Digital Twin development and cloud deployment for a Hybrid Electric Vehicle

8

8.1 Introduction

This chapter guides the user through the process of deploying a Hybrid Electric Vehicle (HEV) model for a Passenger Car, developed using MATLAB®, Simulink®, and Simscape™ into a Raspberry Pi hardware board in real time. Further a Digital Twin of the same HEV model is developed and deployed into the Amazon Cloud Services (AWS).The Raspberry Pi board will send the HEV system inputs, outputs, and states to the Amazon Cloud Services, and the Digital Twin model is ran concurrently on the cloud to perform off-board diagnostics (Off-BD) for detecting failures introduced in the model, which is running in Raspberry Pi. Fig. 8.1 shows the Off-BD steps that are going to be covered in this chapter. First, a couple of failure modes and failure conditions that will be detected by Off-BD is identified, then the block diagram of the Hybrid Electric Vehicle system is shown, after that the MATLAB®-, Simulink®-, and Simscape™-based model of the HEV is deployed on to the Raspberry Pi hardware, which works as the real physical asset in this case. With the inbuilt Wi-Fi capability of the Raspberry Pi, it can be configured as the Edge device to communicate to the cloud to transfer physical asset states from the model, which is running in the Raspberry Pi, to the cloud. A Digital Twin model of the same HEV system is compiled, and a Root Mean Square Error (RMSE) comparison−based diagnostic algorithm is also developed and deployed to AWS. So as the Simscape™ model of the HEV runs in the Raspberry Pi, a Digital Twin of the HEV is also running on the AWS with the same inputs collected from the Raspberry Pi, and real time diagnostic failure detection capabilities are demonstrated with Digital Twins.

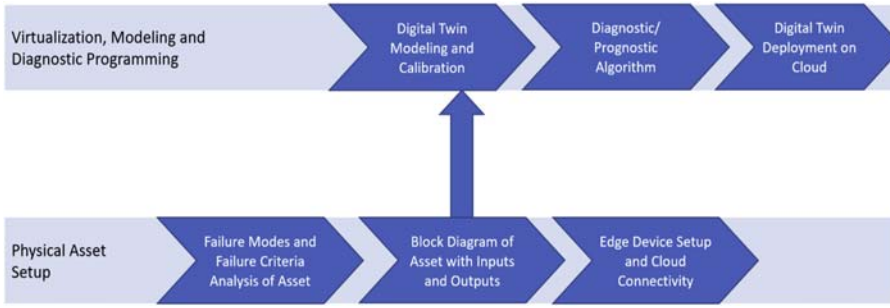


Figure 8.1 Off-BD steps covered in this chapter.

Digital Twin Development and Deployment on the Cloud. https://doi.org/10.1016/B978-0-12-821631-6.00008-6

All the codes used in the chapter can be downloaded for free from *MATLAB® File Exchange*. Follow the below link and search for the ISBN or title of the book:

https://www.mathworks.com/matlabcentral/fileexchange/

Alternatively, the reader can also download the material and other resources from the dedicated website or contact the authors for further help:

https://www.practicalmpc.com/

8.2 Hybrid Electric Vehicle physical asset/hardware setup

A Raspberry Pi computer board is used to run the MATLAB®, Simulink®, and Simscape™ model of the Hybrid Electric Vehicle in real time in a Virtual Hardware in Loop kind of mode. Authors call it Virtual Hardware in Loop because the HEV model is running on a Raspberry Pi real-time hardware (though it is not the actual physical hardware/asset like a vehicle's on-board computer because of the safety limitations to instrument the setup and collect data on a real vehicle, but it is definitely possible if we had the time and resources).We will be running the Simulink model of the HEV from a host computer in External Mode, with the target hardware selected as Raspberry Pi. The host computer and Raspberry Pi should be connected to the same Wi-Fi network as shown in Fig. 8.2.

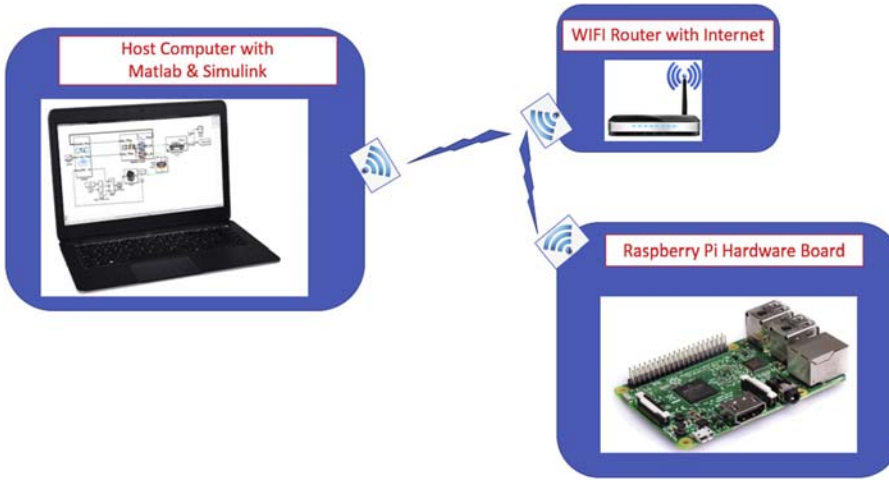


Figure 8.2 Physical asset/hardware setup running HEV model in external mode in real time.

Considering that the readers already have the host computer with MATLAB®, Simulink®, and Simscape™ already setup, and also there is a Wi-Fi router configured and available to connect, this section will focus mainly on configuring and installing

necessary software for Raspberry Pi hardware needed for this chapter. The Raspberry Pi used by the Authors is Raspberry Pi 3 B+, which can be bought from various sources. The steps in this chapter can be applied on a different version of Raspberry Pi board as well as long as it supports Wi-Fi connection. One of the source links available at the time of writing is given in Ref. [1]. After getting a Raspberry Pi board follow the latest instructions from Ref. [2] to install and setup the operating system for the Raspberry Pi hardware, Authors really recommend setting up Raspbian OS, which is tested and verified in this chapter.

We will be using Python for communicating to the HEV model, which will be running on the Raspberry Pi for collecting the real-time data and also communicating to the AWS cloud. We will be first checking if Python is installed on the Raspberry Pi with the OS installation. By default, Python 2.7 should be installed with the OS. From the host computer, a remote connection can be established to the Raspberry Pi using the Putty Desktop App, which can be installed from the link [3]. After installing Putty, open up the Putty Desktop App and enter the IP address of the Raspberry Pi as shown in Fig. 8.3. The IP address of the Raspberry Pi can be obtained using the ***ifconfig*** command by directly connecting the Raspberry Pi hardware to a keyboard, and a monitor. A new window as shown in Fig. 8.4 will be popped up to enter the login credentials of the Raspberry Pi. Enter the login details used at the time of installing the OS of the Raspberry Pi.

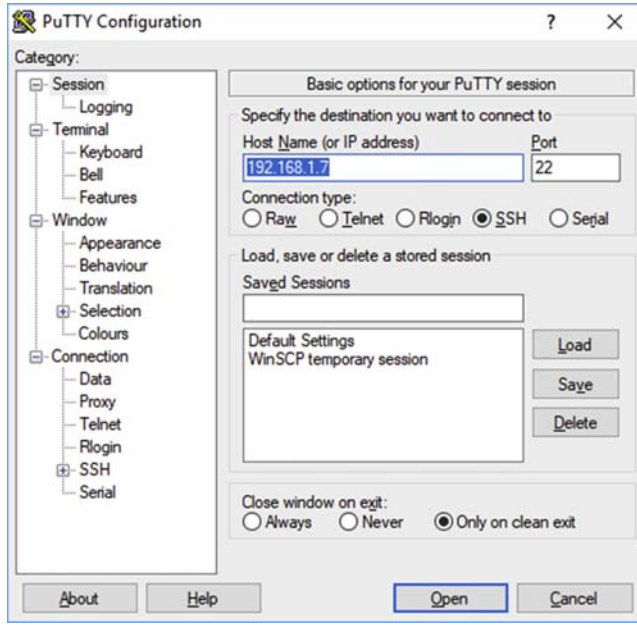


Figure 8.3 Connecting to Raspberry Pi remotely using Putty.

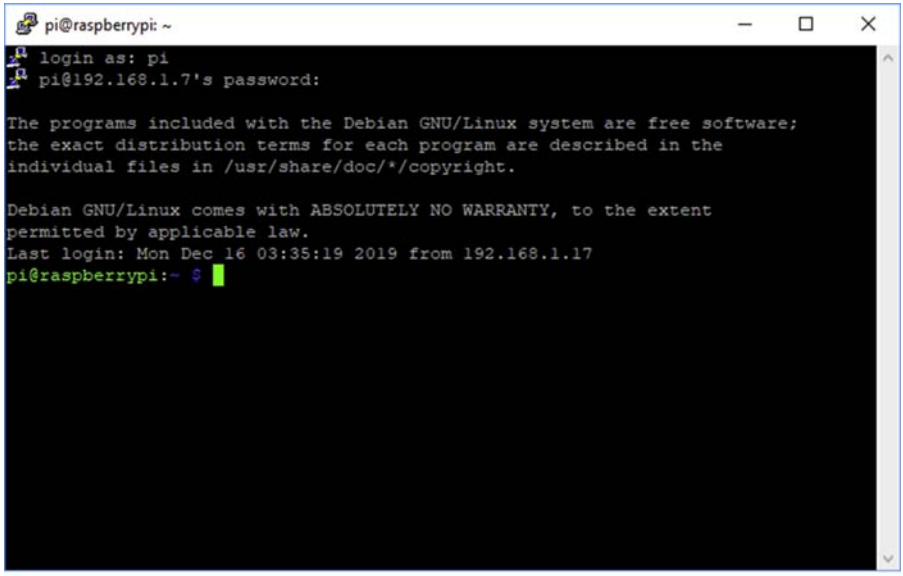


Figure 8.4 Login to Raspberry Pi remotely using credentials.

