Open-Source Internet of Things (IoT) Development Platform with Secure Cloud Communication

Group 23

Instructor: Cristian Grecu

Affiliation: The IoT group, Electrical and Computer Engineering Department, The University of British Columbia

By Jie Xiang,

Dingqing Yang,

Anthony Chui, and

Nikhil Prakash

Table of Content

List of Figures

KEY DOCUMENTS

1. Requirements Specification

The “Open-Source Internet of Things (IoT) Development Platform with Secure Cloud Communication” is a project that focuses on building a development platform for researchers and developers looking to explore and readily build IoT systems that rely upon or require secure cloud communication.

One of the primary goals of the open-source development platform is to design a system that can be freely extended to other applications that require different hardware or sensors to perform. For instance, for our system’s prototype we decided to build a “smart-home” security system that incorporates all the security measures and secure cloud communication protocols we developed and implemented within our system[1]. The smart-home security system prototype includes a motion sensor and camera for intruder detection, a remotely controlled LED in place of an electric lock, a temperature and humidity sensor, as well as a Raspberry Pi device that provides the computer interface with all these sensors as well as the facilitator of communication with the cloud.

For a seamless user interface, we have created a web application and a Slack integration [2] that serve as the dashboard to the system, allowing users to view current and past sensor data, as well as to send commands. Our system is able to communicate data between the hardware and these two platforms in a secure manner by following best practices. We are using Amazon Web Services (AWS) as our cloud provider. The systems securely communicates through both MQTT and HTTP through the WebSocket and WebHook protocol respectively to facilitate real time communication.

In addition to being a development platform, our project investigates and implements security and protection methods for IoT devices and systems that are currently vulnerable and susceptible to specific types of security-breaches or hacks that are common to IoT systems. Some of these IoT specific security concerns include snooping, spoofing, point of access attacks and unauthorized escalation of privileges. We addressed these concerns working in conjunction with a security systems PhD candidate from UBC’s SoC lab to ensure that the platform we developed addressed as many of these possibilities as possible. Fortunately, many of the concerns or security weak points of IoT systems are addressed and handled directly by Amazon AWS. For example, since passwords are authenticated with AWS Lambda, an attack that means to use brute force to to find the correct password is handled automatically via AWS Lambda

[1] “Security and privacy issues for an IoT based smart home.” http://ieeexplore.ieee.org.ezproxy.library.ubc.ca/document/7973622/ Accessed 12 March 2018.
[2] "Building internal integrations for your workspace | Slack." https://api.slack.com/internal-integrations. Accessed 1 Apr. 2018.

security protocols that limit the number of incorrect password attempts. Another important example would be using AWS Cognito for user sign-up and sign-in, which allows users to sign in with their Google or Facebook accounts and thus limiting types of attacks that rely upon the generation of many user accounts. The security concerns that were identified to not be addressed by Amazon AWS are also dealt with and detailed further in Section 2.4 ("Security") of the report.

Our project is not a consumer product that would be available online or at a local store but rather a platform for designers and researchers to develop IoT systems on with secure communication between system components and ease. Developers should be able to take advantage of the flexibility built into the system whereas researches can extend the investigation into the other and new types of security breaches, threats, and attacks on the security protocols and defences of IoT systems. The latter is a real and growing area of concern for existing and future IoT systems in both industry and academia as both defences and attacks continue to grow and evolve with the technology as it becomes increasingly integrated within our societies, industries and personal lives[3] [4].

Our choice of implementation of the prototype, keeping in mind the emphasis on security research and investigation as one of the primary goals of the project, is one of a smart-lock security system that one would commonly expect to be installed in smart-homes of today and of the future. The smart-lock, or an equivalent ON/OFF controlled LED, was chosen primarily because of its relevance to many modern systems and based on the fact that a lock's main purpose is to secure something and thus it makes sense to start by ensuring the security of a device whose main purpose is to provide security for something else. The smart-lock couples well with a camera and motion sensor for the beginnings of a more advanced smart-home security system and is paired with two peripheral sensors to measure both temperature and humidity. All the hardware components are integrated into one coherent test environment and is currently being housed in the System-On-Chip (SoC) Laboratory at the University of British Columbia. It should be noted that the smart home security extension described above is an example of a hardware system that is implemented on our IoT platform, and that our open-source platform can be and is designed to extended to virtually any sensor or hardware system that can interact with a microcontroller (Raspberry Pi) via GPIO (general purpose input/output) pins.

[3] "A roadmap for security challenges in the Internet of Things." https://www.sciencedirect.com/science/article/pii/S2352864817300214 . Accessed 2 April 2018.
[4] "On the Security and Privacy of Internet of Things Architectures and Systems." https://www.researchgate.net/publication/282075370_On_the_Security_and_Privacy_of_Internet_of_Things_Architectures_and_Systems . Accessed 3 April 2018.

Functional requirements

- F1. Minimize the communication delay through the cloud, between nodes, sensors and endpoint receivers. (Maximum and minimum latency specifications would depend on the actual implementation device. For our security system, we are using a motion-triggered security camera and we must keep the latency low enough such that the motion sensor triggers the camera to start capturing)
- F2. Build a website that allows the user to interact with the cloud with bilateral communication by displaying sensor data and sending commands via the cloud
- F3. Establish robust and dependable connection between cloud and each endpoint.
- F4. Secure endpoint devices (Both Pi and the website)
- F5. Implement an API that can be used to read sensor data and send commands
- F6. Write a Python driver for the Raspberry Pi to send sensor data and respond to commands from the cloud
- F7. Store camera images on the cloud and the Raspberry Pi device.
- F8. Persist all sensor data to the cloud

Non-functional requirements

- N1. Have relative low cost for the prototype (within $650 budget)
- N2. The website user interface should be easily understandable and readable
- N3. Have a complete manual of using the app to inform and teach the users
- N4. Clear documentation for extension by others who want to carry the project forward
- N5. Energy efficient design of hardware and driver (possibly guaranteed-continuous power source if doing something power sensitive like a smart-lock, low power consumption needed especially if device needs to be constantly powered on)
- N6. API needs to be RESTful and secure and be used by the website and Raspberry Pi
- N7. Implemented Python driver for the Raspberry Pi needs to be dependable and fault-tolerant
- N8. Raspberry Pi storage must not be congested from images
- N9. Website needs to be responsive and receive sensor data with a low response time

Constraints

- Cost/Budget Constraints: We must keep the project within the $650 allocated budget.
- Time-to-completion Constraints: We have 8 months total to finish the project.
- User Interface Constraints: The main dashboard for the system must be be built in the form of a web-app.
- User Interface and Login Constraints: The main way to login or access certain functionality of the system (security privileges) should be integrated with a common method of logging into apps or user accounts, i.e. Google and Facebook login capabilities so there is less hassle for users to create another unique account and password for our system.
- The hardware system must use a single board computer connected to multiple sensors.
- A cloud web service must be used as the foundation of the backend to facilitate wireless communication.
- Flexibility Constraints: The design of the platform's hardware and software must integrate well with the sensors used (smart home prototype hardware) while maintaining the flexibility of a design and research platform so that it may be readily extended to virtually any type of sensor or system using GPIO.
- The communication between the system's nodes and devices must be minimized in order to provide the user with real time data; this is helpful for a security system but critical for certain systems and sensors applications.
- Continuation Constraints: This project is intended to be a platform that will be built upon or used to accelerate investigations into IoT security for future Capstone or other projects. Therefore certain design procedures and easy-to-follow documentation must be exercised along with generally keeping the cost as low as possible (minimize costs, not just attempt to do it within the $650 budget) so that the platform may be easily and inexpensively reconstructed or improved by others.

