IoT Lab 4

Overview: Cloud Infrastructure

You will learn how to develop a cloud-based infrastructure to support your operations. You will leverage *Amazon Web Service's IoT platform* to build your infrastructure.

One warning: AWS IoT costs money. If you perform these steps right, you should be able to fit within their free tier. However, be continuously aware of what resources you have deployed and **remember to tear them down** when you are done to avoid incurring more costs[1].

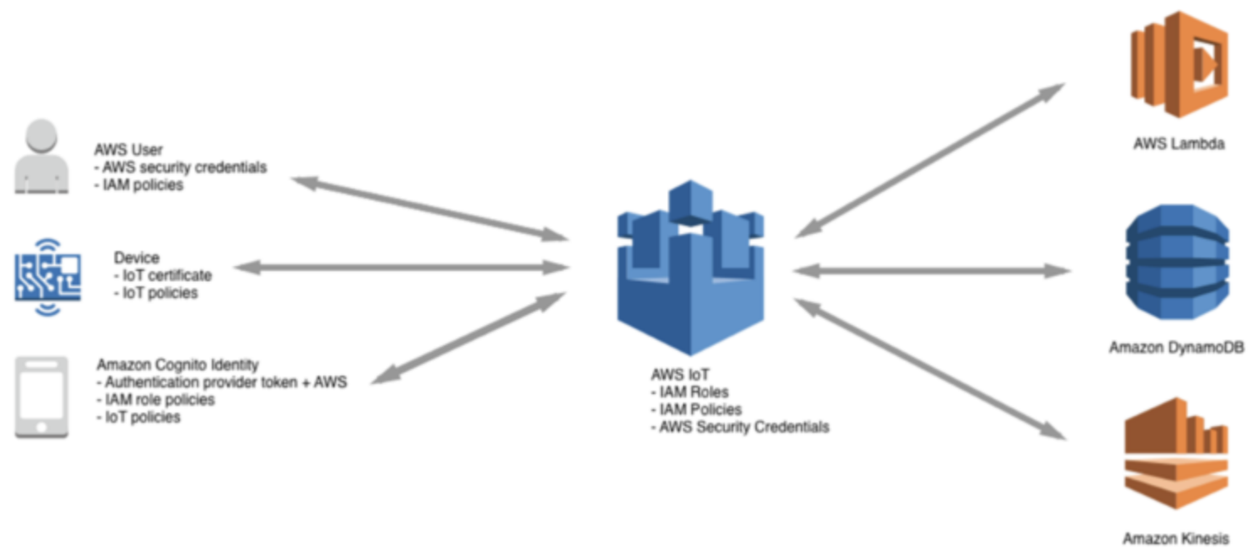


Figure 1: Cloud IoT deployment model

[1]You can set budgets alerts ([Creating a budget - AWS Cost Management]) so that if you are being charged X amount, you'll receive an email. Eg you can set up a $1 budget so that you're "immediately" alerted if you exceed the free tier usage limits.

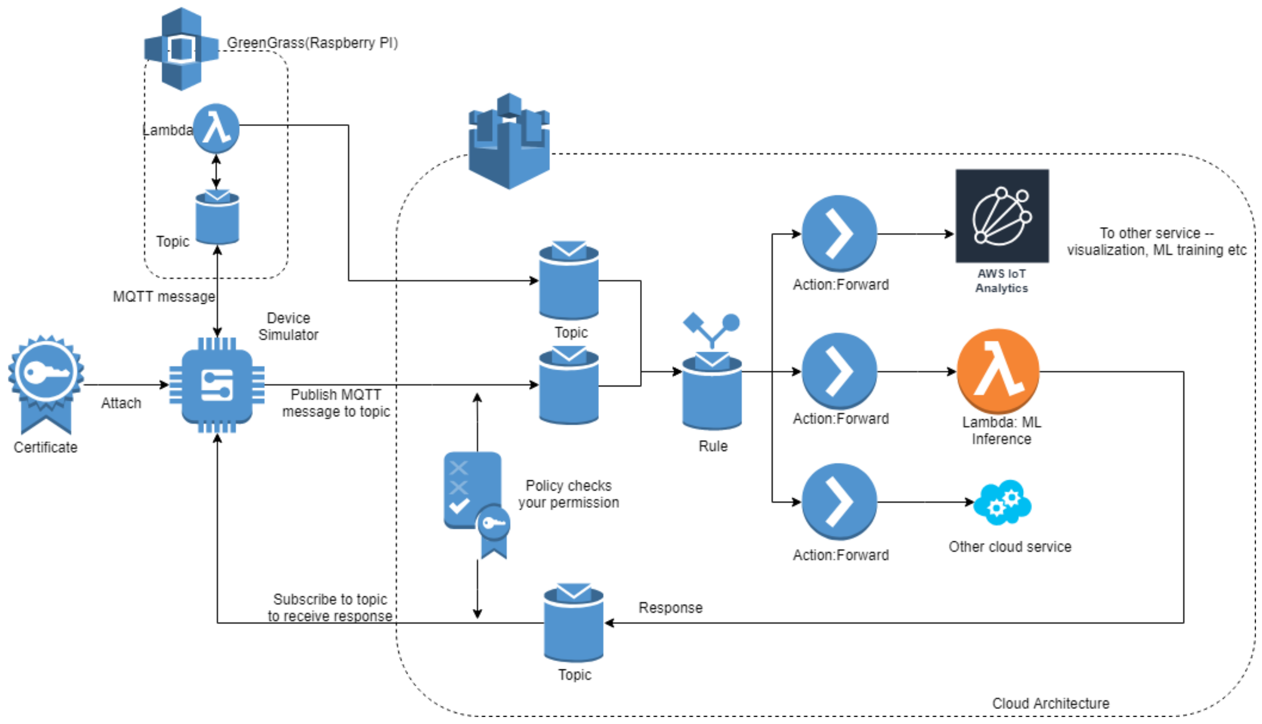


Figure 2: Device Emulator

What we are going to do is create an infrastructure where we have 120,000[2] cars. You will construct an infrastructure on AWS IoT to store their data. You will then construct a new processing framework which monitors cars' emission level and traffic hotspots (optional). This framework will be very useful to monitor engine conditions of your cars and the volume of traffic on the road (optional).

[2] In practice, you may run into memory/cpu limits on your computer - it's ok to do less than this. Just do as many as you can. You should be able to run at least 500-1000 per laptop - bottlenecks are CPU and bandwidth.

1. Build the Cloud

Within AWS IoT, then, please construct an infrastructure using all the IoT components, including a Hub, GreenGrass, IoT Core, IoT Analytics, IoT Device Defender. In particular, please follow the steps below to build your infrastructure.

1.1 Background on AWS IoT

There will be a few terms that appear frequently in this document. Here are some brief explanations for them.

Thing: A “Thing” is a record in the cloud side that represents one physical device. Inside this record, you can add custom information like serial number, product type etc for better manageability. A thing needs to be tied with a certificate (which is in turn associated with a security policy) for it to authenticate with the IoT server.

Publish/Subscribe Model: A messaging pattern where the sender does not specify its receivers. Instead it publishes the message to “topic”. Receivers who subscribe to “topic” will receive messages on that “topic”. The sender does not need to know who the receivers are. Wikipedia: Publish-Subscribe Pattern

Topic: As the name suggests, a publisher can send message to a topic, and other clients who subscribe to this topic will receive message. AWS: MQTT Topics

Broker: The agent that handles the actual routing in the publish/subscribe model. In AWS IoT, the message broker is the IoT core. You can find the broker address in IoT panel -> setting. You will need this broker address later.

QoS: There are three levels of QoS as defined in MQTT protocol. QoS 0 stands for at most once. QoS 1 stands for at least once. QoS 2 for exactly once (AWS IoT does not support QoS 2: MQTT - AWS IoT Core). You are encouraged to try different QoS in this work.

Rule/Action: IoT core handles incoming messages from devices, but to process and respond to the messages, we need other services. A rule specifies what service to be performed for what type of incoming messages. E.g. for messages with topic X, forward them to service Y. Action is the execution of a rule.

