use just referencing the documentation on this page.

Project Observations Statement

- AWS authentication proved to be a bigger hurdle than anticipated (on the client and server Pi sides). While in Project 3 we were able to persistently secure our SQS interactions using an AWS Cognito Identity Pool, we ran into issues using this same technique with Boto3 (client Pi) and the API Gateway (server Pi). Ultimately we had to just maintain local (non-git-distributed) references to our AWS access key id, secret access key, and session token, which is not an ideal solution as these only persist for a matter of hours.
- Interfacing the embedded peripherals connected to the Client Pi proved straight-forward and simple to complete due to the available third-party libraries used. Also use of the Boto3 library for interfacing with AWS components allows for data interfaces to be established quickly and easily.
- Interfacing with AWS Transcribe in order to get real time audio processing proved difficult due to authentication issues. If the team had more time to investigate this functionality this would have been a valuable addition to the project.
- While we felt interfacing S3 to the Server Pi via the API Gateway was more elegant than using API Gateway only for metadata and separate HTTP calls direct to S3 to download images to the Server Pi, it presented an unexpected issue. The only trivial way to send image files via the API Gateway is by encoding them in base64 (as opposed to binary) so that they can be passed in JSON objects. Attempting to pass the images as binary would result in image files downloaded to the Server Pi which were padded everywhere with the pattern 0xEFBD (UTF-8 “REPLACEMENT CHARACTER” aka ). Some time was spent attempting to modify the padded files on the Server Pi side, and some time was spent attempting to make API Gateway play nice with binary encoding (which is not impossible), but ultimately the best solution was to upload the images from the Client Pi as base64-encoded. Of course, AWS Rekognition wouldn't parse base64-encoded images, so in our final solution we upload two copies of each image to S3, one in each encoding.