After remotely connecting to the Raspberry Pi, on the Putty command line, enter the command ***python - -version*** and ***pip - -version*** to check whether the Python software and Python package installer are installed. If installed, it should show up the versions as shown in Fig. 8.5.

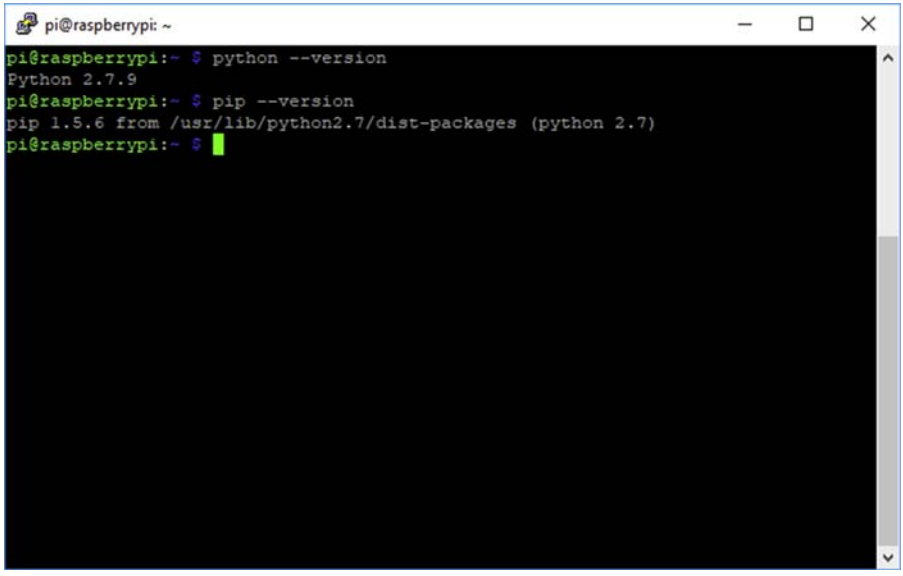


Figure 8.5 Checking Python and PIP versions.

Next we need to install the Mosquitto MQTT communication protocol broker and clients to the Raspberry Pi. MQTT is a lightweight messaging protocol designed for low-cost and low-power embedded systems. This protocol and broker are primarily responsible for receiving all messages, filtering them, deciding who is interested in it, and publishing the messages to all subscribed clients. In our example, the Simscape™ model running on the Raspberry Pi will be sending MQTT messages to the MQTT broker with the real-time inputs, states, and outputs of the HEV system. The broker will then route the message to a Python program, which has subscribed to the broker. The Python program will then buffer the messages and reformat the data and send to the AWS cloud. For installing the Mosquitto MQTT broker, enter the command ***sudo apt-get install mosquitto*** on the Putty command line as shown Fig. 8.6. After that install the Mosquitto client by entering the command ***sudo apt-get install mosquito-clients***, see Fig. 8.7. We will also need to install a Python MQTT library called Paho-MQTT. This library allows Python programs to connect to the MQTT broker and receive and publish messages. Install the Paho-MQTT using the command ***sudo pip install paho-mqtt*** as shown in Fig. 8.8. Paho-MQTT is a specific Python library, which is why in this command we used the pip.

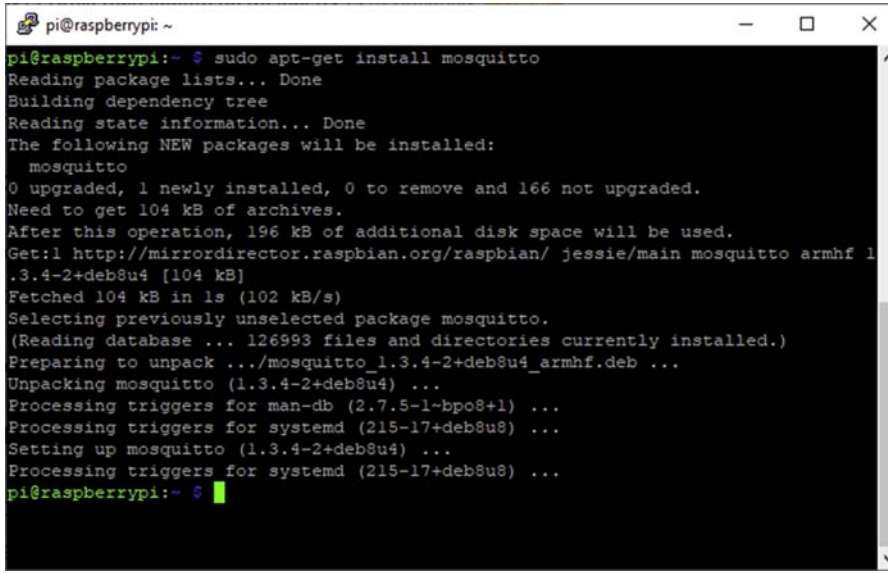


Figure 8.6 Installing Mosquitto MQTT broker.

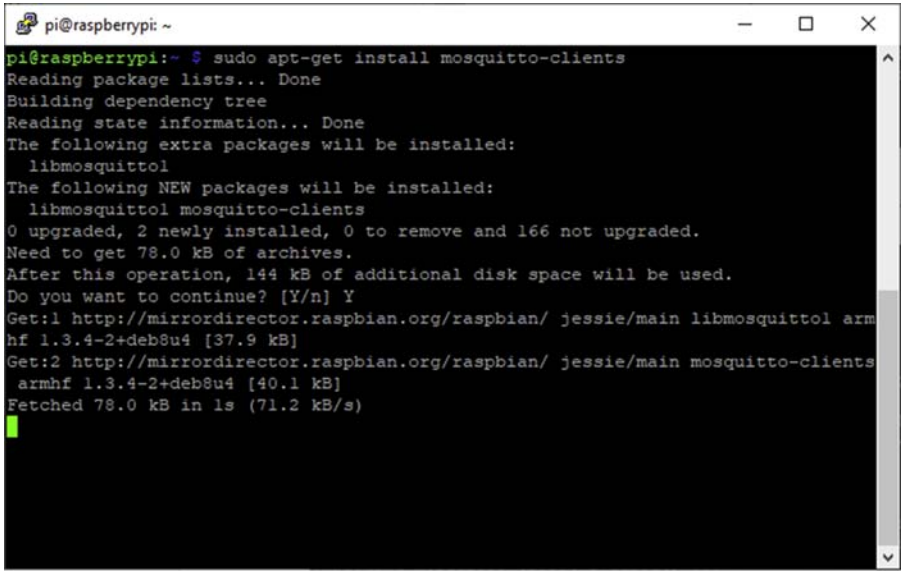


Figure 8.7 Installing Mosquitto MQTT client.

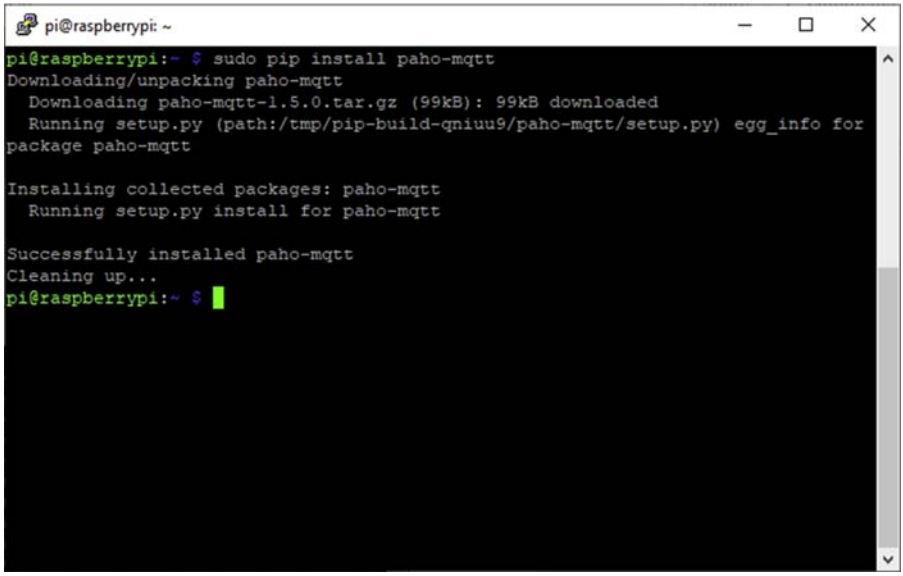


Figure 8.8 Installing Pah0-MQTT client for Python.

8.3 Block diagram of the Hybrid Electric Vehicle system

Fig. 8.9 shows the block diagram of the inputs/outputs and states of the Hybrid Electric Vehicle system. The input to the system is the target speed of the vehicle, and the vehicle controller then acts on the target speed and commands the Electric Motor, Generator, and the Engine in the HEV system accordingly to meet the target vehicle speed. So the output of the system is the actual vehicle speed. Also we are outputting the states of the system such as Motor Speed, Generator Speed, Engine Speed, and Battery SoC for Off-BD monitoring.