Lambda Function: Lambda functions are *microservices* provided by AWS. It offers cheap on-demand function invocations and handles scaling automatically. It is billed as memory used times the amount seconds of execution (MB * s per month). However, as tradeoff, the Lambda function would have limited compute power and library support. You are encouraged to try out other options such as creating some EC2 servers for your backend. For the basic setup, we will use Lambda functions to process the IoT messages.

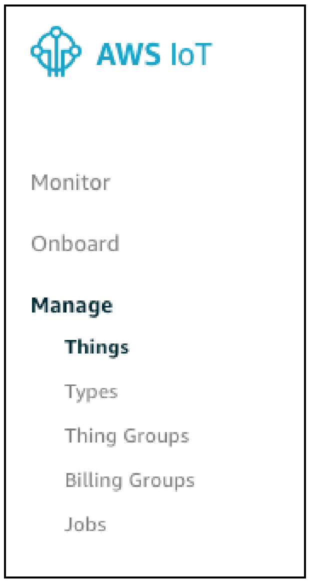
Feel free to read further (e.g., on Wikipedia) about MQTT, and get a rough understanding of the protocol and the corresponding publish/subscribe model.

1.2 AWS Account and AWS IoT Setup

Follow this link for the guide to setting up your AWS account for AWS IoT, including registering for a new account if you haven't already: AWS: Account Setup

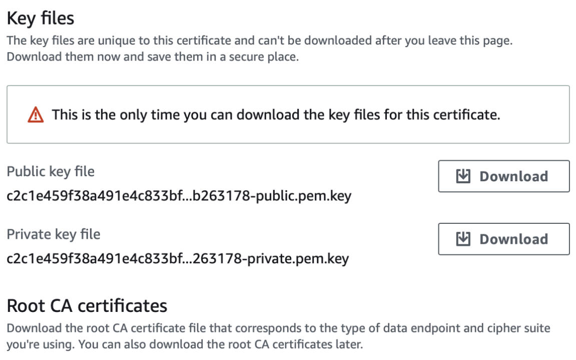
1.3 Device (thing)

Under IOT core panel. Click Things -> Create -> Create a single thing



You can follow the link: AWS: Create Resources

This step essentially creates a registry for the physical device in the IOT core, after registering the "thing", you will be directed to download *three* key files. Your device will use these key files to authenticate with the AWS IoT server.


Key files

The key files are unique to this certificate and can't be downloaded after you leave this page. Download them now and save them in a secure place.

This is the only time you can download the key files for this certificate.

Public key file
c2c1e459f38a491e4c833bf...b263178-public.pem.key

Private key file
c2c1e459f38a491e4c833bf...263178-private.pem.key

Root CA certificates

Download the root CA certificate file that corresponds to the type of data endpoint and cipher suite you're using. You can also download the root CA certificates later.

Note: The thing that you create here does not necessarily have to be used in your simulation. It is only to help you understand what a thing is.

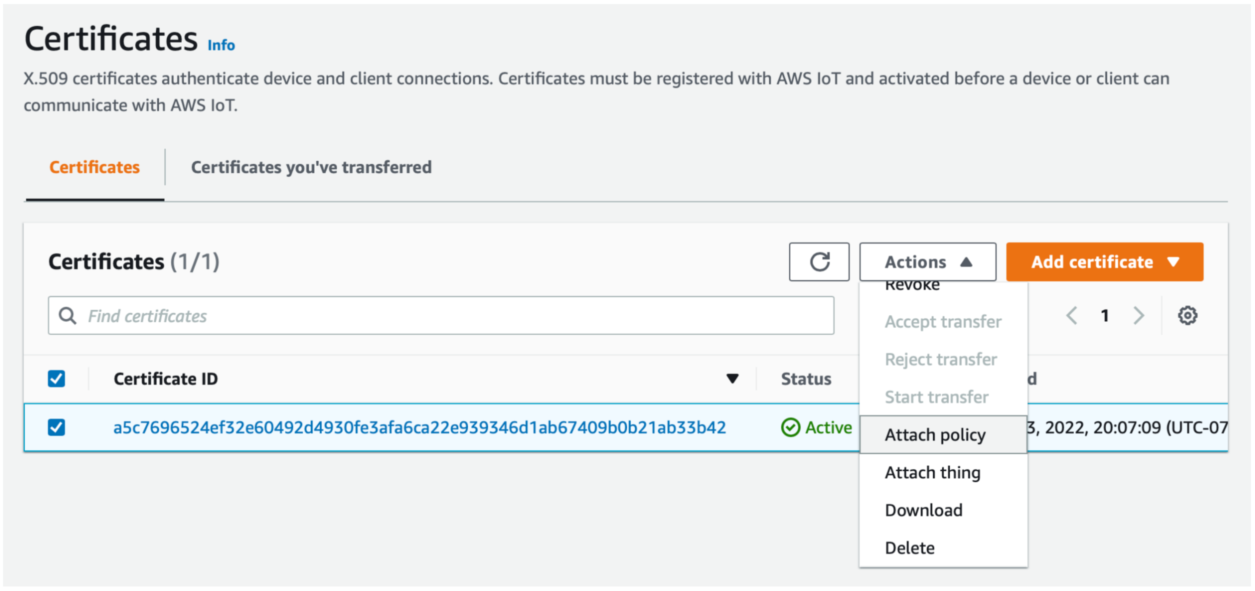
1.4 Configure rules

After you have created the Thing, the next step would be to create "rules" for your device communications. These rules specify what service is to be performed for a given type of incoming messages. You will need to configure one policy and one action rule for your device. A policy specifies what type of communications from the device will be allowed and what resources the incoming message can invoke (something like permissions for incoming messages).

Follow this link for more details: AWS: Create Resources
Note: You need to at least allow publish, subscribe, connect, receive[3] for your devices.

When you have created a policy, attach the policy to the certificate of your devices; you can do this in the **Secure -> Certificates** section of the AWS IoT console by selecting the certificate of your Thing and choosing Actions and then “Attach Policy” action as seen below

[3] Make sure to give iot:Receive permissions as well as iot:Subscribe in the policy.


Certificates Info

X.509 certificates authenticate device and client connections. Certificates must be registered with AWS IoT and activated before a device or client can communicate with AWS IoT.

Certificate ID	Status
a5c7696524ef32e60492d4930fe3afa6ca22e939346d1ab67409b0b21ab33b42	Active

Don't forget to download the certificates, and keep them in a safe place! It'll be used to authenticate your devices with the IoT Core.

1.5 Create many devices

After completing the previous steps, create a Thing group.

1. Select **Manage** -> **Thing Groups** -> **Create thing group**
2. Select **Create static thing group**
3. Enter a name for your Thing group
4. Select **Create thing group**

You can add the Thing you previously created to your new Thing group by navigating to the Thing, selecting the **Thing groups** tab, and **Add to group**.

It would be very time-consuming to click and create one-by-one. To accelerate your progress, use AWS IoT's API to create a number of Things programmatically. Our suggestion is to use AWS SDK.

One reference is: [class IoT.Client - Boto3 Docs 1.21.25 documentation]()
To be able to connect to your IoT core, we need to set our **configuration** file:

Create this dir and file: `~/.aws/credentials`,[4] following the instruction here: Credentials — Boto 3 Docs 1.10.25 documentation[5][6]

When following instructions to create the configuration file, you'll be asked to provide a secret-access-key. See the following to retrieve it: Where's My Secret Access Key? | AWS Security Blog.

To **create each thing**: the following steps should be performed:
create_thing()
create_keys_and_certificate()
attach_policy()
attach_thing_principal()
add_thing_to_thing_group()

Github Example: aws-create-thing-boto3/createThing-Cert.py at master

Section Notes:
(1) Alternatively, you could use the AWS CLI (i.e., CloudShell) to issue the equivalent commands. The CLI can be faster as no additional packages are required, and you do not need to worry with additional authentication. Access cloud shell from the icon in the top right of the AWS Console. It may take 1-2 minutes to fully launch the shell. It's also possible to install AWS CLI on Windows\Linux\Mac, and install the python boto3 library (pip install boto3) to directly run python programs that work with AWS services, from your local machine.[7]

(2) Make sure you change the value of defaultPolicyName and also pass your values for thingArn and thingId before you run the code. These values you can get anytime by opening your ThingGroup you created for this lab.