2. Design Document

The overall architecture of the system is comprised of three main categories - the hardware, the backend (software), and the frontend (software). The hardware is one Raspberry Pi 3 device connected with the sensors and peripherals (motion sensor, camera, LED, and temperature/humidity sensor). The software is divided into the two remaining major categories - the frontend and backend. Just as the name suggests the frontend is where the user interacts with the system. The frontend software is responsible for for sending user commands to the IoT client as well as displaying current and past data for the user. The mechanism for doing this is via a client's host website and integration with the popular team communication platform "Slack". The backend is the chunk of the software that is hidden from the user and is responsible for persisting the data given by hardware devices to the could, and providing the data in response to queries. To drive our backend code, we are using Amazon AWS cloud services. The following diagram visually depicts the three categories, the subcomponents and functionality of each of the categories as well as a basic, high-level illustration of how each of the categories interact and communicate within the overall architecture of the system.

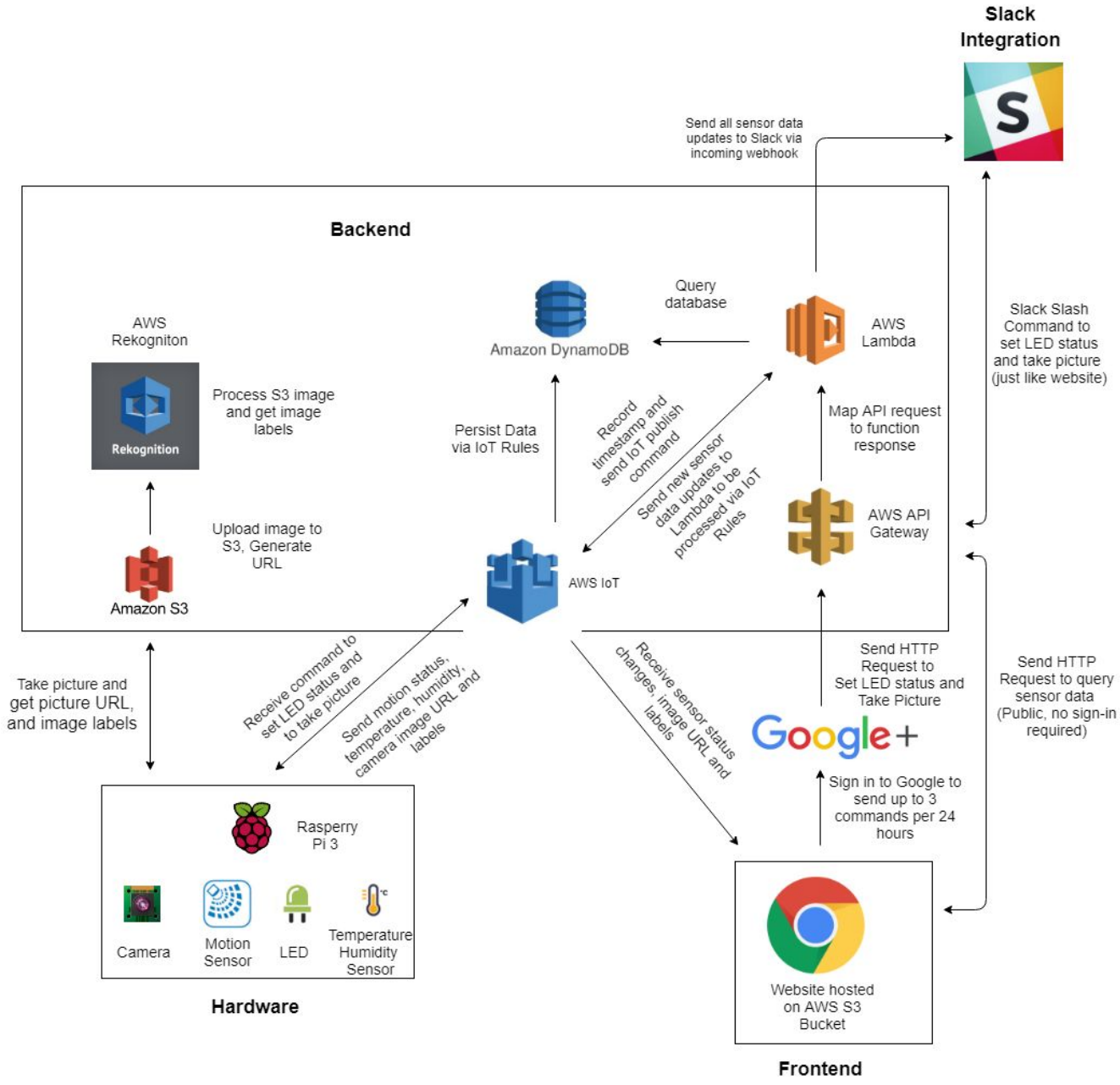

Figure 1. System Architecture.

There are 3 cloud clients: the website, Slack, and the Raspberry Pi device. The website interacts via API Gateway (HTTP) for commands and AWS IoT (MQTT) for subscribes. Sensor updates trigger an IoT rule that invokes a Lambda that sends the sensor update to an incoming webhook to Slack. Slack can also invoke Slash Commands (HTTP) to send commands via API Gateway. The Pi device responds to commands via API Gateway and emits data via AWS IoT.

2.1 Hardware

The single board computer that we are using is a Raspberry Pi 3. It supports a variety of sensors, which is essential for our platform to be extensible. As shown in the Figure. 2, the Pi is connected with an LED, a camera, a PIR motion sensor, and a temperature and humidity sensor. These sensors are just one example of the many configurations and systems that can be realized with our development platform. The system is able to perform the following functionalities with the following sensors.

- Motion detection: Our system hardware detects motion via the PIR motion sensor and communicates to AWS IoT when both motion is detected and when that motion is gone.
- Capturing Pictures: The Raspberry Pi communicates to both AWS IoT and the camera module, in addition to responding to commands to take a picture when motion is detected or when a command is sent from the website or Slack via slash commands.
- Lock Status Display: Our system displays the status of the LED (which represents the “locked” and “unlocked” states of our virtual lock) and responds to commands from users to set the LED status via the same channels as mentioned above in “Capturing Pictures”.
- Temperature and Humidity Measurement: The temperature and humidity sensors send measured data at regular and user-specified intervals to be published as plots on the webapp through AWS IoT and the Raspberry Pi.