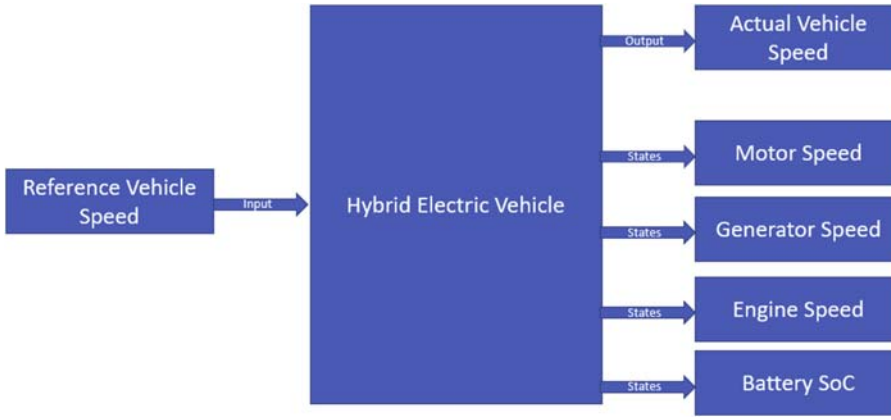


Figure 8.9 Block diagram of the Hybrid Electric Vehicle system.

8.4 Failure modes and diagnostic concept of Hybrid Electric Vehicle system

Fig. 8.10 shows the block diagram of the Off-BD Digital Twin diagnostics process followed in this chapter. The input Vehicle Speed Reference is fed to the HEV model which is running on the Raspberry Pi and also sent to the cloud along with the actual states and outputs of the system from the hardware. In the AWS cloud, a Digital Twin model of the same HEV system will be running with the same input and the states and outputs of the Digital Twin model are compared with the actual data collected from the hardware. A RMSE-based diagnostic detection and decision-making is developed to compare the actual and digital twin outputs and states. In this chapter, an example to detect the Throttle failure of the Internal Combustion Engine in the HEV system is demonstrated. A similar approach can be followed to detect failures of the other components of the system as well. A throttle failure is introduced into the model

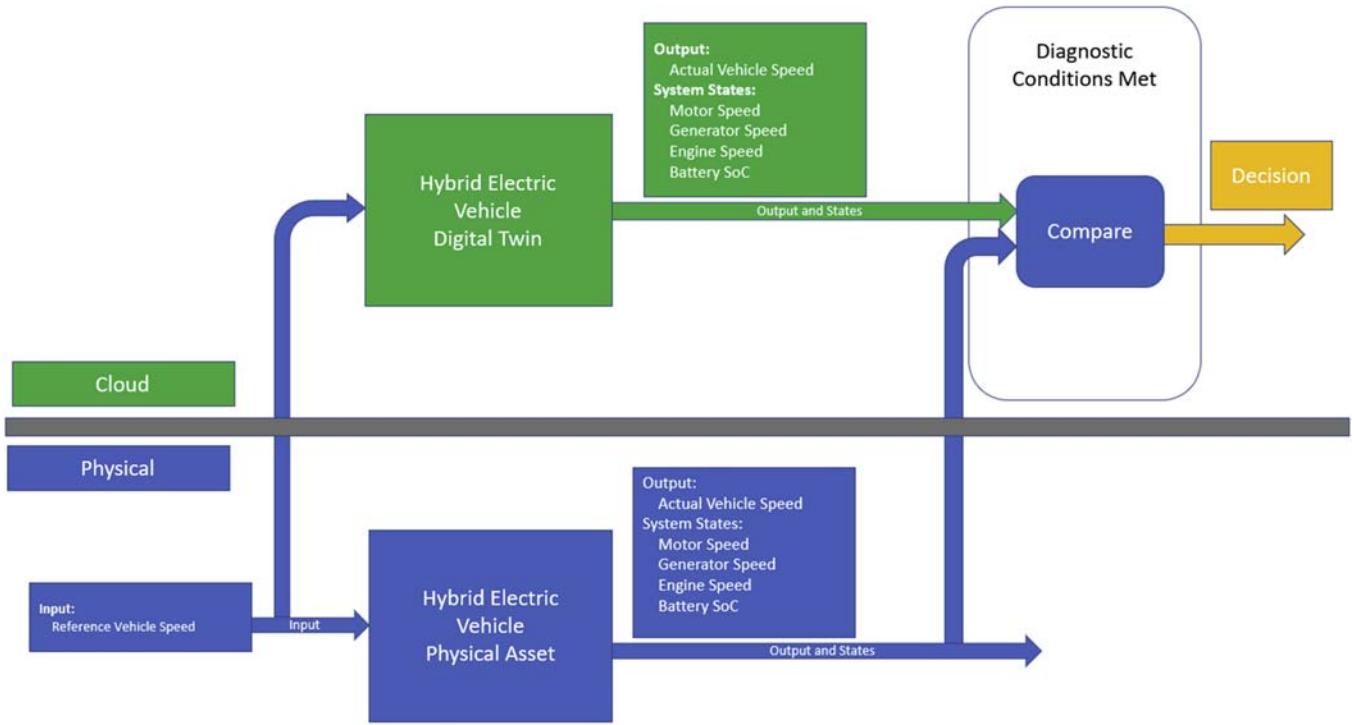


Figure 8.10 Off-BD diagnostics process for Hybrid Electric Vehicle system.

that is running in Raspberry Pi by overriding the Throttle which is commanded internally by the HEV controller for a given vehicle speed reference to zero. Because the Throttle is zero for the model which is running in Raspberry Pi, the HEV system will behave differently and its states and outputs will be different when compared to the states and outputs of the Digital Twin model for which the input is only the vehicle speed reference and no other failure conditions. This difference in the outputs and states between the Actual and Digital Twin data is identified, and if the RMSE is greater than a threshold, a failure condition is flagged and the User is notified directly from the cloud using a Text or E-mail notification.

8.5 Simscape™ model of a Hybrid Electric Vehicle system

Hybrid Electric Vehicle system is a multidisciplinary multi physicssystem including Mechanical systems such as Engine, Transmission and Powertrain, Chemical dynamics of Internal Combustion recipe of the Engine, Battery systems, and the Electrical systems including Generator, Motor, etc. Based on the Author's Automotive Industry experience, various levels of fidelity models are utilized in the industry depending on the specific application for which the models are being used for. It could be a simple map-based model for each component, a full physics-based model for all components, or a mix where some components are modeled using maps and some components with higher fidelity physics-based modeling. But the challenge always will be validating the model performance with data from the real physical system/asset and drawing a line for the usefulness of the models for the applications that are being considered. Considering the complexity of the HEV system, and all the various HEV system combinations and variations out there, the main focus in this chapter is not really to follow the step-by-step process of developing the model and validating it, instead Authors will focus more on using a validated HEV model that is readily available in MATLAB Central File Exchange portal and deploying the model to the Raspberry Pi hardware and also developing and deploying the Digital Twin on the same model to the AWS cloud and doing a real-time Off-BD.

The HEV model published in MATLAB Central link [4] is used in this chapter. This is a Series−Parallel Hybrid Electric Vehicle system model, where the vehicle drivetrain can be powered by the internal combustion engine working alone, the electric model working alone, or by both the combustion engine and motor system working together. There is a supervisory power split controller that tries to optimize the power split by trying to operate the engine at its optimal operating region. More details about the various hybrid electric configurations can be found in Ref. [5]. Fig. 8.11 shows a high-level block diagram and the flow of the power and connections in the series parallel HEV.

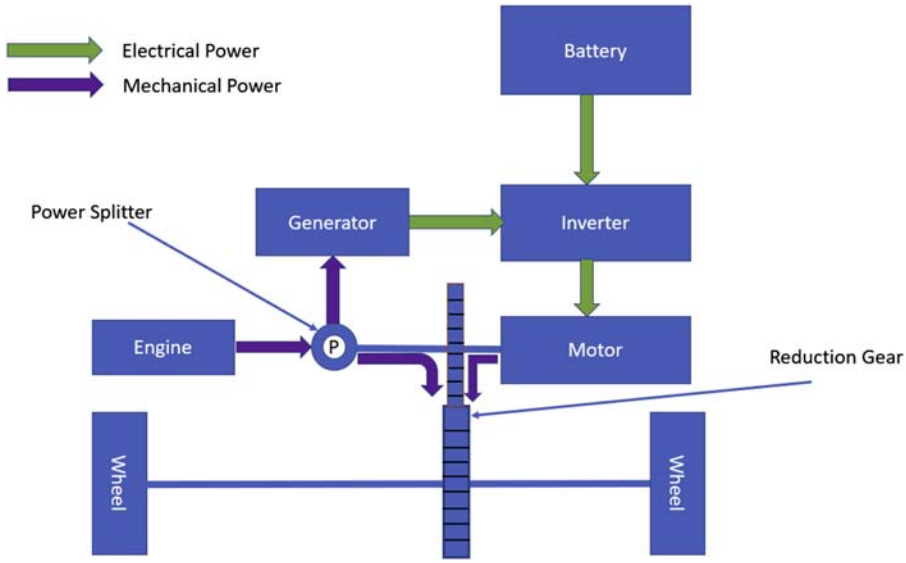


Figure 8.11 Series–Parallel Hybrid Electric Vehicle configuration [5].

The MATLAB Central HEV model by default is configured to run with a continuous time-step solver, but we have to run it with a fixed-step discrete time solver in order to run it on the Raspberry Pi hardware. So the Authors have made some necessary changes on the model downloaded from MATLAB Central to work with a discrete-time solver with a fixed step size of 0.1 s. Fig. 8.12 shows the top-level Simulink diagram of the HEV Simscape™ model. A description of various systems and components of the model is given further, as already mentioned, and the Authors did not really develop this model, so it is mainly explaining what was already in the MATLAB Central downloaded model and its purpose.