[4] More background: https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html and https://docs.aws.amazon.com/sdk-for-php/v3/developer-guide/guide_credentials_profiles.html.
[5] Note that for a Windows PC, you would have to set up your environment variables, as listed in the configuration page. See https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-envvars.html on how to configure this.

[6] Here is a cross platform way, so you don't need to set environment variable outside this program:
```
client = boto3.client('iot', aws_access_key_id='<access_key_id>',
aws_secret_access_key='<secret_access_key>', region_name='<region, e.g.
us-east-1>')
```

[7] For a short video on installing AWS CLI and configuring the credentials see https://www.youtube.com/watch?v=n3KacV0UlSM and then use the command 'aws iot list-things' to check if AWS CLI has been configured correctly (the previously added thing should be visible).

(3) If you're unable to find the thingId for the group you created by clicking on the ThingGroup, try creating a ThingGroup on the AWS CLI: create-thing-group — AWS CLI 1.24.0 Command Reference

1.6 Use the device simulator

Ideally, when you have created the certificates for your device, you will need to upload these certificates to your physical devices and write[8] an MQTT application for connecting to the IoT core.

However, since we don't have physical devices, we will use the emulator as a substitute. Please specify your certificates directories and number of devices in the client code. (lab4_emulator_client.py) In your lab4_emulator_client.py file, please replace the data path with your current vehicle data folder ("your_folder/vehicle{}.csv"). Please see the vehicle data in 2.1.

Note: this sample code is written using python MQTT client. You will need to install the package *AWSIoTPythonSDK*. If you'd like to use another language, you can write your own version following the same setup as in the client.py.

Note the messages in the client are not sent out periodically since doing that may overflow the free tier limit. After you launched client.py, press *s* to let all devices send one message each time.

Additionally, you can verify messages are being sent over the wire through the MQTT test client in the AWS IoT console. Enter the topic filter in the console that you specified in the Python script in the subscribeAsync and publishAsync methods, then run the Python script. You will see the messages appear in the console.

Note 1: A call to `AWSIoTMQTTClient.configureCredentials()` needs to have the Amazon root certificate authority .pem file presented. This can be acquired directly from Amazon here. Or download the PEM file directly here.
Note 2: The lab4_emulator_client.py file presented with this lab is only a starting point. While the overall sequence of events is correct, you may require substantial changes to certain functions to make it work with your project.

[8] This is the ideal case when you're deploying IoT devices (like sensors) in the field. You write an MQTT app that runs on the device.

In our case, we're providing you with a client emulator script that already has MQTT publish/subscribe code. You'll have to modify it, of course.

Note 3 (Endpoint): The Lab4 emulator client requires an AWS IoT Core endpoint URL to be specified. You can find this by navigating to the AWS IoT Core console, and then choose **Settings** from the left navigation. Then copy the **Endpoint** from the **Device data endpoint** section.

Another useful resource can be found here, which also covers Note 1 about configureCredentials and Note 3 on the endpoint. It's also useful to skim through this link to help in understanding how to modify the given lab4_emulator_client.py file.

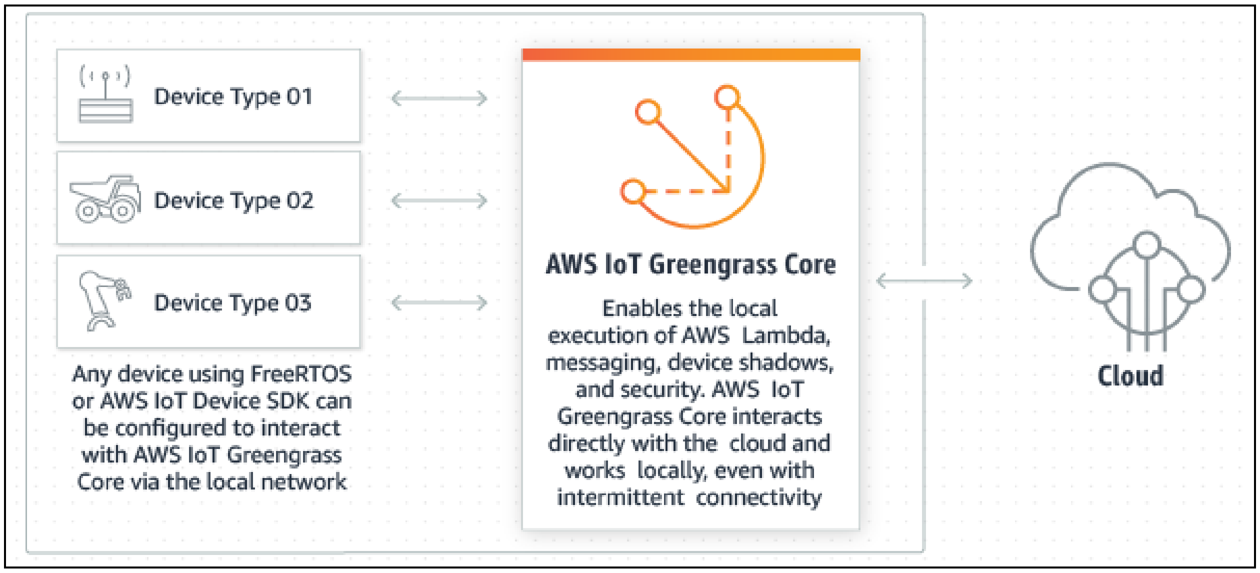
Checkpoint: At this point, you should try different things with the emulator. Set up multiple devices as subscribers and some devices as publishers. Publish different data from the devices and see how it's received by the subscribers. This is a good checkpoint to understand how the publish/subscribe model works with AWS IoT Core.

1.7 GreenGrass

After following the steps above, you may have a basic understanding of how data is sent from devices that have access to AWS IoT Cloud. But what about local devices that do not have direct access to AWS IoT Cloud? How do different local devices talk to each other on a local network? (e.g your car wants to communicate with your smart watch). We need a solution that extends AWS capabilities to local devices. This is where *GreenGrass* comes in.

GreenGrass is software that extends AWS capabilities to local devices. It runs on your local device but it allows the developers to do all the settings and configurations on the cloud. The local device running GreenGrass software is called a **GreenGrass Core**. The configuration set on the cloud defines how local devices communicate with each other or communicate with the cloud through the GreenGrass Core. You can deploy your configurations with an easy click on the Deploy button.

It should be noticed that in this lab, you do not need to write much code on GreenGrass Core. The code you need to write should only be implementing the simulator. The following graph shows the relation between GreenGrass Core, Device, GreenGrass Group and Cloud.


Device Type 01
Device Type 02
Device Type 03
Any device using FreeRTOS or AWS IoT Device SDK can be configured to interact with AWS IoT Greengrass Core via the local network
AWS IoT Greengrass Core
Enables the local execution of AWS Lambda, messaging, device shadows, and security. AWS IoT Greengrass Core interacts directly with the cloud and works locally, even with intermittent connectivity
Cloud
Figure 5: Basic Architecture of AWS IoT GreenGrass

The communication flow is as follows:
GreenGrass ‘Device’ (Emulator Client) -> GreenGrass Core (RPi) -> AWS IoT Cloud

In this lab, we are using the Simulator Client as a device. You may use either a Raspberry PI with (AWS documentation) or an EC2 host working as GreenGrass Core (AWS documentation). The messages are sent and received from ‘Topics’ (See section 1.0 for definition). It is recommended that different Topics should be defined for sending and receiving information on the same device (will be useful when you have to send results back to your devices).