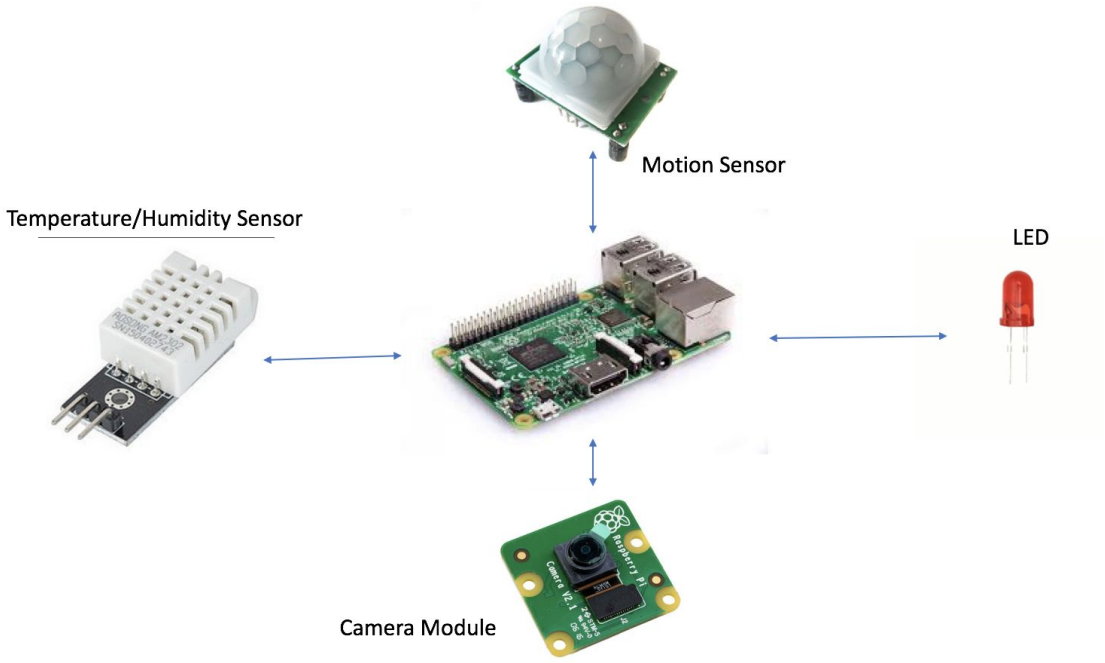

Figure 2. Hardware Connection of Raspberry pi

2.1.1 Lock vs LED

As mentioned above, in place of a electrically controlled lock we use an LED controlled with a single control bit for the two states of the LED (ON and OFF). The reason for this is that a digital lock is controlled and operates in a very similar way to an LED. An electric lock can turn on and off (switch between states of LOCK and UNLOCKED) by the appropriate electrical signal in the exact same way a standard LED does. We did not purchase a commercial lock for development purposes, since a commercial lock is something that inherently prevents other people from manipulating and tampering with it. Moreover, representing an electric lock with an electrically equivalent yet simpler and easier to procure LED allowed the design efforts to be focused on platform development rather than a single component of the hardware prototype. This design decision was approved by and done with consultation from the clients who agreed with the conclusions stated above.

2.2 Backend

For the backend of our system, we are using numerous cloud services from AWS, namely: AWS IoT DynamoDB, Lambda, API Gateway, S3, Rekognition, and Cognito. The following is a brief description of each AWS service that we are using, before we dive into the specific details of our backend implementation.

1. *AWS IoT*

AWS IoT is an Internet of Things platform that enables the user to easily connect devices to Amazon AWS Services and other devices, both on and offline. The core AWS IoT features include the AWS IoT Device SDK (to connect hardware, mobile, and web application to AWS IoT), the Device Gateway (serving as the entry point for IoT devices connecting to AWS), the Registry, Device Shadow and the Rules Engine.

2. *DynamoDB*

DynamoDB, is a fast and flexible cloud NoSQL database service. It is known for cost-effectiveness, it's flexible data model, reliable performance and automatic scaling of throughput capacity (for instance our system has a high throughput when motion is detected and a relatively low throughput otherwise - it would not be cost or resource effective to have a high throughput capacity running at all times).

3. *Lambda*

Amazon AWS Lambda is a compute service that lets users run code without the overhead cost of purchasing or maintaining their own individual servers. Lambda only charges you when your code is running. This coupled with the fact that AWS Lambda executes your code only when needed and scales automatically, allows for the cost effective running of systems with the variability of only a few sparse requests a day to thousands of requests per second.

4. *API Gateway*

Amazon API Gateway is platform that facilitates and helps developers publish, maintain, monitor and secure APIs (Application Programming Interfaces) without managing servers.

5. *S3*

S3 is an abbreviation for Amazon Simple Storage Service. It is a data storage infrastructure that can provide scalable and inexpensive data storage because it emphasizes maximizing the benefits of scale. I.e. On a much smaller scale; 10 businesses can save a lot of money by taking advantage of scale by pooling resources and having a single shared data center rather than 10 individual but less capable ones.

6. *Rekognition*

Amazon Rekognition is computer vision for image and video analysis. Using Rekognition, you can detect and identify objects, people, text, scenes as well as high-level facial analysis and recognition. It is particularly useful for developers who would like to use image analysis and computer vision without prior or deep knowledge of machine learning.

7. *Cognito*

Amazon Cognito allows developers to add and integrate user sign-up/sign-in and access control (user privileges) to web and mobile apps. Sign-up/sign-in supports popular social identity networks/providers so users do not need to create a unique account for each web or mobile application they use."

As mentioned above, the Raspberry Pi device will be receiving commands and sending sensor statuses to certain topics in AWS IoT. The website and our Slack channel will be notified of any sensor status changes by subscribing to the same topics in AWS IoT. All sensor data published to AWS IoT are persisted to DynamoDB by using IoT rules[5]. AWS Lambda functions are used to query DynamoDB for past and current data with a given sensor parameter and a time range. We are also using Lambda functions to send commands like toggling and LED or taking a picture to the Pi device by publishing messages to topics in AWS IoT. The AWS API Gateway is then used to expose the implemented Lambda functions as an API endpoint. The

[5] "Rules for AWS IoT - AWS IoT - AWS Documentation." https://docs.aws.amazon.com/iot/latest/developerguide/iot-rules.html. Accessed 11 Feb. 2018.

front-end of the website then uses these API endpoints to retrieve sensor data to visualize in a user-friendly interface, as well as to send commands. The Pi device also uses these API endpoints to send data back to the cloud, such as responding with the new LED status, or the URL of the picture just taken. Our implementation of Slack Slash Commands[6] also use these API endpoints so that users in our Slack channel can send commands. Table 1 below shows all the APIs we have created.

Function	Functionality	API Endpoint Example Usage
CurrStatus	Get the current status of all sensors	GET https://uniqueid.execute-api.us-west-2.amazonaws.com/prod/currstatus
Status	Query past data for a given sensor and time range (Unix time)	GET https://uniqueid.execute-api.us-west-2.amazonaws.com/prod/status/motion?timestart=0
SetStatus	Send command to set status of LED	PUT https://uniqueid.execute-api.us-west-2.amazonaws.com/prod/setstatus/led
TakePicture	Send command to take picture	POST https://uniqueid.execute-api.us-west-2.amazonaws.com/prod/takepicture
GetPicture	Get the URL of the last picture taken, as well as the associating computer vision labels	GET https://uniqueid.execute-api.us-west-2.amazonaws.com/prod/getpicture
PublishImage	Update to the cloud the URL of the last picture taken, as well as the associating computer vision labels	PUT https://uniqueid.execute-api.us-west-2.amazonaws.com/prod/publishimage

[6] "Slash Commands - Slack API." https://api.slack.com/slash-commands. Accessed 5 Apr. 2018.

UpdateTemperature	Update to the cloud the temperature and humidity	PUT https://uniqueid.execute-api.us-west-2.amazonaws.com/prod/updatetemperature
CheckPassword	Check whether the password is valid to use for API calls that need it	POST https://uniqueid.execute-api.us-west-2.amazonaws.com/prod/checkpassword
CheckGoogleOauth	This Lambda function is automatically invoked prior to each API request to send a command. It checks the provided OAuth token from Google to see if it is valid. If so, decrement the number of commands the user has available. Otherwise, reject the call.	Not meant to be used in API Gateway.
SlackSlashCommand	This is invoked whenever the user uses a Slash Command in Slack. Not meant to be used anywhere other than through Slack (a token is checked when making this call)	POST https://uniqueid.execute-api.us-west-2.amazonaws.com/prod/slackslashcommand
Slack	An IoT Rule invokes this Lambda function whenever a sensor publishes updated data to AWS IoT	Not meant to be used in API Gateway.

Table 1. List of Lambda functions and their API endpoints

We are implementing what is known as a serverless architecture[7] for our backend. The combination of API Gateway and Lambda for maintaining a scalable API and running code on the cloud respectively without provisioning servers is what constitutes a serverless architecture in AWS. There are many benefits to a serverless architecture compared to typical backend architectures. The main reason is for simplicity, because we do not need to manage infrastructure and server management. Although our backend is still being run on a server, AWS handles all of the server management for us. The other approach is to use AWS EC2, which provides cloud machines that we would use as our servers. The reason we choose not to go

[7] "Serverless Computing - Amazon Web ...." https://aws.amazon.com/serverless/. Accessed 26 Nov. 2017.

with this approach is because we do not want to manage code backup, server maintenance, security updates, or downtime, which are all problems associated with managing servers. In terms of cost, EC2 pricing is based on machine uptime, whereas serverless frameworks like Lambda and API gateway are billed based on requests[8]. Since our server obviously needs to be up constantly, by using the serverless approach instead, we would be billed on a more granular basis and save money. Ultimately at a high level, what we really care about is mapping API calls to functions and returning the output. By choosing the serverless approach, we could reduce complexity and overhead and save development time and costs. Below is an image of how we are implementing the serverless architecture.