The HEV model mainly consists of the following components:

1. Vehicle Speed Controller Module

A time-based target vehicle speed is selected for the HEV system from a few set of vehicle speed profiles and fed using From Workspace blocks as shown in Fig. 8.13. Further, this target vehicle speed and the actual vehicle speed are fed to a High-Level Proportional Integral (PI) controller to derive the acceleration command or brake request to decelerate the vehicle as required as shown in Fig. 8.14. If the actual vehicle speed is less than the reference speed, the high-level controller will request more acceleration, and this acceleration request will be a low-level supervisory controller to produce the required torque either from motor, engine, or both to meet the required acceleration. But if the actual vehicle speed is higher than the reference speed, the controller will request deceleration through the brake pedal signal, which will be directly applied to the vehicle dynamics system, which will be converted to a brake torque to decelerate the vehicle.

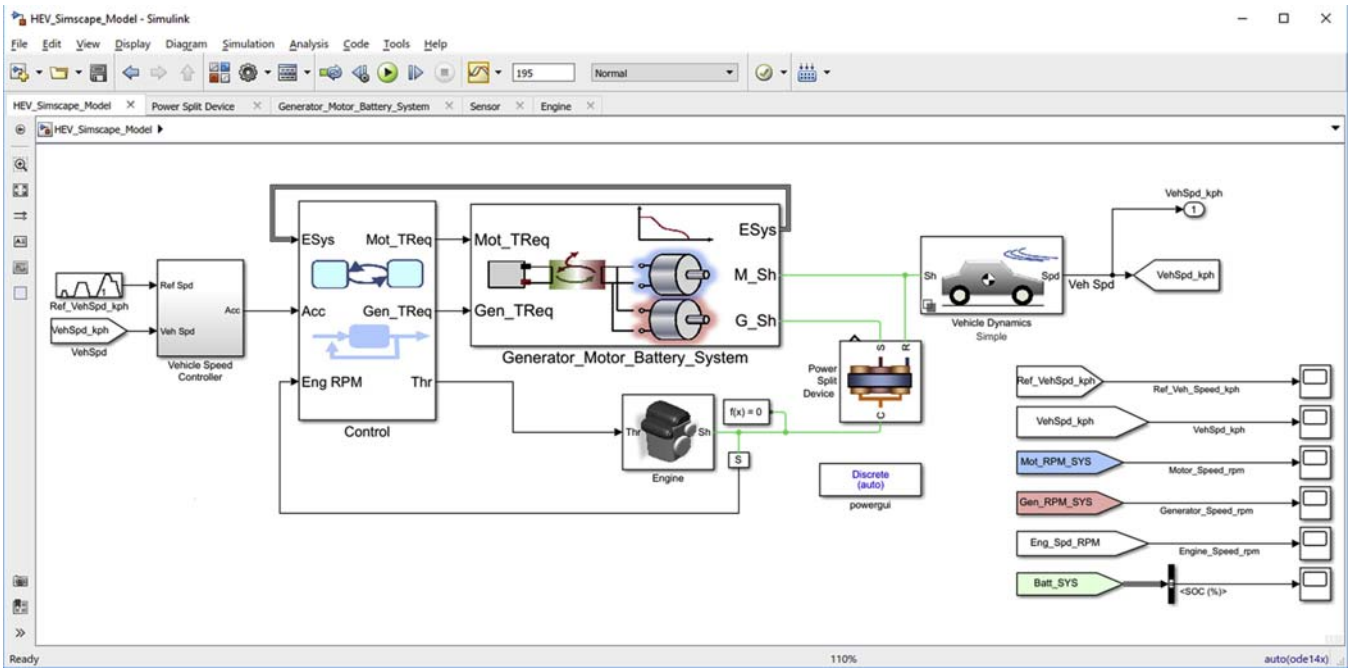


Figure 8.12 Top-Level HEV Simscape™ model.

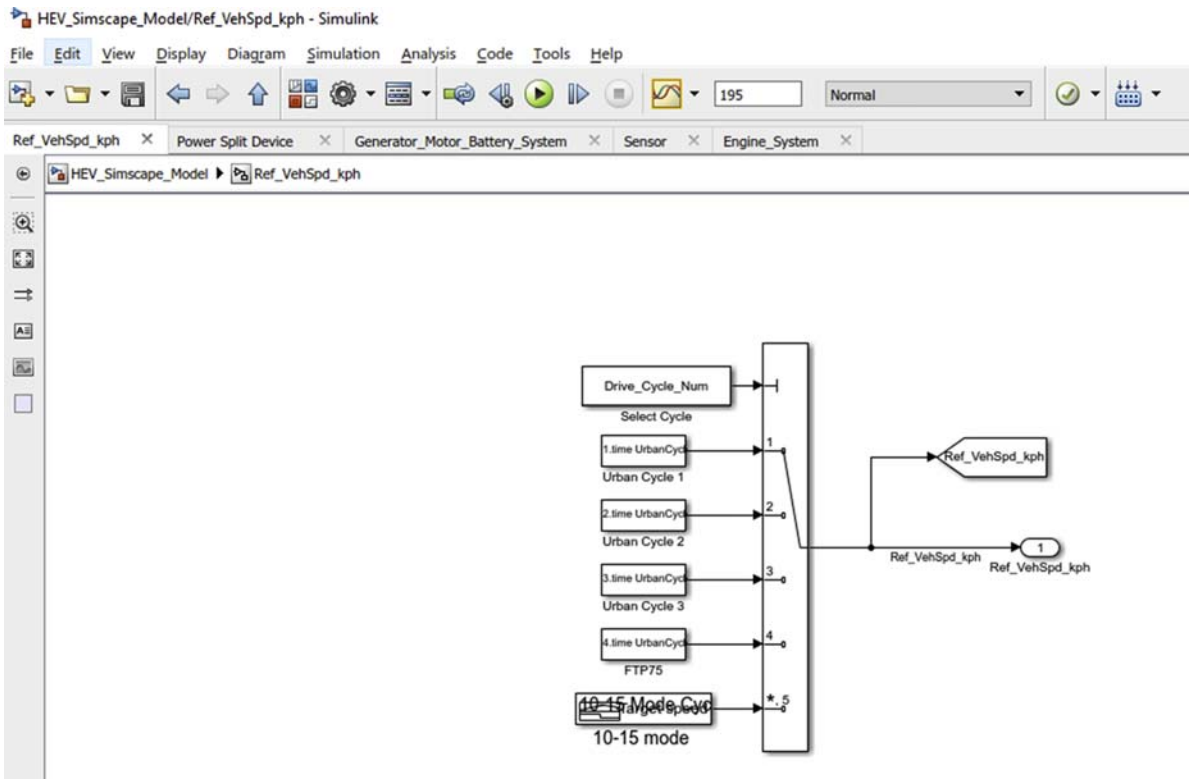


Figure 8.13 Target vehicle speed selection.

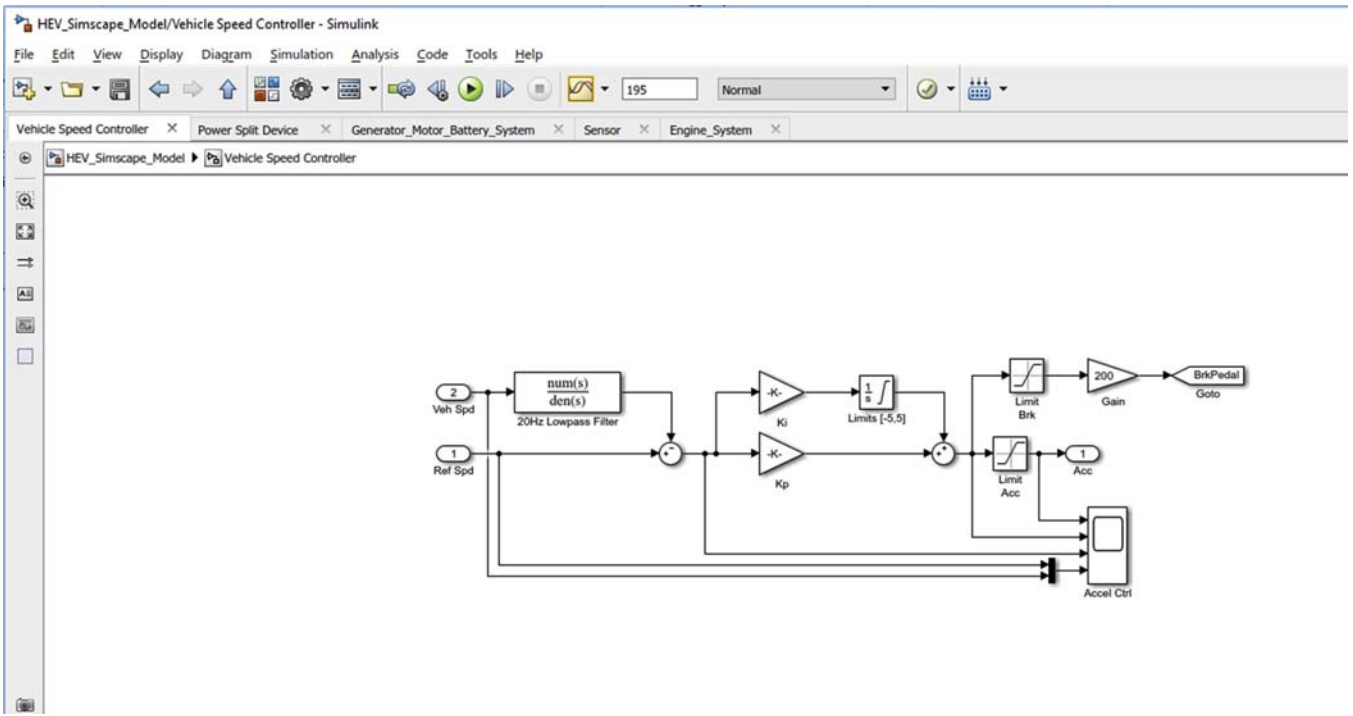


Figure 8.14 Proportional integral controller to control the acceleration and brake of HEV.