There are several important components in GreenGrass:

GreenGrass Group: A GreenGrass Group is a set of one GreenGrass Core and several GreenGrass Devices. GreenGrass Devices can communicate with each other or the AWS IoT Cloud through the GreenGrass Core.

GreenGrass Core: GreenGrass Core can process data from local devices, securely routes messages, and more.

GreenGrass Device: A Group contains up to 200 local Devices that can securely exchange messages.

Subscription: A subscription includes a topic (defined in section 1.0), source and destination. Both source and destination can be Services (In this lab, our service is IoT Cloud), Devices and Lambdas. By deploying the subscription to the GreenGrass Core, we enable the communication through source to destination through the topic. E.g We can define source to be our simulator, destination to be IoT Cloud and topic to be ‘device/data’. That means our simulator can send messages to IoT Cloud through topic ‘device/data’ topic.

Steps to enable the communication flow:

Requirements: A Raspberry Pi 3 Model B with a 8GB microSD card is recommended (tested) or an Amazon EC2 instance.

(1) Setup system on your GreenGrass Core (Raspberry Pi or EC2, both would work) Detailed steps . Note: Do NOT run the recommended "sudo rpi-update b81a11258fc911170b40a0b09bbd63c84bc5ad59" command on a Raspberry Pi 3 B+, that version is too old to work with the B+. If you do this and end up with an unbootable Pi 3 B+, copy the contents of the "/boot.bak" folder from the root/data partition into the boot partition on the SD card, making sure to replace all the files that are there .

(2) Create a GreenGrass Group. Register your GreenGrass Core in the GreenGrass Group. Install and Start GreenGrass Software on your GreenGrass Core.
Detailed steps. Be sure to check that your /greengrass/config/config.json file has been updated correctly. You may need to manually configure - reference this link for detailed instructions.

(3) Create and deploy Subscriptions on your GreenGrass core. You may already have created Lambda Functions in sections before, in this case, refer to the tutorial on the part of including the existing Lambda Function into your GreenGrass Group.
In this step you can try to send and receive messages with GreenGrass core.
Detailed steps

(4) Register Devices in GreenGrass Group. (Also similar to the section 1.4 Create Many Devices) After registering, you will get three cert files. These files help identify your simulator as a Device defined in the GreenGrass Group. (In our case, these three cert files identify our one laptop as a device in the GreenGrass group)

Detailed steps

EC2 Users: If you run into a Connection Refused issue when executing python3 basicDiscovery.py args, refer to the manual endpoint entry at the bottom of the manual. Please use the public EC2 IP address, not the private EC2 IP address.

(5) Modify *basicDiscovery.py*. Send and receive test data with self-defined JSON message data by incorporating additional fields/arguments.

[Checkpoint]

Important:

(1) Above is the basic idea of how GreenGrass works. Please follow the tutorial (specifically, follow Modules 1-4) for how to deploy Lambda functions and thereby make connections between devices and AWS Cloud. This tutorial describes step 1 -- step 4 in detail.

(2) For Step 5, please read through and try to understand the file `**basicDiscovery.py**` which is provided in tutorial Module 4 and try to modify the code to send your own defined information type.

(3) For Step 5, if you are running Greengrass Core on an EC2 instance, it is important to run the device simulator script (i.e., *basicDiscovery,py*) from the EC2 instance directly OR manually enter the endpoint by entering your EC2 instance's public IP address into the **Connectivity** settings (at port 8883) – you will need to remove the first two entries for this to work (i.e., the 127.x.x.x entries). Keep in mind if you turn off your instance, you will be assigned a new public IP at startup and you will need to redo this step. Both the core and the simulator must be run from the same network. Use `yum install python3-pip` and then `pip3 install AWSIoTPythonSDK`

(4) For Step 4, if you are re-using existing Things created in step 1.3, it is important that you add a greengrass rule to the policy associated with your Things. You can do this by modifying the JSON of your existing policy with the rule below, then setting the new policy version as active.

```
{
   "Effect": "Allow",
   "Action": "greengrass:*",
   "Resource": "*"
 }
```

(5) If you have issues with connection to your raspberry pi, make sure to check your firewall rules set up in a previous lab to ensure the needed ports are open.

(6) For Raspberry Pi, if you are able to start Greengrass daemon but it fails to run some components, you can check what is failing under "....../greengrass/ggc/var/log/system/runtime.log"

(7) If you see one of the errors in the log as "The following cgroup subsystems are not mounted: devices, memory" or "The following cgroup subsystems failed to unmount: devices, memory", ensure you are running on CGROUP not CGROUP2. You can choose CGROUP by adding the following in /boot/cmdline.txt: systemd.unified_cgroup_hierarchy=0

2. Data Inference

2.1 Process incoming messages in Lambda

In this part, we will be using Greengrass and Lambda to achieve local device communication and data processing. We will be writing some scripts to analyze the emission level of vehicles. The emission analyzer is a useful diagnostic tool used by modern cars which reveals the exact composition of the exhaust gasses that are coming out of the engine. And you can see if the fuel mixture is running rich or lean, and if a misfire is ignition, compression or fuel related. Analyzing emissions could also be used to diagnose a wide variety of engine performance problems, including checking the operating efficiency of a catalytic converter, verifying the operation of the upstream and downstream oxygen sensors — especially when the OBD II system has set codes for a lean fuel mixture, low oxygen sensor switching activity or a converter code. So monitoring cars' emission is very useful for troubleshooting and repairing today's vehicles. [9]

Emission could be analyzed in a variety of configurations, including 4-gas and 5-gas units, and portable and stationary units. Four-gas units read hydrocarbons (HC), carbon monoxide (CO), carbon dioxide (CO2) and oxygen (O2). Five-gas units read all of these plus nitric oxides (NOx).

In this lab, rather than implementing a fully working emission analyzer, we simplify the problem to analyze the carbon dioxide (CO2) emission level of the vehicle. All you need to do is to send your vehicle emission data (row by row) to the cloud, write a lambda function to calculate the maximum CO2 emission values of that particular vehicle and send the results back to the car.

You can use the CO2 reading to determine if a problem is fuel, ignition or compression related by creating a momentary rich or lean condition and noting the change in the CO2 reading. If CO2 is low because the fuel mixture is rich, the level should drop even more when the mixture is made richer by feeding some propane into the engine. If the fuel mixture is rich and you create a lean condition by pulling off a vacuum hose, CO2 should rise until the engine starts to lean misfire. If artificially enriching or leaning the mixture fails to change CO2 at all, the problem is not fuel related, but is due to ignition misfire or a compression leak.

You'll be using this vehicle data for the lab and we have included emission data for five vehicles. We will use AWS Lambda functions to calculate the maximum of CO2 emission of that vehicle. We will provide this deployment package and you will need to modify the process_emission.py file to process incoming messages.

Let's get started! Please perform the following steps

[9] https://www.tomorrowstechnician.com/tool-rules-sniffing-out-engine-problems/

1. Download deploy_package.zip and modify the process_emission.py
2. Create a Lambda function through AWS console
3. Deploy your modified Lambda package
4. Add your Lambda function to your Greengrass group
5. Create proper subscriptions and configuration for your Lambda.
6. Create proper subscriptions for your vehicles.

For detailed instruction see 1.7.3 and 1.7.4.
Useful Links: Module 3 (part 2): Lambda functions on AWS IoT Greengrass
Also: Lambda Function Handler in Python

Checking out log file located in
/greengrass/ggc/var/log/user/aws-region/account_id/lambda_name.log is very helpful for debugging lambda function

2.2 Return your results back to device

Still inside process_emission.py. After you finish processing the message , you will need to publish the outputs back to a subscriber device, using the publish/subscribe MQTT model. Write your code to send these results back to subscribed devices.

One important question for you: MQTT uses the publish/subscribe model, which means all subscribers to a topic will receive the message published to that topic. Should a device user receive the output for another user? Please consider this when writing your code to return results back to the device.

Once you have done the previous steps, you can run the simulator to see the results.