[8] "AWS Lambda – Pricing - Amazon Web Services." https://aws.amazon.com/lambda/pricing/. Accessed 11 Feb. 2018.

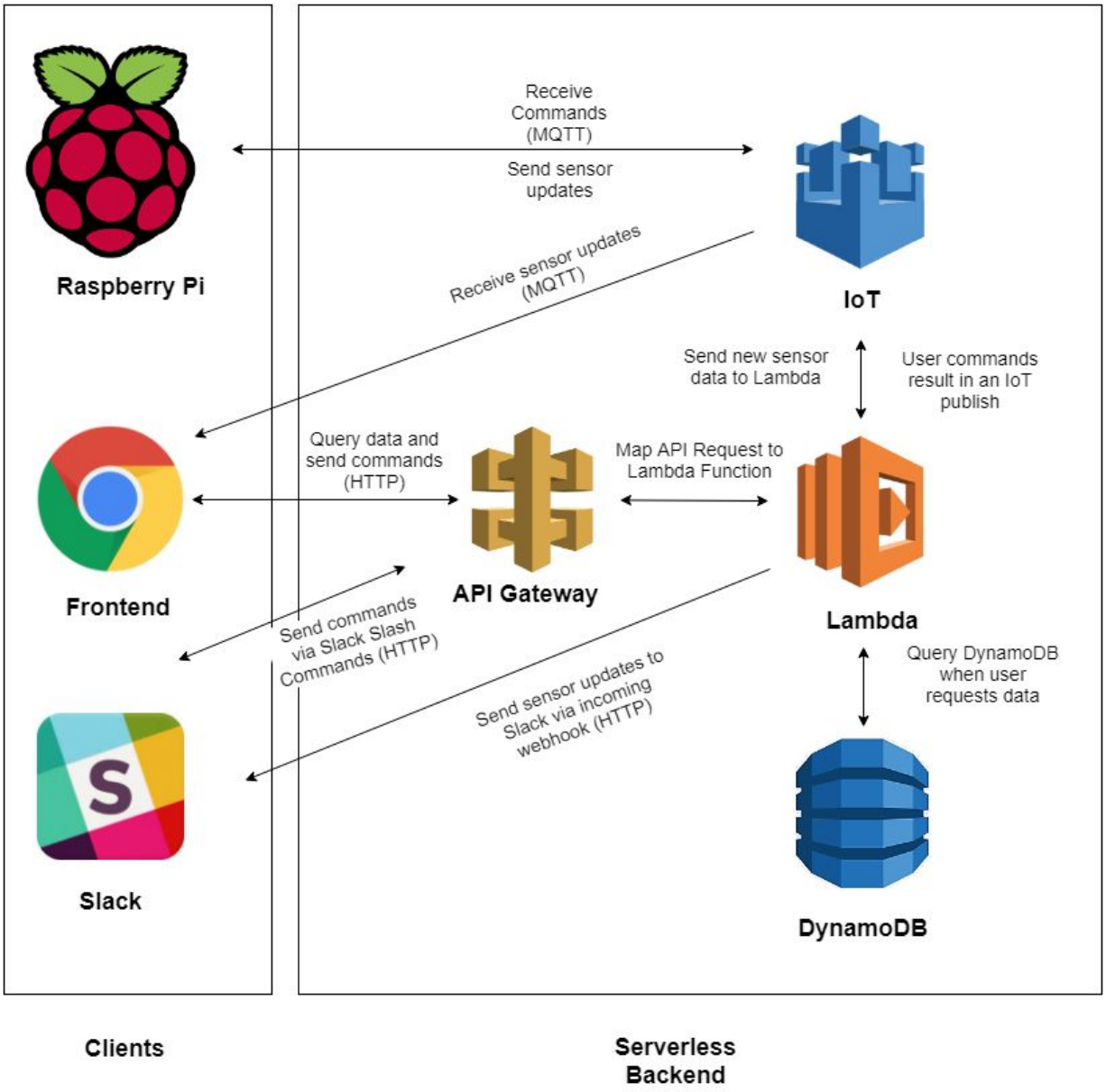

Figure 3. Serverless Architecture.

No machines are being explicitly managed by us as servers. API Gateway lets us define APIs, Lambda lets us define functions, IoT lets us manipulate sensor data, and DynamoDB persists data without thinking about servers

There are multiple choices we could use for the database. The client suggested MongoDB, but we will be using DynamoDB because of the built-in integrations with AWS to simplify development of the backend. For example, AWS IoT has what is known as IoT rules that

can automatically persist all messages sent to IoT to DynamoDB without writing code[9]. This functionality does not exist for MongoDB, and by using it we would need to spend more time writing code to persist data, increasing the complexity of our project.

We are also using AWS S3 and Rekognition to parse camera images. A camera image is taken when the user clicks the associating button on the website, or if the motion sensor detects motion. Either way, a message is sent to a topic on AWS IoT, and the Pi device having subscribed to that topic, responds by taking a picture. The picture is uploaded to S3 to be stored on the cloud, and a URL is also generated. AWS Rekognition then parses the image on the cloud to analyze it for computer vision labels, specifically humans. After receiving the image URL and the associating computer vision labels, the Pi devices sends them via an API Gateway endpoint that publishes the message to AWS IoT. Both the website and Slack channel will then be notified of the image and display them accordingly. If a human is detected, a Slack notification is displayed for everyone, and an alert dialog is shown on the website. We do not want to send the raw image data to AWS IoT for the Slack and website to parse. This is because AWS IoT publishes have a maximum data size of 128kb[10] that is too low for high quality images. This is why we are using a cloud image hosting service like S3 as a medium. Another advantage is that Rekognition is integrated with S3, and can interact with images already uploaded to S3, rather than uploading the image again. Below is an image of the images and the computer vision labels being sent to Slack, and alerting the channel that a human is detected.

On average, the amount of delay between sending a message from one node and receiving it in another is about 86 milliseconds. This is the minimum delay for any implemented feature that relies on publish and subscribe. The measurements were performed on nodes located in Vancouver, B.C against the backend server located in Oregon. This delay is within the order of magnitude of a typical ping to the server from Vancouver (30 milliseconds) and will feel real-time in the eyes of a user. This delay is the result of a tradeoff between implemented security features as described in the security section of the report.

---

[9] "Creating a DynamoDB Rule - AWS IoT - AWS Documentation." http://docs.aws.amazon.com/iot/latest/developerguide/iot-ddb-rule.html. Accessed 26 Nov. 2017.
[10] "AWS Service Limits - Amazon Web Services - AWS Documentation." https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html. Accessed 11 Feb. 2018.