2. Supervisory Power Split Controller Module

This module or subsystem is responsible for converting the positive acceleration request of the high-level vehicle speed controller into corresponding Motor command, Generator command, and the Engine Throttle command requests to generate torques from these components to meet the required acceleration by operating these components at its optimal region. As shown in Figs. 8.15 and 8.16, this subsystem includes a mode selection logic that decides which of the devices from Motor/Generator/Engine to be operating and the required power from each based on the current operating conditions, state of the system, and acceleration demands. Further, based on selected mode, four lower-level controllers are used for controlling the torque demands from Motor, Generator, and Engine and also maintaining the charge levels of the battery.

3. Generator, Motor, DC−DC Converter, and Battery System

This subsystem includes the Motor, Generator, DC−DC Converter, and Battery system models or the so called plant models. This whole subsystem is modeled using various block sets from Simscape™ toolbox. The motor and generator subsystems take the torque requests from the respective controllers and respond to it as rotational motion to produce the required torque. The DC−DC converter will charge the battery to store the electrical energy.

4. Engine System

The engine system or the engine plant model takes the throttle command from the engine controller and convert that to a rotational motion to produce the demanded torque. See Fig. 8.17 for the engine system plant model implemented using Simscape™ blocks.

5. Power Split System

The power split device hooks the combustion engine, generator, and electric motor together and the vehicle driveline system together. This power split device is the essential part of the HEV system, and it makes the routing of power from the various power sources possible with its planetary gear system. Fig. 8.18 shows the planetary gear system logic in the HEV model implemented using Simscape™ blocks.

6. Vehicle Dynamics System

The vehicle dynamics subsystem implements the longitudinal vehicle dynamics equation given below. Here F_{wheel} is the tractive force which propels the vehicle, and F_{brake} is the braking force generated when pressing the brake pedal. Note that the BrkPedal signal is coming from the high-level vehicle speed controller subsystem, and it will be greater than zero if the vehicle speed controller decides that the vehicle needs to decelerate. $F_{drag}, F_{gravity}, F_{rolling}$ are, respectively, the forces due to aerodynamic drag, gravity, and rolling resistance, which are acting on the vehicle. The brake, drag, gravitational, and rolling resistance forces are opposing the tractive force, which

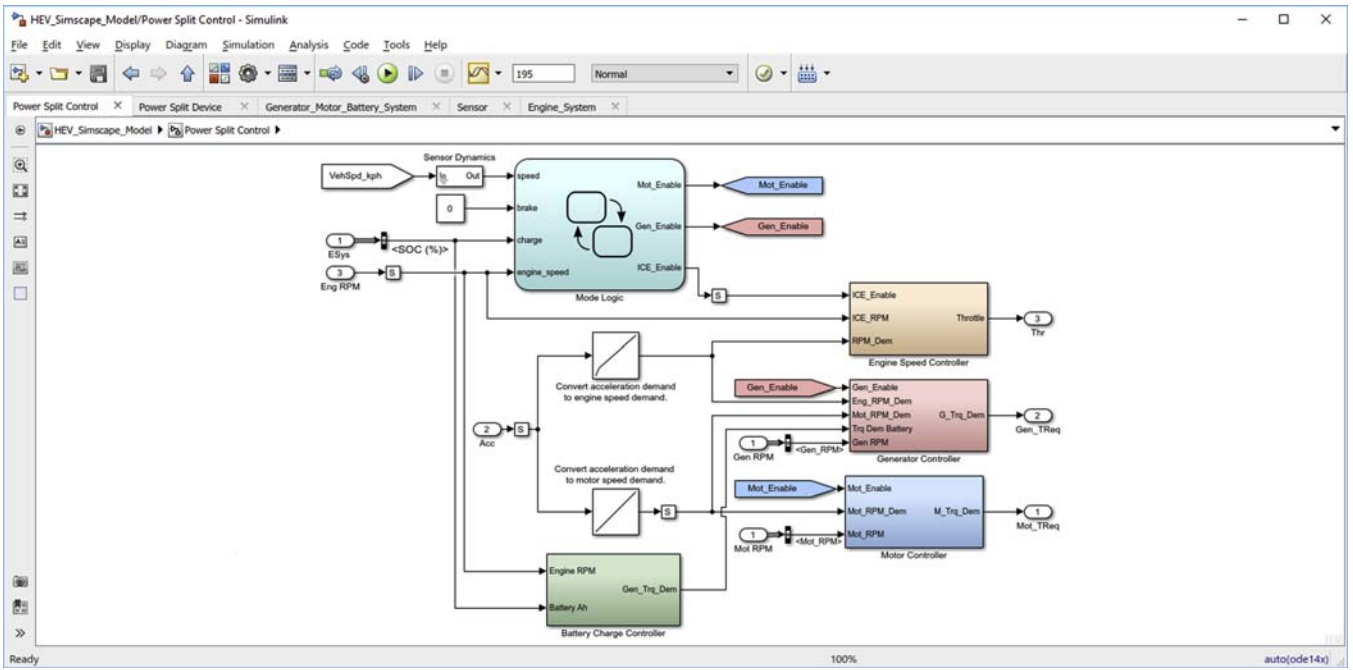


Figure 8.15 Supervisory Power Split Controller Module.

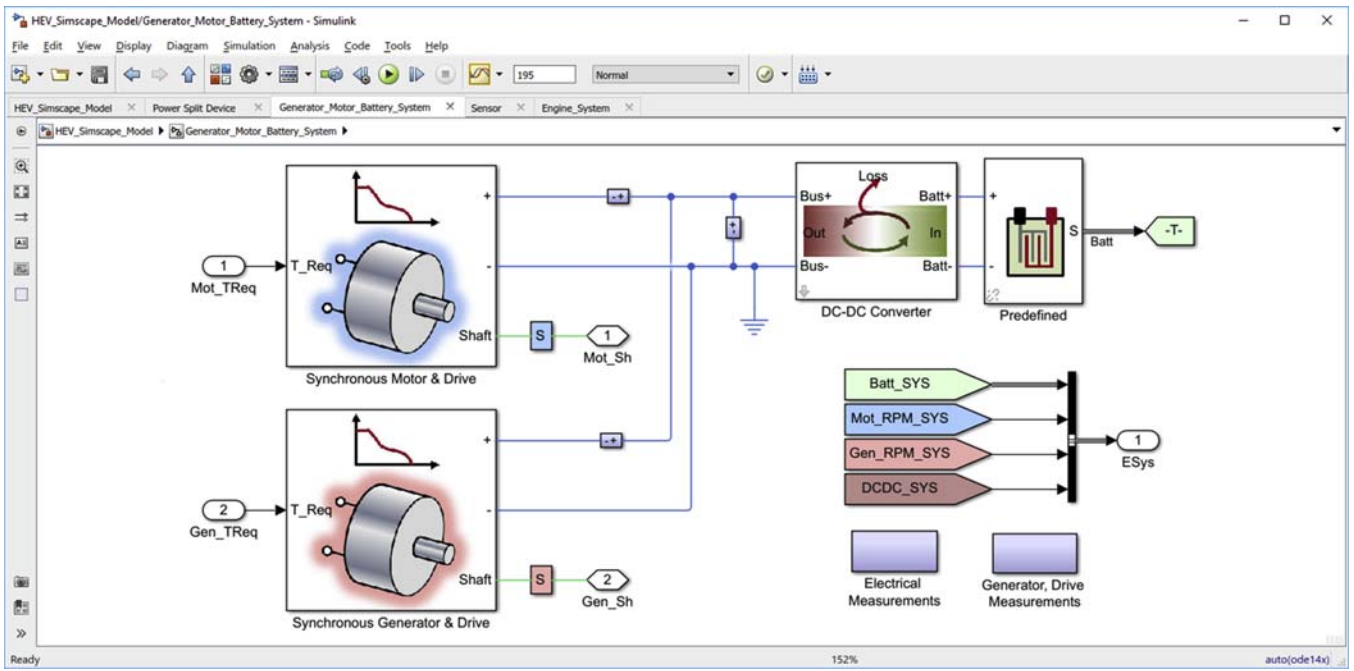


Figure 8.16 Generator, Motor, DC−DC Converter, and Battery system plant.

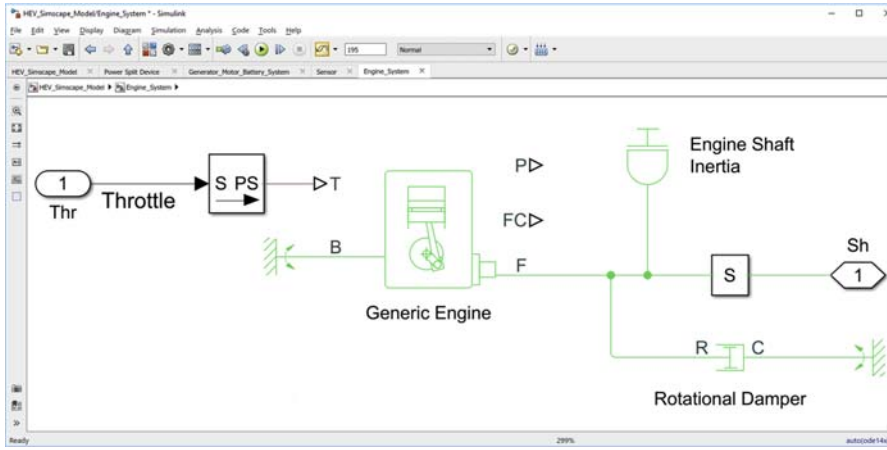


Figure 8.17 Engine system plant.