3. Use IoT Analytics to visualize the Data

Data visualization is important for a cloud application. Both the user and the developer can gain a deep understanding of the data by gathering and plotting them. In this part we will build a data archive and visualize the data along with the results on the AWS IoT Analytics.

3.1 Background

AWS IoT Analytics: part of the AWS IoT ecosystem, which allows the user to create a channel to send data to and store them in a SQL database.

AWS Sagemaker: fully-managed platform that enables developers and data scientists to quickly and easily build, train, and deploy machine learning models at any scale. We will only use the Jupyter notebook it provides to do the data visualization.

Jupyter Notebook: http://jupyter.org (in case you don't know what it is)

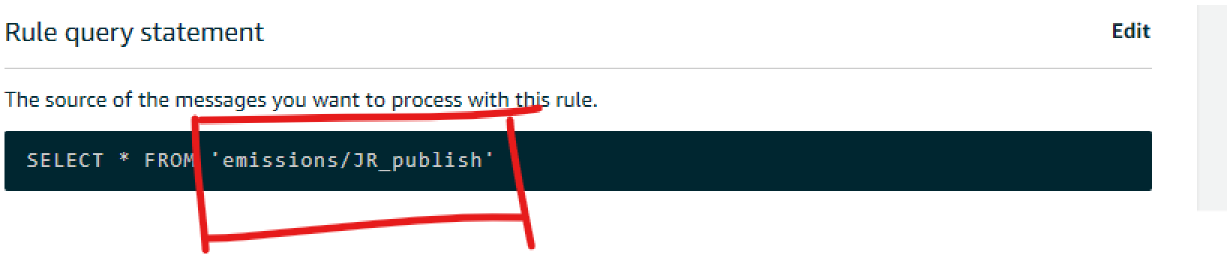
You should do this part by following this tutorial (outdated, doesn't cover all the settings but still generally relevant):
Getting started with AWS IoT Analytics

When you go through the wizard and still don't see data populating, it might be because of a misassigned role in your analytics rule. Double check your analytics rule (automatically created if you used the analytics setup wizard) in AWS IoT Core, and click the edit button next to your "send to IoT Analytics Channel". Just because the Action says the IAM role is set (which you did during the setup wizard) doesn't mean it's actually set - you can confirm whether the role is set or not by clicking "edit" and verifying the IAM role in there.

3.2 Setup data path

Set up channel, pipeline, and data store in Iot Analytics. For those settings it doesn't mention, either provide a reasonable value or just leave it as default. In addition, **It is important to record things like IAM role and topic as you go**, since you may not be able to see them again after you created them.

You can reference the topic you originally used to set up your channel by going to IoT Core > Act>Rules (a rule is automatically created if you used the analytics setup wizard). Under 'Rule query statement', you'll see the query pulling from the topic:



The source of the messages you want to process with this rule.

```
SELECT * FROM 'emissions/JR_publish'
```

3.3 Set up the dataset and test if you can receive data

Next, set up the data set and test if you can receive data. In particular, set up a sql data set under AWS Iot Analytics, and test if you can receive data by using the test section in IoT core.

3.4 Create Jupyter Notebook[10]

To visualize the data create a Jupyter notebook on AWS Sagemaker (see background or the tutorial in 3.1) over your dataset. You also need to give its IAM role the permission to access data in your dataset. You can do this by creating a policy in the IAM Management console.

3.5 Visualize the data

Open your Jupyter Notebook and do some visualization of the data. You can load the data from your data set using pandas:

```
dataset_url = client.get_dataset_content(datasetName =
dataset)['entries'][0]['dataURI']
data = pd.read_csv(dataset_url)
```

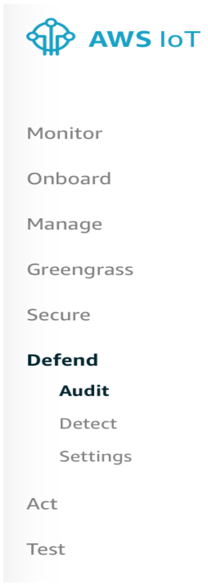
[10] There is also an online version of Jupyter Notebook host by Azure called Azure Notebook. You can use that instead if you like. It's like Jupyter Notebook + Google Drive.

4. AWS IoT Device Defender (Optional, Extra Credit)

AWS IoT Device Defender is an IoT security service that allows you to audit the configuration of your devices, monitor connected devices to detect abnormal behavior, and to mitigate security risks. It gives you the ability to enforce consistent security policies across your AWS IoT device fleet and respond quickly when devices are compromised.

IoT fleets may consist of large numbers of devices that have diverse capabilities, are long-lived, and are geographically distributed. These characteristics make fleet setup complex and error-prone. And since devices are often constrained in computational power, memory, and storage capabilities, this limits the use of encryption and other forms of security on the devices themselves. Also, devices often use software with known vulnerabilities. The combination of these factors makes IoT fleets an attractive target for hackers and makes it difficult to secure your device fleet on an ongoing basis.

AWS IoT Device Defender addresses these challenges by providing tools to identify security issues and deviations from best practices. AWS IoT Device Defender can audit device fleets to ensure they adhere to security best practices and detect abnormal behavior on devices.



Notice: This lab can likely be done with minimal cost if you just have the Defender turned on only when you need it. At time of writing, Audit is priced based on the number of devices connected to the IoT core, while Detect is priced based on the number of metric datapoints. AWS's free Tier for new users for the first month can reduce costs and it won't cost much if you just connect just a few devices. Just be careful to turn off the service when you are done.

To enable AWS IoT Defender, you can go to AWS IoT Core session and click on the “Defend” Icon. There are two functionalities of AWS IoT Defender:

1. **Audit**
   An AWS IoT Device Defender audit looks at account and device-related settings and policies to ensure security measures are in place. An audit can help you detect any drifts from security best practices or proper access policies, such as multiple devices using the same identity, or overly-permissive policies that allow one device to read and update data for many other devices. You can run audits as needed ("on-demand audits") or schedule them to be run periodically ("scheduled audits").
   An AWS IoT Device Defender audit runs a set of predefined checks (Audit checks) for common IoT security best practices and device vulnerabilities. Examples of pre-defined checks include policies that grant permission to read or update data on multiple devices, devices that share an identity (X.509 certificate), or certificates that are expiring or have been revoked but are still active.
2. **Detect**
   AWS IoT Device Defender Detect allows you to identify unusual behavior that may indicate a compromised device by monitoring the behavior of your devices. Using a combination of cloud-side metrics (from AWS IoT) and device-side metrics (from agents you install on your devices) you can detect changes in connection patterns, devices that communicate to unauthorized or unrecognized endpoints, and changes in inbound and outbound device traffic patterns. You create security profiles, which contain definitions of expected device behaviors, and assign them to a group of devices or to all the devices in your fleet. AWS IoT Device Defender Detect uses these security profiles to detect anomalies and send alerts via AWS CloudWatch metrics and AWS SNS notifications.

   AWS IoT Device Defender Detect is capable of detecting a variety of security issues frequently found in connected devices:

   Traffic from a device to a known malicious IP address or to an unauthorized endpoint that indicates a potential malicious command and control channel.
   Anomalous traffic, such as a spike in outbound traffic, that indicates a device is participating in a DDoS.
   Devices with remote management interfaces and ports that are remotely accessible.
   A spike in the rate of messages sent to your account— so that a rogue device does not end up costing you in per-message charges.

Extra credit: Configure Audit and Detect to report the abnormal behavior (attacks like DDOS, brute force, permission change, etc...).

Define Behaviors like:

```
{
  "name": "Listening TCP Ports",
  "metric": "aws:listening-tcp-ports",
  "criteria": {
    "comparisonOperator": "in-port-set",
    "value": {
      "ports": [ 8888 ]
    }
  }
}
Will be triggered and create an alert if the device connects to other ports that
are not 8888.
```

For more detailed instructions refer to the following link:
https://docs.aws.amazon.com/iot/latest/developerguide/iot-dg.pdf

Commands for Audit & Detect:
https://docs.aws.amazon.com/iot/latest/developerguide/device-defender-audit.html
https://docs.aws.amazon.com/iot/latest/developerguide/DetectCommands.html

Another useful resource is AWS online tech talks
Manage Security of Your IoT Devices with AWS IoT Device Defender - AWS Online Tech …

It provided a big picture of AWS IOT device defender and a demo at the end of the video.