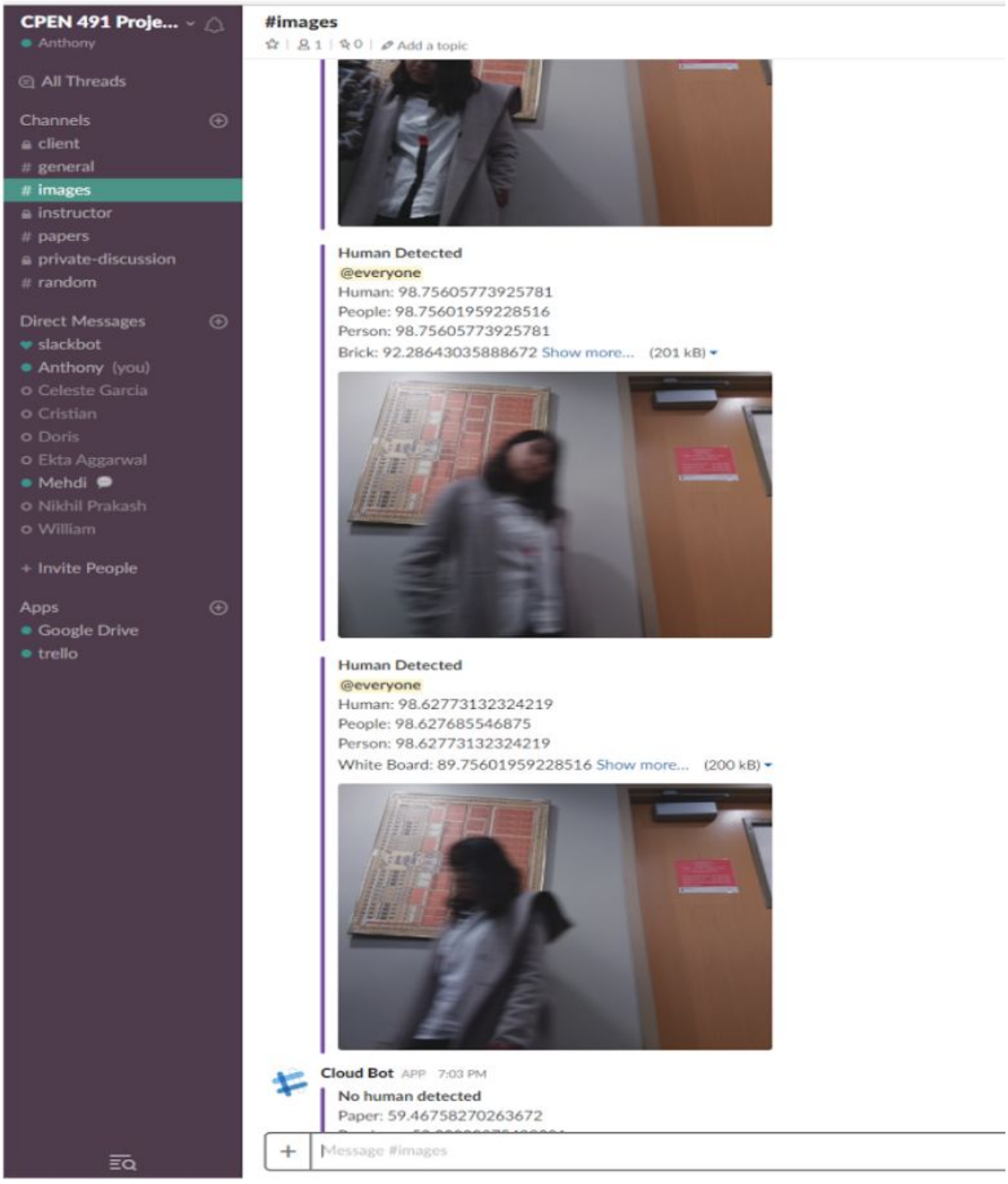

Figure 4. Slack Integration Interface

This shows Slack receiving camera images from the cloud and notifying everyone in the channel if a human is detected

2.3 Security

From the perspective of the website, it interacts with the cloud through either the API Gateway or subscribing to topics through AWS IoT. As mentioned earlier, the former is for retrieving past data and sending commands to the Pi device and the latter is for receiving sensor updates in real time. This means there are two connections that need to be secured.

In the case of communicating to AWS IoT, all traffic is enforced to be encrypted over TLS[11], preventing vulnerabilities like man in the middle attacks. To connect to AWS IoT, we use AWS Cognito to grant credentials for the website to access. We create a federated identity[12] in Cognito that comes with a pool ID, and we give it a role for unauthenticated users with certain permissions. The website then simply provides the pool ID and receives the respective permissions that we gave the federated identity. For our use case, we want any user to access our website and be able to read data, but not write to data. This means we allow them to subscribe to certain topics, but not publish to them. We want to lock down the permissions to exactly what the website can do with our AWS account, and no more and no less. This way, even if a user finds out the pool ID, the only actions they can perform are exactly what the website can perform. In this case, they would only be able to retrieve real-time data from the sensors, but are not able to send commands to the sensors. Below is an image of the IAM[13] permissions for the website.

[11] "Security and Identity for AWS IoT - AWS IoT - AWS Documentation." https://docs.aws.amazon.com/iot/latest/developerguide/iot-security-identity.html. Accessed 2 Apr. 2018.
[12] "Amazon Cognito Federated Identities - Amazon Cognito - AWS ...." https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-identity.html. Accessed 12 Feb. 2018.
[13] "Identity and Access Management (IAM) - Amazon Web Services (AWS)." https://aws.amazon.com/iam/. Accessed 11 Feb. 2018.

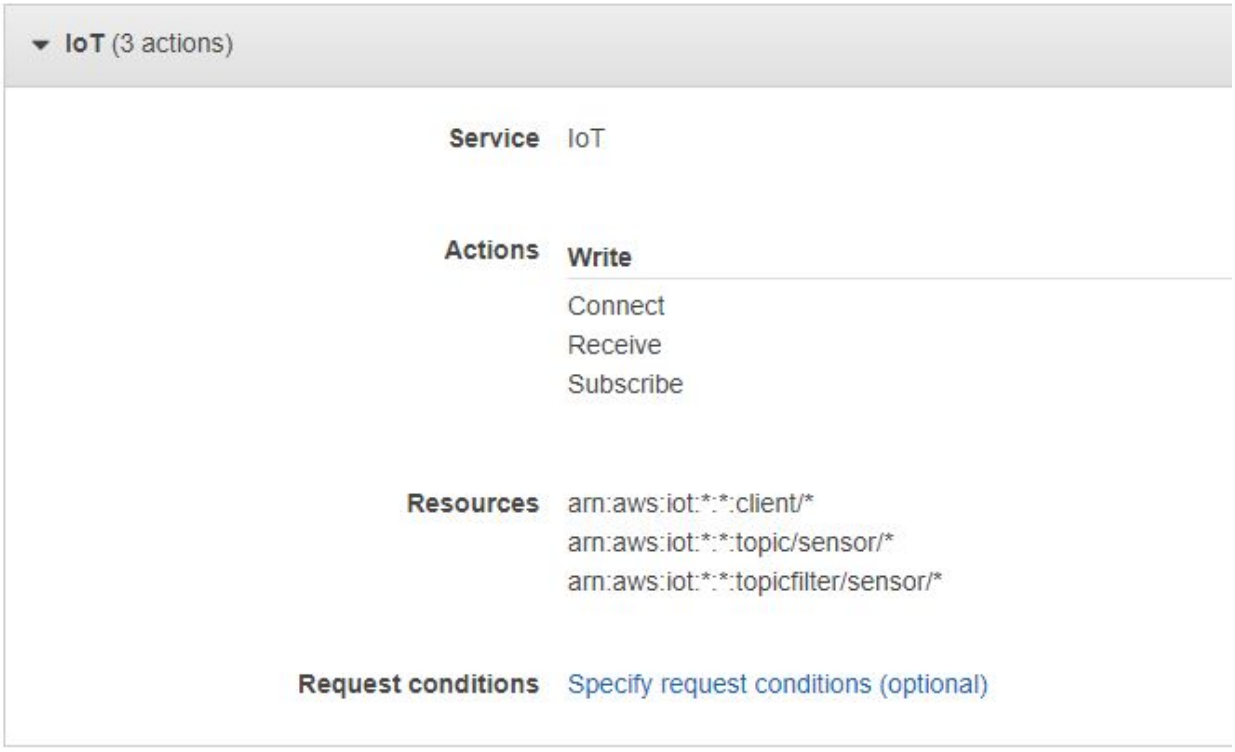

Figure 5. Required Permissions to AWS.