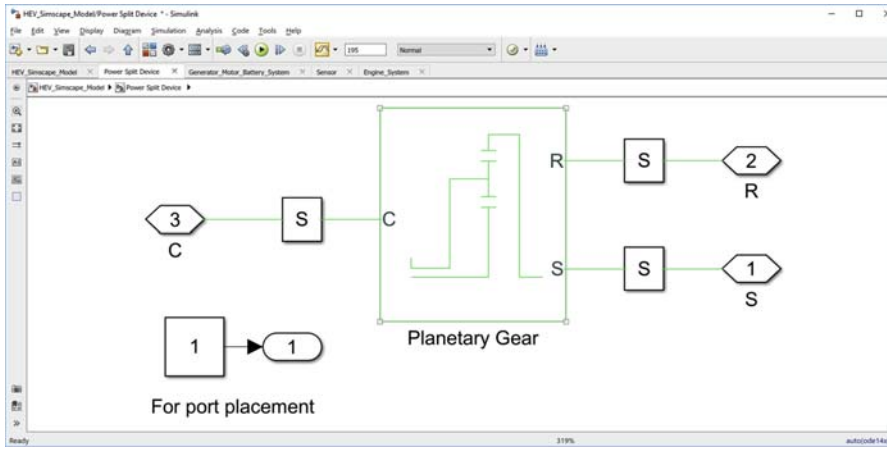


Figure 8.18 Power split planetary gear system plant.

is why all these forces are subtracted in the force balance equation. m_{eqv} is the equivalent mass of the vehicle, which is not a fixed mass, it varies based on the gear ratio mainly. $\dot{v}$ is the vehicle acceleration. Fig. 8.19 shows the vehicle dynamics subsystem implementation using Simscape™ and Simulink blocks.

$$m_{eqv} * \dot{v} = F_{wheel} - F_{brake} - F_{drag} - F_{gravity} - F_{rolling}$$

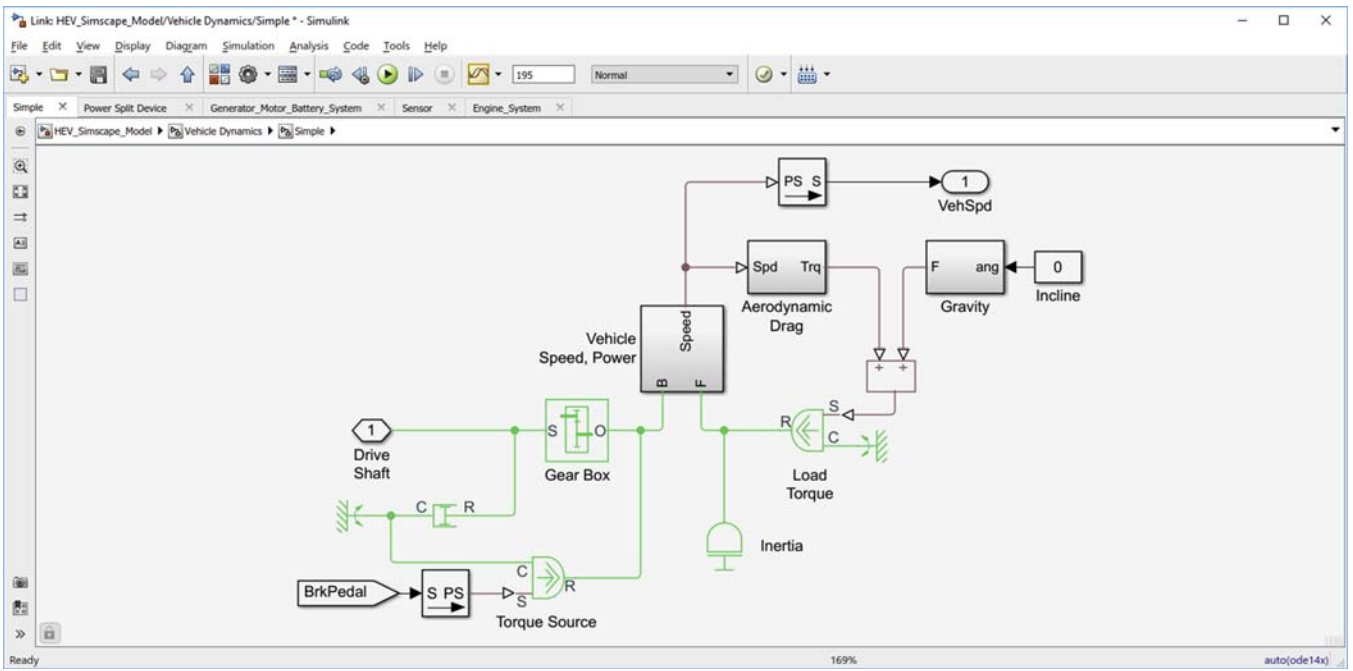


Figure 8.19 Vehicle dynamics system plant.

7. Data Monitoring and Logging

All the inputs, outputs, and states of interest of the system are logged and monitored using scope blocks from Simulink as shown in Fig. 8.20.

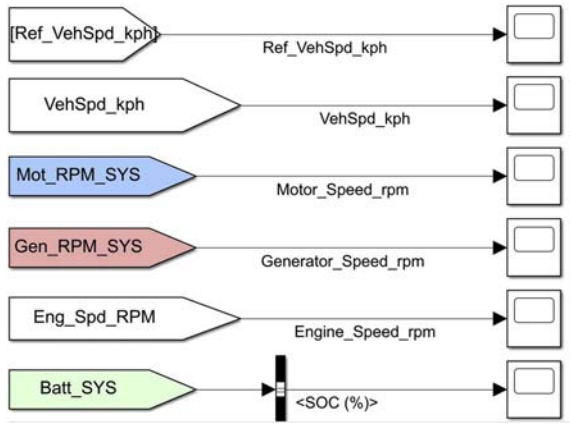


Figure 8.20 Data monitoring and logging.

Open the model HEV_Simscape_Model.slx and run the simulation from the attached folder under **Code_Files\Discrete_Time_HEV_Model** for the Chapter 8. When we open the model all the required paths will be added to the MATLAB path, and after the simulation is completed, it will plot all the key signals HEV system such as Reference and Actual Vehicle speed, Motor, Generator, and Engine Speeds and Battery SoC, etc, as shown in Fig. 8.21. It can be seen from the plot that the Engine, Motor, and Generator work together to meet the power required for vehicle speed reference. It has a pretty good vehicle speed tracking, and also during the Acceleration events, the Battery charge also depletes some, but during the steady-state and deceleration events, the controller tries to charge the battery back.

Next we will prepare to run the HEV Simscape™ on the Raspberry Pi hardware. In addition to the MATLAB® and its toolboxes already installed and used for the previous chapters, we need to install the hardware support package for Raspberry Pi for developing and deploying Simulink controls logic into the Raspberry Pi hardware. Mathworks® provides a hardware support package to develop, simulate and program algorithms, and configure and access sensors and actuators using Simulink blocks with the Simulink® Support Package for Raspberry Pi hardware. Using MATLAB® and Simulink® external mode interactive simulations, parameter tuning, signal monitoring, and logging can be performed, as the algorithms run real time on the hardware board.

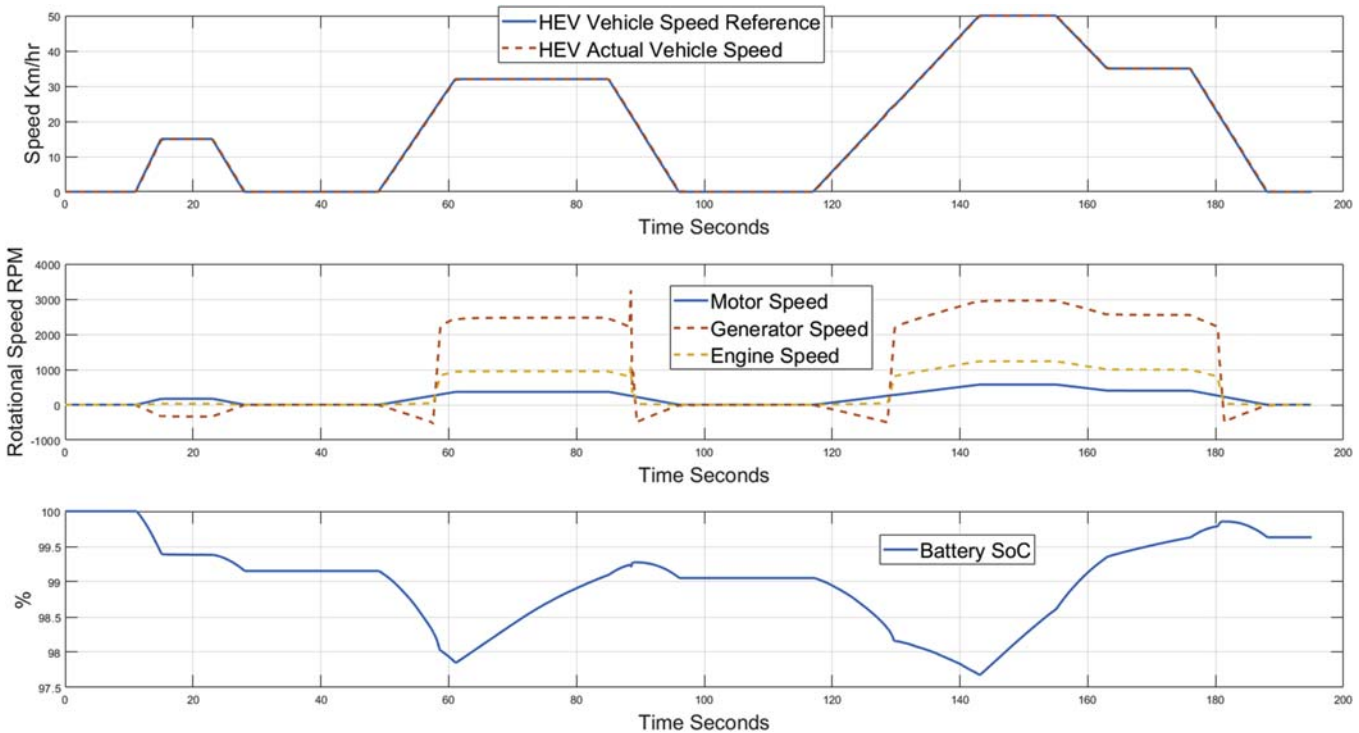


Figure 8.21 HEV vehicle simulation results for an urban cycle.