5. Submission and Grading

Like the previous labs, for the first four tasks you should work as a team of at most **4** people. Each group can assign each student different tasks, and may get together to aggregate the system at the end. Submissions should include a demo video, showing emulated IoT devices can successfully send to / receive from IoT core and GreenGrass core, as well as the inference results and data visualization. Only one team member should upload a submission: a single .pdf document, where the first page has a link to your video, a link to your Gitlab repository, and the list of team members. A paper report isn't directly required, only a video.

1. **Build the Cloud**
   The students should leverage IoT components including a Hub, GreenGrass, IoT Core, IoT Analytics, IoT Device Defender to construct an infrastructure.
   a. If the students have set up, configured and emulated the device group and rules, and are able to demonstrate communication between GreenGrass devices and core they get 30 points.
   b. If the students are able to configure rules, create devices and emulate them but the GreenGrass communication flow appears to be broken they get 20 points.
   c. If the students answer the background questions and explain their approaches for configuring the devices and emulator and GreenGrass communication but neither of them are functioning properly they get 10 points.
   d. If the students do not report any configuration they get 0 points.

2. **Data Inference**
   The students must utilize Lambda to analyze emission levels of vehicles and return the results back to the device. Please describe your design approach for returning the results.
   a. If the students are able to correctly implement the message processing to analyze the CO2 level, create the Lambda function with appropriate rules, publish the results back to the subscriber device and simulate the entire process they get 30 points.
   b. If the Lambda function has been created and deployed and the pipeline to analyze and send/receive data is configured correctly but the simulation/results produced by service.py is incorrect they get 15 points.

   c. If nothing works but the students show what they tried and the issues they faced, they get 2-3 points.

3. **Use IoT Analytics to visualize the Data**
   Using AWS IoT Analytics, students must create a SQL database and then visualize the data using the Jupyter notebook provided with AWS Sagemaker. Students should have multiple (2-3 at minimum) high quality visualizations. You can use any Python plotting library to achieve this. Please put these figures in your report. Please also discuss what these visualizations tell you about your data.
   a. If the students successfully create the pipeline using IoT Analytics and Sagemaker, and have 2-3 high quality visualizations, they get 30 points.
   b. If the data pipeline is not functioning properly, or the student has low quality visualizations, they get 20 points.
   c. If the student does not report any visualizations, they get 0 points.

4. **AWS IoT Device Defender (Extra Credit)**
   This is an optional portion of the lab, and is extra credit. There are 3 main parts: configuring Audit, configuring Detect, and then using them to report abnormal behavior.
   a. If the students successfully configure Audit and Detect, and demonstrate them in action in their report and video, they get 10 points.
   b. If the student only demonstrates one of Audit or Detect, or their configuration only partially works, they get 5 points.
   c. If the student does not attempt this section, they get 0 points.

Submission/Points	**Points**
Build the Cloud	30
Data Inference	30
Iot Analytics	30
Video	30
Peer review	NA
Total	**120**

Extra Credit

Submission/Points	**Points**
IoT Defender (Audit and Detect)	10
Appendix 1	10
Appendix 2	10

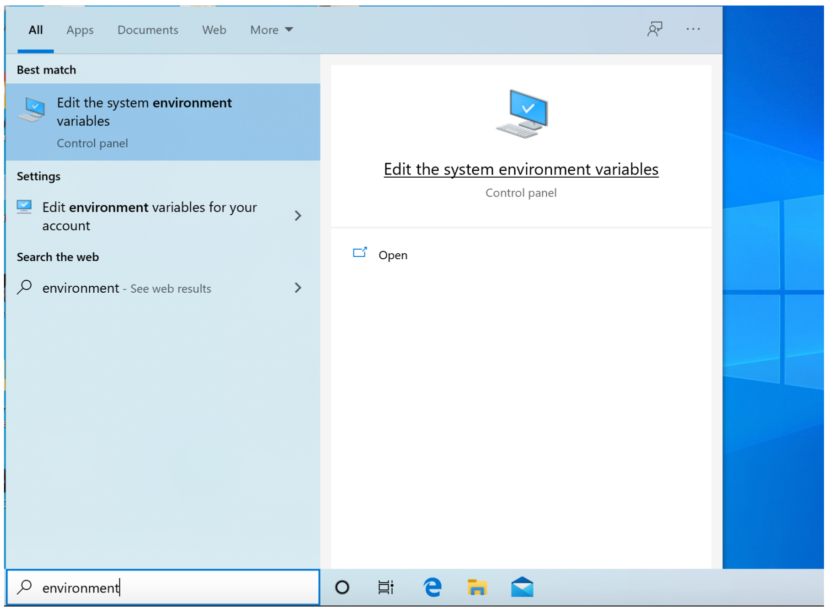
Appendix 1: **(Optional) Experiment with other networks**

Intro + Install and Configure Sumo

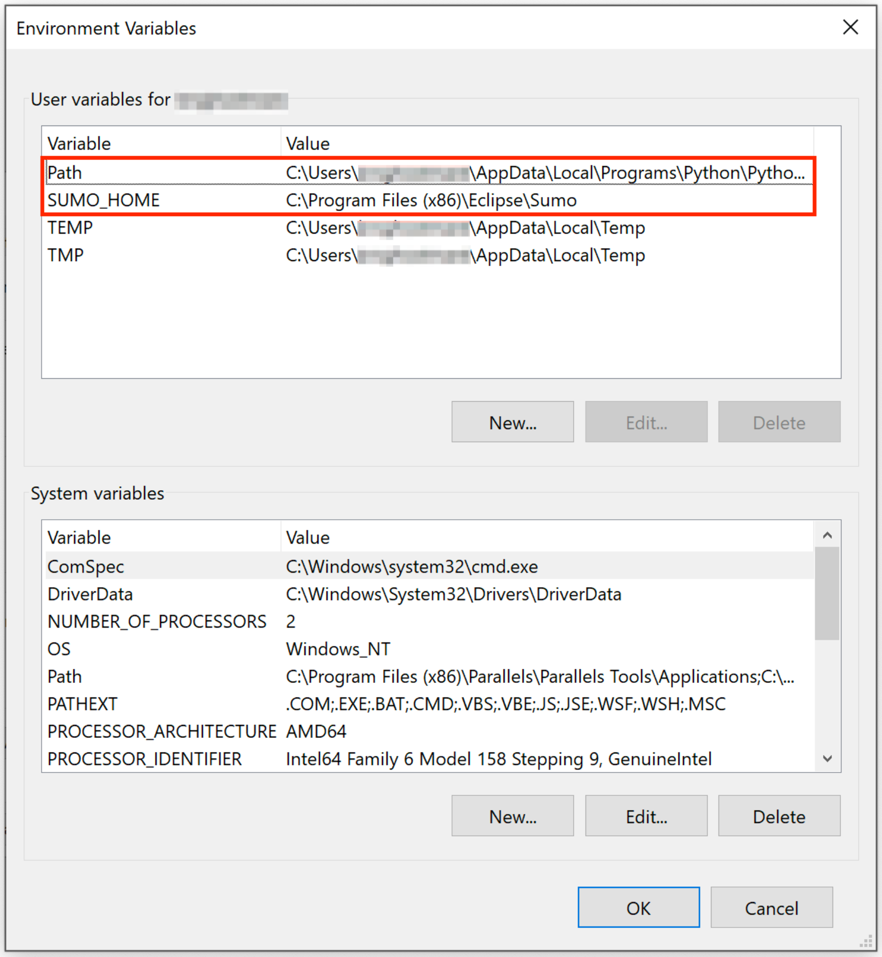
Sumo is an open source traffic simulation software used worldwide for many different projects. The software is developed by the *German Aerospace Center*, and is still constantly being updated. To download, go to https://sumo.dlr.de/docs/Downloads.php and you will see the options for macOS, Windows, or Linux. We suggest you use this software on a Windows or Linux machine because the sumo-gui failed constantly on macOS at the time we wrote this lab. Below we will provide the instructions for operating Sumo on Windows.