This shows the permissions that the website has to AWS. The website can only use AWS IoT and connect, receive, and subscribe to certain topics.

In the case of making an API request to the API Gateway, all endpoints are secured with HTTPS[14] by default, preventing vulnerabilities like the man in the middles attacks just like with communicating to AWS IoT. Our API consists of calls that retrieve data as well as sending data and commands. The former is open to the public, since we want sensor data to be publically viewable. The latter however, requires signing into Google. These APIs, such as setting LED status, or taking a picture, require a Google OAuth token, received from signing into Google, in the *authorizationToken* header of all API requests. A Lambda function, *CheckGoogleOAuth*, that checks the token, is then automatically invoked for all APIs that require authentication. This Lambda function validates the token, and records the user's account information into a

[14] "API Gateway FAQs - Amazon AWS." https://aws.amazon.com/api-gateway/faqs/. Accessed 2 Apr. 2018.

DynamoDB table, and decrements the number of commands they have left. We allow users to make 3 commands every 24 hours. As per best practices from the official Google documentation[15], we send this token rather than raw information about the user from the frontend. The token is essentially formed by a encrypting a concatenation of the user's public information. We need to receive the encrypted data and decrypt it in our backend to ensure the validity of the data. Client code can be manipulated by the user, and if we were to simply send the raw information to the backend, we could be receiving false data.

For administrator access, we also allow a master password in place of a token in the header of each API request. This allows administrators to bypass the limits of commands per day to their own system. Obviously, we do this master password validation on the backend, which is the Lambda function in this case. Javascript code in the client side can be inspected by anyone, and it would not make sense to do password validation there. The client's website also uses HTTPS, which securely transfers the password to AWS, without allowing for vulnerabilities like man in the middle attacks.

[15] "Authenticate with a backend server | Google Sign-In for Websites ...." https://developers.google.com/identity/sign-in/web/backend-auth. Accessed 2 Apr. 2018.

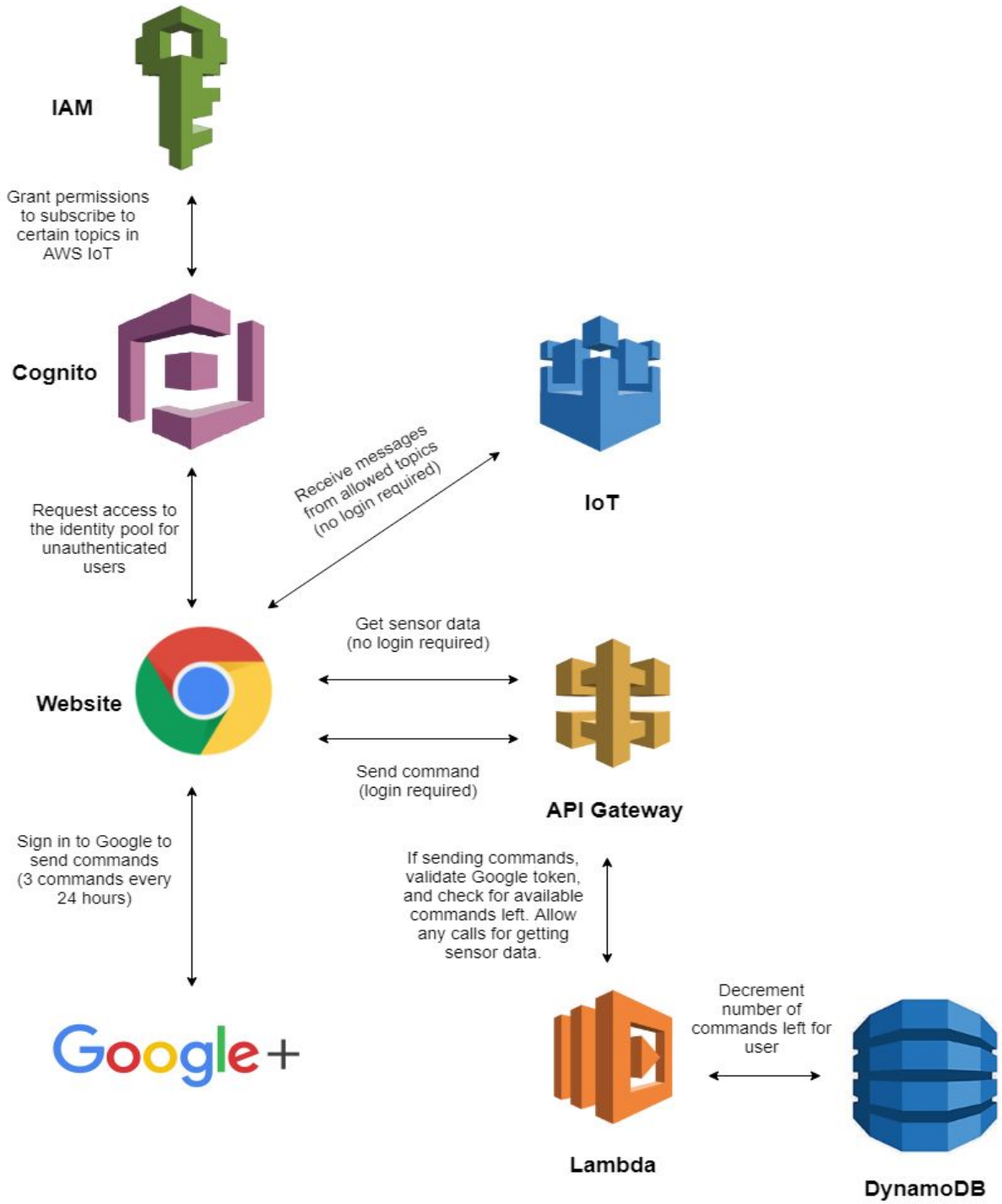

Figure 6. Security Mechanisms of the Website connecting to AWS IoT and API Gateway. Sending commands to API Gateway requires the user to be signed to Google. Querying sensor data does not require to be signed in. All users can connect to AWS IoT to receive sensor data in from sensor topics in real time, but cannot publish to topics.

In implementing the Slack integration into our system, we need to protect our Slack incoming webhook[16] URL. In this case, we process all messages sent to Slack in the backend via Lambda functions. IoT rules are used here to process all new sensor updates, which then forwards the messages to a Lambda function that sends updates to Slack. Again, we are processing the Slack integration in the backend rather than the frontend. Implementing this feature on the frontend would reveal the Slack webhook URL, resulting in users manipulating our Slack channel. We also store the password as an environment variable in Lambda, so that we do not accidentally commit it to our Github repository. In implementing the Slash Command integration in Slack, we need to check that it was our Slack channel that sent the Slash Command to our backend Lambda function. To do so, we check the accompanying token in the API request and verify it to be the one was generated when creating the Slash Command. We follow best practices from the documentation of Slack, and we again save the token as an environment variable in Lambda for aforementioned reasons.

The Raspberry Pi device connects to the cloud via AWS IoT, API Gateway, S3, and Rekognition. However, unlike the environment of a web browser, the Pi device is able to store files locally. We use X.509 certificates[17] to authenticate our device to use AWS IoT. This is in line with best security practices by AWS[18]. On the AWS IoT console, we can then fine-tune the exact permissions that the Pi device can have access to. We can also revoke access to these certificates, and monitor the activity of each certificate from the AWS IoT console. The advantages of using X.509 certificates compared to other authentication methods such as username and password is that the secret key never leaves the device. In using the other AWS services such as S3, Rekognition, and API Gateways that don't accept certificates, environment variables are used to store the API keys necessary. Again, none of these keys can then accidentally be committed to our Github repository.