Follow the below steps to install the Simulink support package for Raspberry Pi hardware:

1. From the MATLAB® window go to Home >> Add-Ons >> Get Hardware Support Packages. Fig. 8.22 shows the Add-On Explorer GUI.
2. Click on the Simulink® Support Package for Raspberry Pi Hardware option, highlighted in Fig. 8.23, and it will guide into next window with an Install button as shown in Fig. 8.24. Click on it. For any support package installation, User needs to log on to a MathWorks account using the option shown in Fig. 8.25. Log on using the account, or create a new account if the User doesn't have it already.
3. Wait for the installation to be completed. An installation progress window is shown in Fig. 8.26. After the successful installation, it will show on the Add on Explorer that the Simulink Support package for Raspberry Pi hardware is installed as shown in Figs. 8.27 and 8.28.

Next we will make the necessary changes to the HEV model to run it on the Raspberry Pi hardware. The host computer running MATLAB and Simulink will be communicating to the Raspberry Pi hardware using the Wi-Fi protocol. We already know the IP address of Raspberry Pi when we connected to it using Putty desktop App, which will be used to connect and download the Simulink model to the Pi hardware. Open the model **HEV_Simscape_Model_Rasp_Pi.slx** from the attached folder under **Code_Files\ HEV_Model_for_Raspberry_Pi** for the Chapter 8.This is pretty much the same model we used earlier to run the desktop simulation from the host computer, but some setting changes are made to run it on the Raspberry Pi, such as the Hardware Implementation option has been set to Raspberry Pi board, simulation mode is selected to be External for running the interactive simulation on the hardware, etc. Also a simple LED blink logic is added on to the model in order to verify when this model runs on the PI hardware, and it will blink the on-board LED on the PI.

On this model based on the IP address of the Raspberry Pi hardware, user needs to update it by going to ***Simulation >> Model Configuration Parameters >> Hardware Implementation >> Target Hardware Resources*** as shown in Fig. 8.29. The login credentials also needs to be updated if they are different from the default one shown in Fig. 8.29.

After updating the IP address and login credentials, the model is ready to be deployed on to the PI hardware. Click on the simulation button from the model. It will generate code from the model, compile it for the Raspberry Pi target hardware, and generate the executable and will start running it on the PI hardware. This process can take a few minutes, but the progress can be monitored in the Simulink Diagnostic viewer as shown in Fig. 8.30. Once it is start running, we can see the green LED on the PI board will be blinking and also we can monitor the signals directly in Simulink like we did for desktop simulation. It can be noted that since this simulation is running on the hardware board, the simulation will be running in real time and it will take 195 real seconds to finish the 195 s in simulation compared to the desktop simulation.

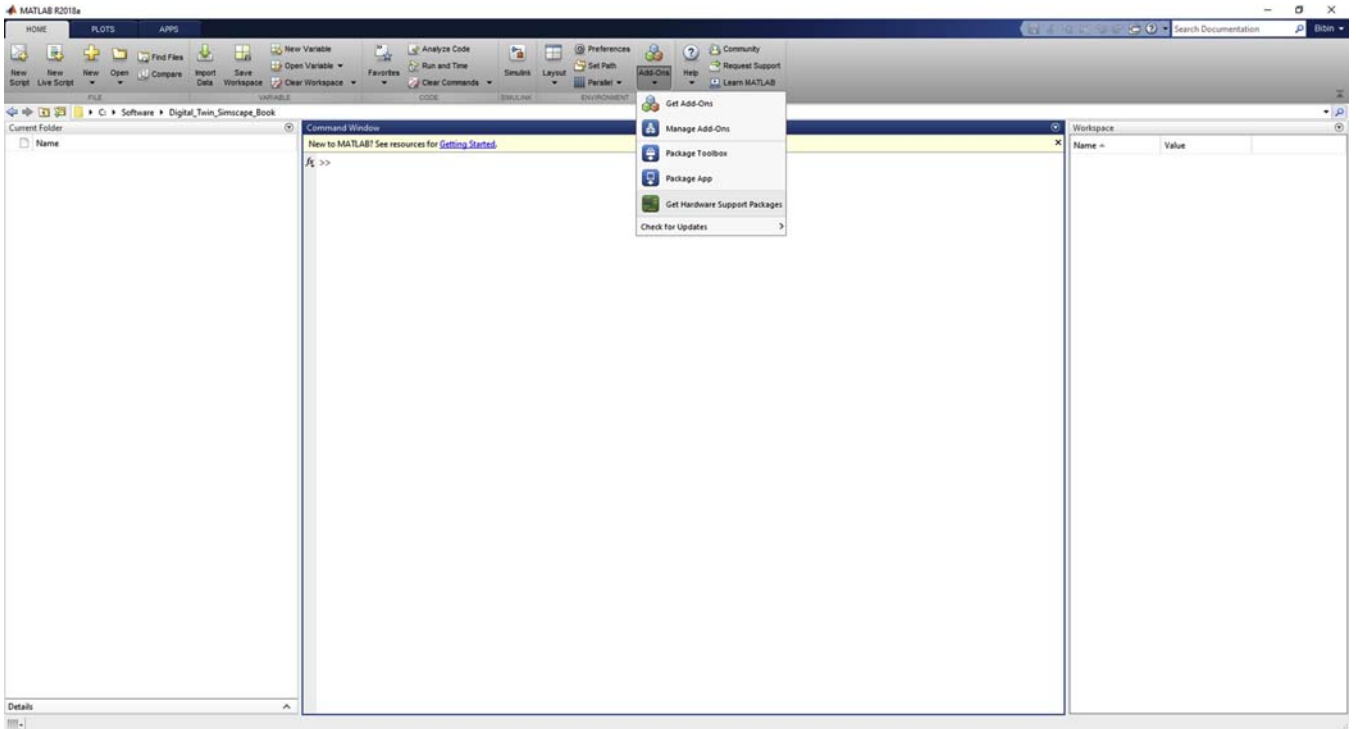


Figure 8.22 Add-on Explorer GUI.

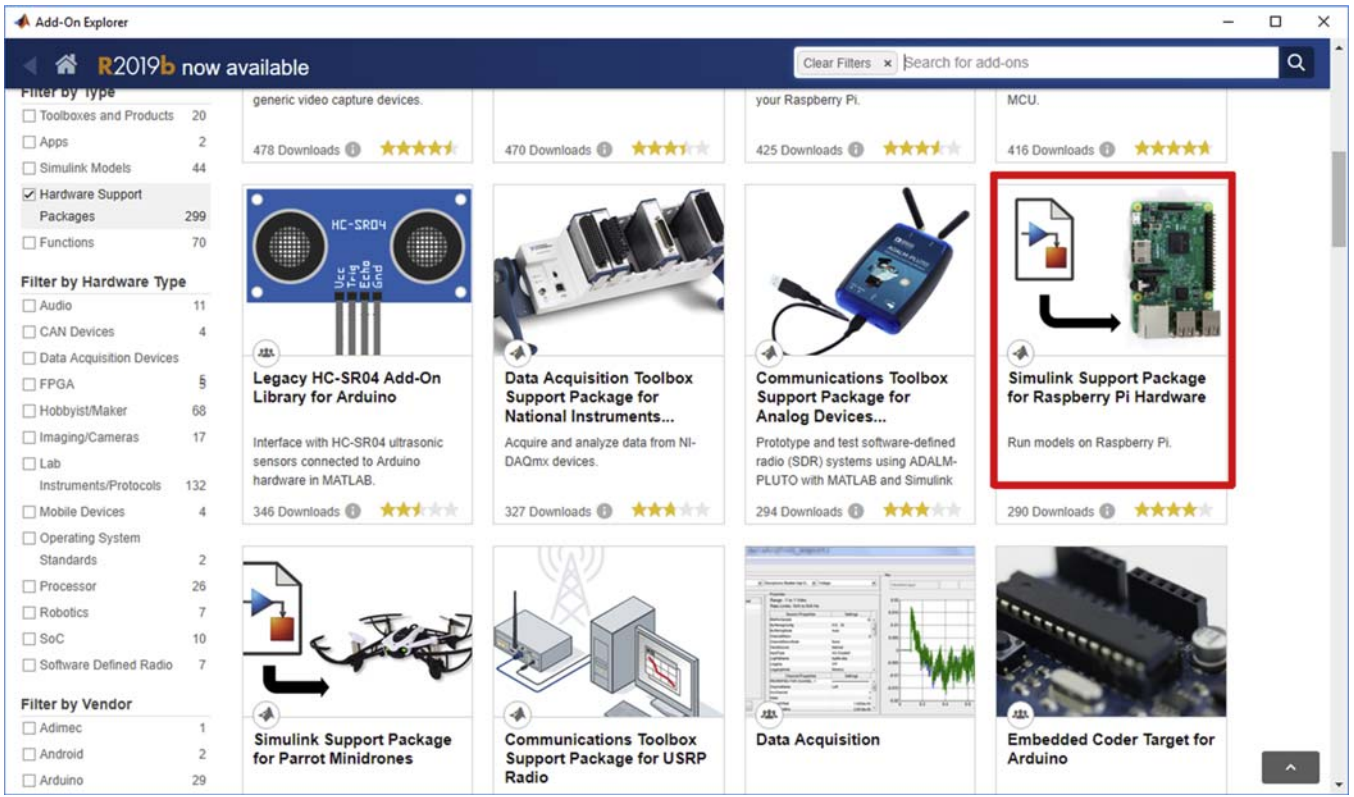


Figure 8.23 Simulink Support Package for Raspberry Pi Hardware.