What you need to do:

1. Download and install the **64-bit installer** or **32-bit installer** (.msi).
2. Configure the environment path
    a. Go to "Edit the system environment variables" by typing **environment** in the search box.


All
Apps
Documents
Web
More
Best match
Edit the system environment variables
Control panel
Settings
Edit environment variables for your account
Search the web
environment - See web results
Edit the system environment variables
Control panel
Open
environment

    b. Under the User variables add ;C:\Program Files (x86)\Eclipse\Sumo\bin to the end of **Path** (Don't remove other paths). Also, set the **SUMO_HOME** value to C:\Program Files (x86)\Eclipse\Sumo
    *Note: change C:\Program Files (x86)\Eclipse\Sumo for **Path** and **SUMO_HOME** to the directory where your sumo is installed. Add a new variable **SUMO_HOME** if you're missing one.


Environment Variables

User variables for [redacted]

Variable	Value
Path	C:\Users\[redacted]\AppData\Local\Programs\Python\Pytho...
SUMO_HOME	C:\Program Files (x86)\Eclipse\Sumo
TEMP	C:\Users\[redacted]\AppData\Local\Temp
TMP	C:\Users\[redacted]\AppData\Local\Temp

System variables

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS	2
OS	Windows_NT
Path	C:\Program Files (x86)\Parallels\Parallels Tools\Applications;C:\...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE	AMD64
PROCESSOR_IDENTIFIER	Intel64 Family 6 Model 158 Stepping 9, GenuineIntel

comm

c. Now your Command Line is ready to run Sumo commands.

<u>Download and Convert Maps</u>

There are few methods to set up the map and have it run in Sumo. One easy way is to run the python script provided in the Sumo package. A map along with other simulation elements (cars, pedestrians, bikes, etc.) will be generated. Another way is to download .osm file from OpenStreetMap (https://www.openstreetmap.org), and then convert the map to .net.xml format used in Sumo. Note that the first method only provides a small-sized map, but generating the whole Urbana-Champaign area will not be an issue. On the other hand, the second method only provides the map without other simulation elements, but you can run a very large map using this method. Make sure you have **Python** on your computer.

What you need to do:

Method 1 (maps + simulation elements) -

1. Open the Sumo Gui you installed (This will be used in the last step)
2. Go to C:\Program Files (x86)\Eclipse\Sumo\tools on your command line.

3. Type `python osmWebWizard.py` to run the script, and a web browser should pop up.
4. Search for a location and zoom to desired map size.
5. Set the parameters. Under ⚙, set the options[11] for the map. “Add Polygons” adds buildings and other structures to your road map. “Duration” is the simulation time in seconds. Right-Hand traffic is assumed. Or, you can simply use the default setting.
6. Set the parameters for 🚘. Add any vehicle types you want in your simulation. You should adjust the **Through Traffic Factor** and **Count**. For a small map, the default values should be okay, but for a larger map like the entire Urbana-Champaign, start small with 1 or 2 for both values, or otherwise the generation will be too slow.
7. Then click generate.
8. After generation is complete (This could take some time based on the size of the map and parameters you specified, the files will be in C:\Users\<Your Username>\Sumo with name in `year-month-day-time` format
9. The generated simulation will automatically appear in the gui after generation is complete.

Method 2 (Map Only) -

1. Go to https://www.openstreetmap.org
2. Search for a location and zoom to desired map size.
3. Click on ‘export.’
4. (Optional) If you want a larger map, go to https://download.geofabrik.de/ and download maps with .bz2 extension.
5. Once you have your map, move on to next steps to convert them to Sumo format.
6. Open the command line in the directory you saved your map.
7. In the command line type:

   ```
   python "C:\Program Files (x86)\Eclipse\Sumo\tools\osmBuild.py"
   --osm-file filtered-test.osm --vehicle-classes passenger
   ```

   [redacted]osmBuild.py script is used to convert the map. You should replace `C:\Program Files (x86)\Eclipse\Sumo\tools` with the path which contains osmBuild.py on your computer. Also, “--” indicates options used[12].
8. You should now see a file with .net.xml in your directory

‘

***

Simulation

[11] Fore more details on using OSMWebWizard please visit: https://sumo.dlr.de/docs/Tutorials/OSMWebWizard.html

[12] For a more detailed list of option refer to https://sumo.dlr.de/docs/Networks/Import/OpenStreetMap.html#import_scripts

To simulate traffic, you would need cars and other elements. In method 1, all the necessary elements can be generated in one process. However, in method 2, you would need to generate routes for vehicles and pedestrians. You can write the xml file line by line but since we want millions of cars, we would show you an easier way to do it using python scripts.

What you need to do:

Method 1 - (If you followed the steps above and already have the gui open it should already be there)

1. Refer to Download and Convert Maps
2. Double click on osm.sumocfg in the directory, sumo-gui should open. Click the run button to run the simulation.

Method 2 -

1. Open command line in your project directory with .net.xml (sumo network file).
2. Type:
   ```
   python <SUMO_DIRECTORY>\tools\randomTrips.py -n
   sumo_network.net.xml -b start_time -e end_time -p ((end_time -
   start_time) / total_cars) --vehicle-class passenger --validate
   -o output_name.trips.xml
   ```

   Note that:
   -n: input network, change the name sumo_network
   -e: route time in seconds
   -b: binomial distribution
   -p: period (values after this option should be in floating number)
   total_cars: replace with the total amount of cars
   --validate: to make sure routes are valid

3. A file with .trips.xml will be generated
4. If you want to generate other traffic elements like buses, bikes, etc. you would need to hard code them because randomTrips.py is only for cars.
5. Write a configuration file with extension .sumocfg. You can download the osm.sumocfg in the file link we provide you at this link. Change the values of `net-file` and `route-files` in the configuration file to match your file name.
6. Run the simulation by typing `sumo -c <file-name>.sumocfg` and wait until it is finished. The output files specified under `<output></output>` in the configuration file will be generated.

Appendix 2: **(Optional) Predict the traffic hotspot**

After getting the simulation running, you can train a model to predict the traffic hotspot on the map which can be used for route planning. We will provide a parser in google drive to generate the training data for traffic hotspot. Download it and put it into the sumo simulation directory (C:\Users\<Your Username>\Sumo) which has `year-month-day-time` format.

However, In order to use the parser, we will need to acquire the car locations through the simulation, road network and traffic jam information from sumo. Go to your sumocfg directory (aka the `year-month-day-time` directory) and run sumo simulation with following command in terminal:

```
sumo -c <file-name>.sumocfg --collision.action warn
--time-to-teleport -1 --fcd-output fcd.xml --queue-output
queue.xml
```

This command will disable the teleportation of cars when traffic jams occur, and after running it you should have following file in current directory:

1. FCD (floating car data) file, which contains the location and speed of each car at each timestep.
2. Queue file, which contains the total length and time of car waiting on each road at each timestep. This would be used to determine traffic hotspot.
3. Sumo network file (.net.xml), which should be in this directory before simulation running. It contains the all road network information. If you used `osmWebWizard.py` to generate the simulation, it will have the name of "**osm.net.xml**"