---

[16] "Incoming Webhooks - Slack API." https://api.slack.com/incoming-webhooks. Accessed 11 Feb. 2018.
[17] "X.509 Certificates and AWS IoT - AWS IoT - AWS Documentation." https://docs.aws.amazon.com/iot/latest/developerguide/managing-device-certs.html. Accessed 11 Feb. 2018.
[18] "AWS IoT Authentication - AWS IoT - AWS Documentation." https://docs.aws.amazon.com/iot/latest/developerguide/iot-authentication.html. Accessed 11 Feb. 2018.

2.4 Frontend

We created a website that serves as the dashboard to the system, allowing users to visualize sensor data as well as send commands. It is created with the intention that it is easily extensible for integrating new sensors or adjusting graphing capabilities. We are using vanilla Javascript with the support of Chart.js for plotting sensor data. This is enough for the functionality of the website. The website only needs to display sensor statuses in real time from AWS IoT, and retrieve past data and send commands via API Gateway. We are using a single page dashboard design to accomplish this. We thought of using frameworks like Angular or React, but we believe it provides too much extra complexity to the website. On the other hand, by using a graphing framework like Chart.js, development time is significantly reduced. We only need to provide data points, and graphing options, and Chart.js handles the entire plotting of the graphs for us. This also enables users to build interface easily when adding new sensor in the system. We use this framework to display both binary digital data for ON/OFF and analog measurement coming from temperature and humidity sensor. We also have separate plots for real time data and past data according to a time range, as shown in the website. Additional functionalities such as limiting maximum number of data points displayed on a graph are implemented as well and are easy to be applied for new sensor plots.

As mentioned, we implemented a Google sign in feature to prevent abuse of controlling sensors on the website. The website has capabilities to toggle the LED and send commands to take picture, which should be of limited access to anonymous users. The user signs in to Google through a provided button from the official Google documentation[19]. After signing in, we can access public information from any user to keep track of them. Google uses cookies to keep users signed in, so we provide a sign out button if the user wishes to do so.

[19] "Google Sign-In for Websites | Google Developers." https://developers.google.com/identity/sign-in/web/. Accessed 2 Apr. 2018.

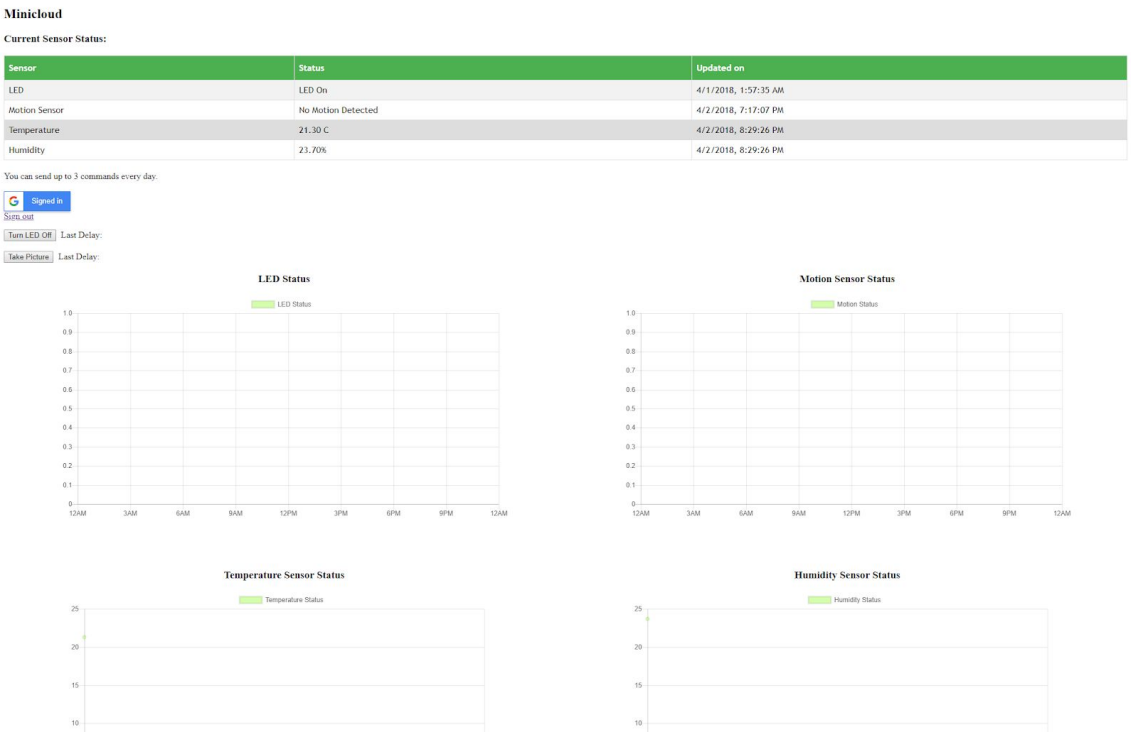

Figure 7. Platform Frontend

Our frontend can get sensor updates in real time, plot data, and receive camera images with computer vision analysis. To send commands, the user must sign in to Google.

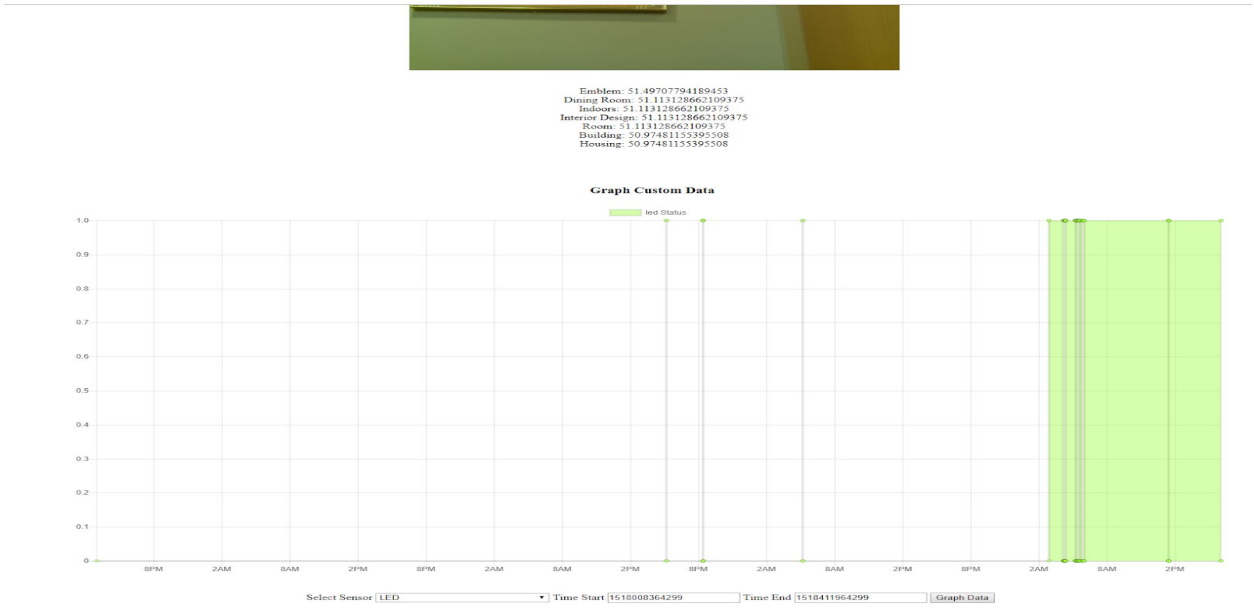

Figure 8. Graphing Custom Data

The frontend can also graph a chosen sensor's data according to a time range

3. Validation Document

In this section, we provide the corresponding validation approaches for each of the functional (labeled as F1, F2, etc) and nonfunctional (labeled as N1, N2, etc) requirements illustrated in the requirements section.

- To meet F1, measure the communication latency between website and Raspberry Pi device to cloud to ensure all the latencies are below a few seconds.
- To meet F2, send messages from website to our endpoint devices to turn on and off LEDs. Also, check if the website an obtain and display the correct message from our Raspberry Pi device.
- To meet F3, test the communication between extreme conditions (low temperature, strong mechanic vibration)
- To meet F4, attack the endpoint devices by a variety of approaches to see how secure our endpoint devices can be.
- To meet F5, test the implemented API and see if measured data make sense and command sent can be received at backend.
- To meet F6, driver cloud communication is tested by seeing if it receives command and responds correctly to them.
- To meet F7, check if images are properly stored on devices and the cloud.
- To meet F8, manually validate data obtained in the cloud and compare the data obtained in the devices
- To meet N1, calculate and compare the cost between various design choices and optimize the cost as much as we can.
- To meet N2 and N3, invite people from a variety of background to try our system and collect feedback about user experiences and clarity of the manual.
- To meet N4, invite other students and clients to identify if there is sufficient information for them to carry over the project.

- To meet N5, measure and compare the power consumption between our system and other possible design choices to make sure we are using the most power efficient system.
- To meet N6, ensure the right HTTP verbs are used for each API operation, and that any API operation that manipulates data requires authentication.
- To meet N7, monitor the driver for a period of usage and make sure it does not crash
- To meet N8, make the image folder size greater than 100 mb, and check if the oldest image is deleted to reduce its size
- To meet N9, a burst of data is created for testing purposes and website responsiveness is then measured and observed

Validation Approach in Details:

Hardware Test

Test Name	TEST-01: Capability to detect motion
Description	Run the python code for PIR motion sensor on raspberry pi, and then move hand in the front of motion sensor.
Expected Result	The flag LED will be turn on only when the motion happens in front. In other words, expecting a low sensitive motion sensor, which only detect the motion in front of it.
Actual Result	Widely range of motion being detected. Any tiny motion in the corner will trigger flag LED.
Modification	Covering the motion sensor with cylindrical mask.
Modified Result	Motion sensor accurately capture the motion in the front.

Test Name	TEST-02: Capability to take picture when motion is detected

Description	Run the python code for PIR motion sensor and camera module on raspberry pi, and then move hand in the front of motion sensor.
Expected Result	When the motion flag LED is turning on, a picture with moving hand will be taken by camera.
Actual Result	The camera module captured the picture with moving hand.

Test Name	TEST-03: Transmit message from Pi to AWS cloud
Description	Run the python code for PIR motion sensor and camera module on Raspberry Pi, and then move hand in the front of motion sensor.
Expected Result	AWS will receive the updated motion status and image captured by camera
Actual Result	AWS receives the captured image and motion is detected

Frontend Test

Test Name	TEST-4: Get Current Status
Description	Use API to get the current status of all sensors
Expected Result	All table and charts displays the current status of sensors
Actual Result	All table and charts is able to display the current status of sensors

Test Name	TEST-5: Set LED Status
Description	Use API to change the status of LED. See if our view adjusted properly
Expected Result	Real-time data changes as the way we manipulate LED

Actual Result	Real-time data in charts and table changes as expected

Test Name	TEST-6: Take Picture
Description	Use API to take a picture. See if a picture is returned to the website.
Expected Result	Picture and computer vision labels shows up on website after a short delay
Actual Result	Picture and computer vision labels shows up on website as expected

4. List of Deliverables

- Complete source code written on all Raspberry Pi, website, and cloud
- Public source code available on Github with an open source license that is to be decided
- Overall documentation in how to use the source code (readme file), along with comments within the code
- One Raspberry Pi 3 device, and sensors of our choice, all mounted on a test bench
- Instructions on setting up AWS services used for the project
- Working implementation of a secure end to end communication between website, Slack, Raspberry Pi, and the cloud
- Website that can receive real time sensor updates, graph past data, and send commands
- Python driver for the Raspberry Pi device that can receive commands from the the cloud and emit sensor updates to the cloud
- An API that can retrieve sensor data and send commands, with security implementations for the latter

References

1. “Security and privacy issues for an IoT based smart home.” http://ieeexplore.ieee.org.ezproxy.library.ubc.ca/document/7973622/ Accessed 12 March 2018.
2. "Building internal integrations for your workspace | Slack." https://api.slack.com/internal-integrations. Accessed 1 Apr. 2018.
3. “A roadmap for security challenges in the Internet of Things.” https://www.sciencedirect.com/science/article/pii/S2352864817300214 . Accessed 2 April 2018.
4. “On the Security and Privacy of Internet of Things Architectures and Systems.” https://www.researchgate.net/publication/282075370_On_the_Security_and_Privacy_of_Internet_of_Things_Architectures_and_Systems . Accessed 3 April 2018.
5. "Rules for AWS IoT - AWS IoT - AWS Documentation." https://docs.aws.amazon.com/iot/latest/developerguide/iot-rules.html. Accessed 11 Feb. 2018.
6. "Serverless Computing - Amazon Web ...." https://aws.amazon.com/serverless/. Accessed 26 Nov. 2017.
7. "AWS Lambda – Pricing - Amazon Web Services." https://aws.amazon.com/lambda/pricing/. Accessed 11 Feb. 2018.
8. "Creating a DynamoDB Rule - AWS IoT - AWS Documentation." http://docs.aws.amazon.com/iot/latest/developerguide/iot-ddb-rule.html. Accessed 26 Nov. 2017.
9. "AWS Service Limits - Amazon Web Services - AWS Documentation."
10. Security and Identity for AWS IoT - AWS IoT - AWS Documentation." https://docs.aws.amazon.com/iot/latest/developerguide/iot-security-identity.html. Accessed 2 Apr. 2018.
11. "Amazon Cognito Federated Identities - Amazon Cognito - AWS ...." https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-identity.html. Accessed 12 Feb. 2018.
12. "Identity and Access Management (IAM) - Amazon Web Services (AWS)." https://aws.amazon.com/iam/. Accessed 11 Feb. 2018.
13. "API Gateway FAQs - Amazon AWS." https://aws.amazon.com/api-gateway/faqs/. Accessed 2 Apr. 2018.
14. "Authenticate with a backend server | Google Sign-In for Websites ...." https://developers.google.com/identity/sign-in/web/backend-auth. Accessed 2 Apr. 2018.
15. "Incoming Webhooks - Slack API." https://api.slack.com/incoming-webhooks. Accessed 11 Feb. 2018.
16. "Slash Commands - Slack API." https://api.slack.com/slash-commands. Accessed 2 Apr. 2018.
17. "X.509 Certificates and AWS IoT - AWS IoT - AWS Documentation." https://docs.aws.amazon.com/iot/latest/developerguide/managing-device-certs.html. Accessed 11 Feb. 2018.
18. "AWS IoT Authentication - AWS IoT - AWS Documentation." https://docs.aws.amazon.com/iot/latest/developerguide/iot-authentication.html. Accessed 11 Feb. 2018.
19. "Google - Amazon Cognito - AWS Documentation." https://docs.aws.amazon.com/cognito/latest/developerguide/google.html. Accessed 11 Feb. 2018.

Appendix - Source Code and Documentation of System Setup

Repository: https://github.com/mbiuki/minicloud

Functionality	**Github Repository Location**
Python code that the Raspberry Pi devices run to communicate with IoT. Includes instructions on how to setup and run the code.	https://github.com/mbiuki/minicloud/tree/master/RaspberryPi
Contains all backend AWS Lambda functions. Includes instructions on how to setup all backend AWS services.	https://github.com/mbiuki/minicloud/tree/master/WebServices
Frontend code for the website in vanilla JS that consumes the API endpoints and subscribes to messages from AWS IoT. Includes instructions on how to setup the frontend with AWS Cognito, AWS SDK, and Google Plus Sign-in.	https://github.com/mbiuki/minicloud/tree/master/Frontend