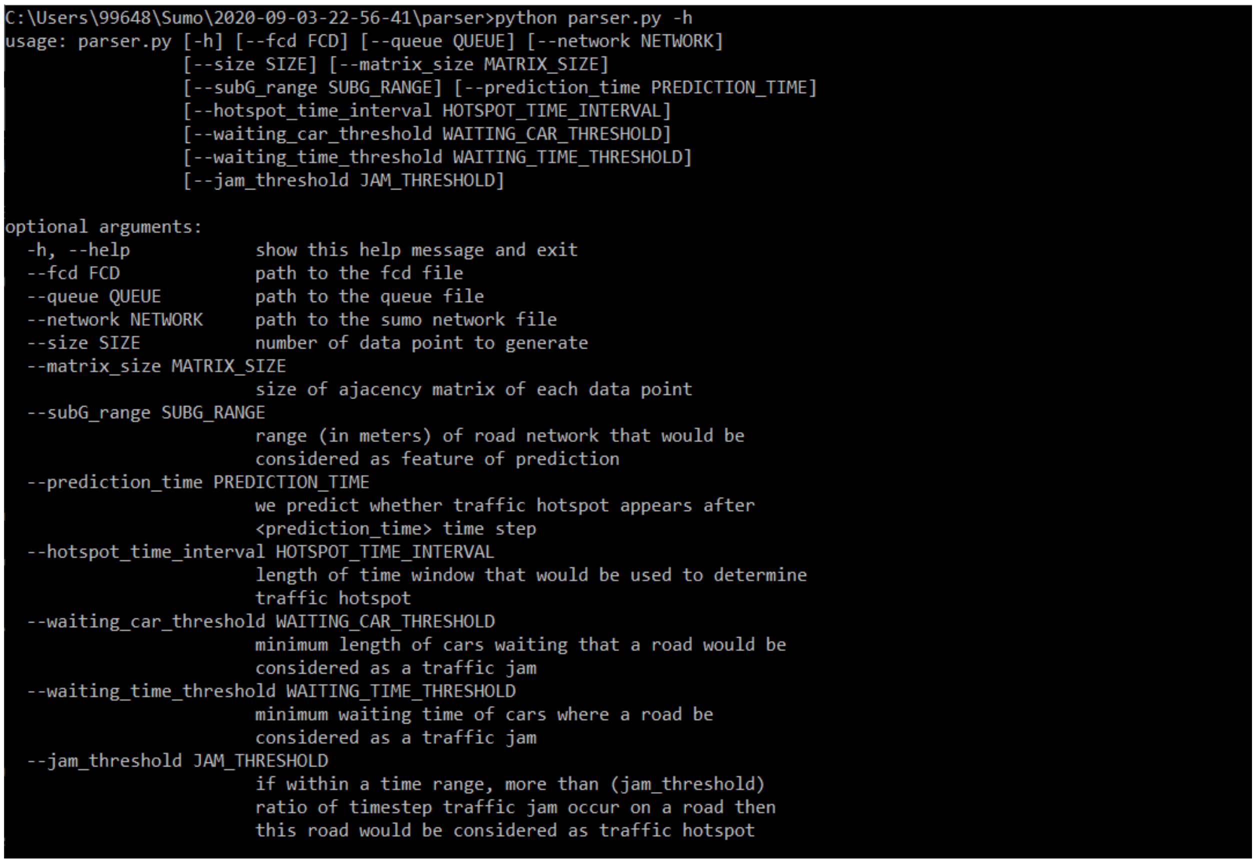
Alternatively, you can also get sample FCD, queue, and sumo network file from google drive. After getting those files, you can run the parser using following command:

```
python parser.py --fcd <fcd-file-path> --queue
<queue-file-path>   --network <sumo-network-path>
```

You can also use your own definition of hotspot by providing parameters to the parser, explore your options with the command:

```
python parser.py -h
```

And it will show you all the arguments of parsers. All arguments except path to fcd, queue and network file are optional. Program will use preset default values for them if you don't provide custom parameters. And all optional arguments are used to customize the standard for traffic hotspot:



```
C:\Users\99648\Sumo\2020-09-03-22-56-41\parser>python parser.py -h
usage: parser.py [-h] [--fcd FCD] [--queue QUEUE] [--network NETWORK]
                 [--size SIZE] [--matrix_size MATRIX_SIZE]
                 [--subG_range SUBG_RANGE] [--prediction_time PREDICTION_TIME]
                 [--hotspot_time_interval HOTSPOT_TIME_INTERVAL]
                 [--waiting_car_threshold WAITING_CAR_THRESHOLD]
                 [--waiting_time_threshold WAITING_TIME_THRESHOLD]
                 [--jam_threshold JAM_THRESHOLD]

optional arguments:
  -h, --help            show this help message and exit
  --fcd FCD             path to the fcd file
  --queue QUEUE         path to the queue file
  --network NETWORK     path to the sumo network file
  --size SIZE           number of data point to generate
  --matrix_size MATRIX_SIZE
                        size of ajacency matrix of each data point
  --subG_range SUBG_RANGE
                        range (in meters) of road network that would be
                        considered as feature of prediction
  --prediction_time PREDICTION_TIME
                        we predict whether traffic hotspot appears after
                        <prediction_time> time step
  --hotspot_time_interval HOTSPOT_TIME_INTERVAL
                        length of time window that would be used to determine
                        traffic hotspot
  --waiting_car_threshold WAITING_CAR_THRESHOLD
                        minimum length of cars waiting that a road would be
                        considered as a traffic jam
  --waiting_time_threshold WAITING_TIME_THRESHOLD
                        minimum waiting time of cars where a road be
                        considered as a traffic jam
  --jam_threshold JAM_THRESHOLD
                        if within a time range, more than (jam_threshold)
                        ratio of timestep traffic jam occur on a road then
                        this road would be considered as traffic hotspot
```

Parser should generate two files with the name “**data.txt**” and “**label.txt**” which contain features and labels respectively. And you can also find sample “data.txt” and “label.txt” files in the google drive. Then you can use Jupyter notebook file **“hotspot_training.ipynb”** to train a model based on “**data.txt**” and “**label.txt** to predict if traffic hotspot will appear around certain location”

In the “hotspot_training.ipynb” file, it will (1) merge data from “feature.txt” and “label.txt” to a tensorflow dataset data structure; (2) tensorflow dataset to training set, validation set and test set which are used to train the model, determine the accuracy of model and test the model respectively; (3) use the trainset to train the model to predict if certain location would be a traffic hotspot. The next section is a detailed description of what the parser does, if you wish to customize the standard for hotspot, we should read it.

Parser Code Description:
In general, what parser.py does is sampling road segments, gathering the traffic information nearby as feature and determine if nearby area of selected road will contain traffic hotspot in certain time steps, below is specific explanation of it :

1. Build the graph of road network using network file;
    - In the graph, each node is a road junction and each edge is a road.
2. Random sample road and time step in the simulation;

3. Build a subgraph near the sampled road using graph traversal to capture the geometric information;
    - It uses Dijkstra's algorithm to build a subgraph which includes closest #<matrix_size> road junction to the chosen road whose trip distance from road is smaller than <subG_range>.
4. Build the feature file using subgraphs and fcd file;
    - In the features, we want to include (**a)** topological information of the area; (how roads connect, length of roads etc.) **(b)** number of cars on sampled roads in sampled timestep.
    - In order to save the topological information, the parser would use adjacency matrix as data structure of feature, And then we will incorporate information like length of roads and number of cars on each road by adding two data fields to the matrix entries that represent the roads (aka. edge of graph).
    - Note here, since fcd is a big file, we may not put the whole file into memory, which means, each time we read from it, we need to iterate over the whole file piece by piece, which requires O(N) time (where N is size of file). As a result, to handle the big data like this we can't process our data points one by one, which will cost O(ND) time, let D be the # data points. Instead, we need to modify the reading file code to process all the data points at one iteration which will only cost O(N+D) time.
5. create the label file using queue file;
    - Given the subgraph we have, we check the Queue file to see that if in the given time ranges have a traffic hotspot.
    - Interested time range: let the sample timestep to be tsp, then the interested range of time would be [tsp+<predication_time>, tsp+<prediction_time> + <hotspot_time_interval>].
    - At a certain timestep, if the length of cars waiting on the road is larger than <waiting_car_threshold>, and the cars have more than <waiting_time_threshold> time steps, then the road is traffic hotspot at the given timestep,
    - For a range of timestep, if the ratio of timestep that contains traffic hotspot is larger than <jam_threshold> then the road of given range of timestep would be considered as a traffic hotspot.