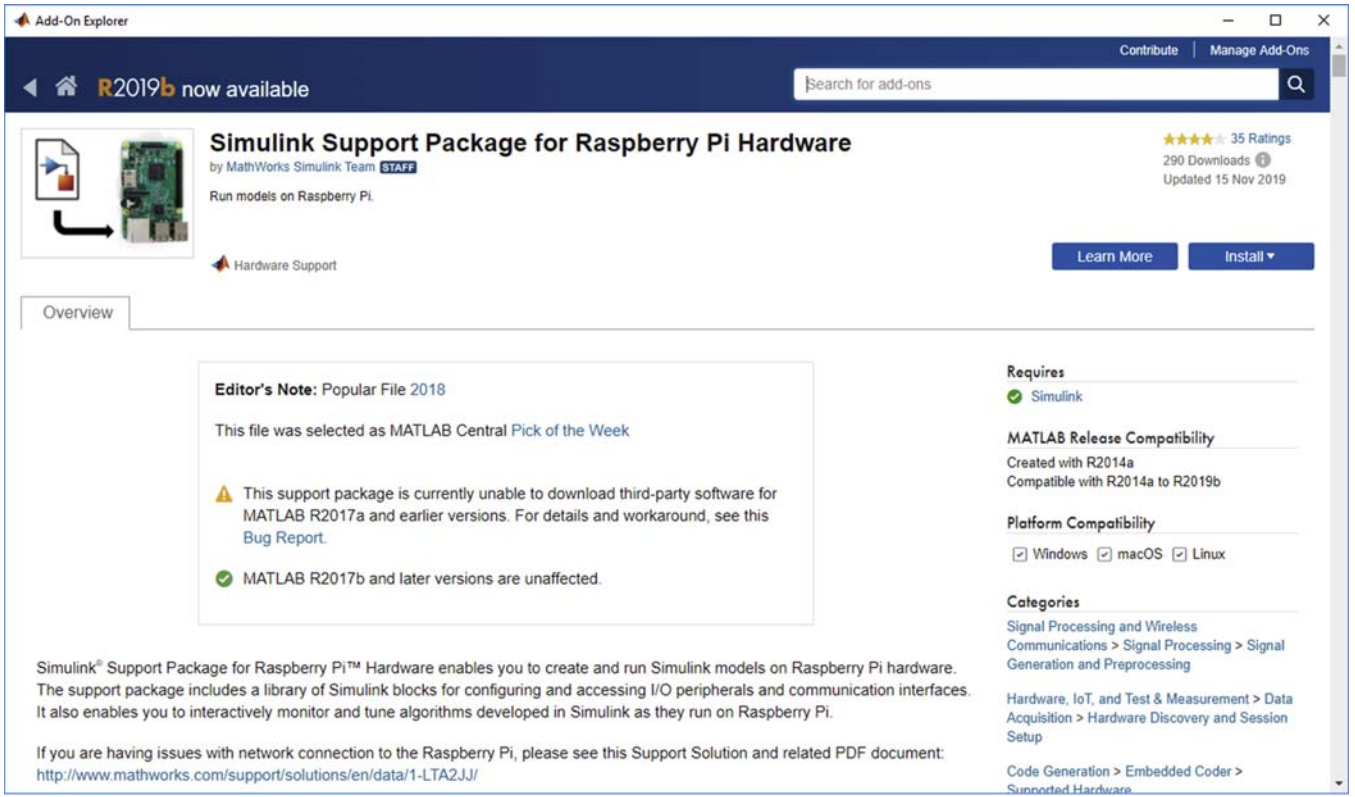


Figure 8.24 Add-On Explorer with Install Button.

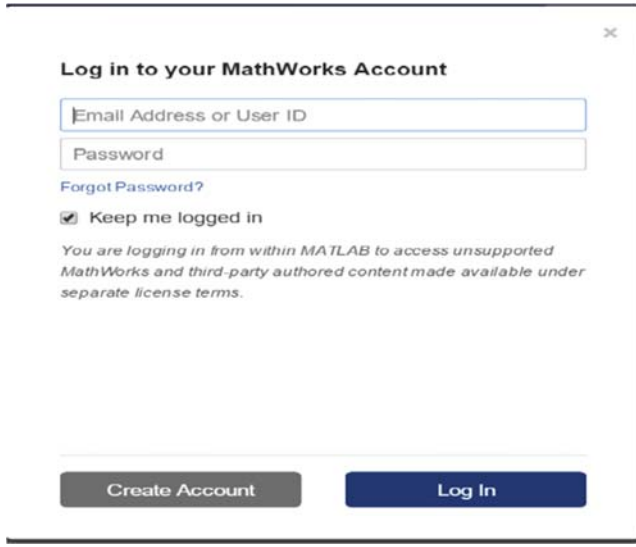


Figure 8.25 MathWorks account login.

8.6 EDGE device setup and cloud connectivity

Now that the HEV model is running in real time in the Raspberry Pi hardware, the next step is to collect the inputs, outputs, and states from the hardware and send to the AWS cloud Digital Twin for the Off-BD processing. As mentioned earlier, we will be using the MQTT communication protocol for communicating between the hardware and the AWS. So first we need to add the MQTT interface to the Simulink model. Unfortunately, in MATLAB 2018a Raspberry Pi Hardware support package, the MQTT interface is not included by default, but because of the limitation mentioned earlier for running interactive simulations on the PI hardware, we have to use MATLAB 2018a. So the Authors have included a few files that need to be copied to the Raspberry Pi Hardware package installation folder. The hardware installation folder can be easily found by typing ***which embdlinuxlib*** on the MATLAB command window.

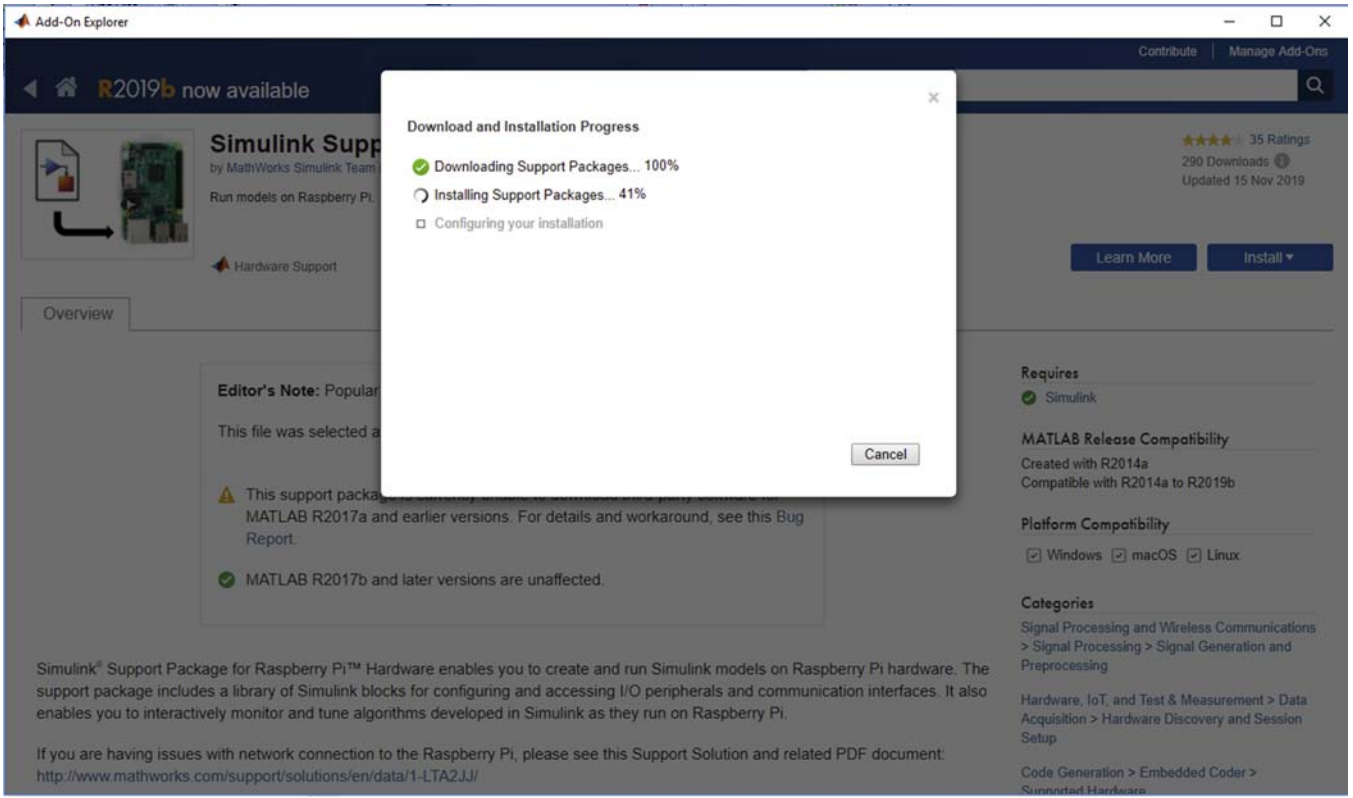


Figure 8.26 Support package installation progress.

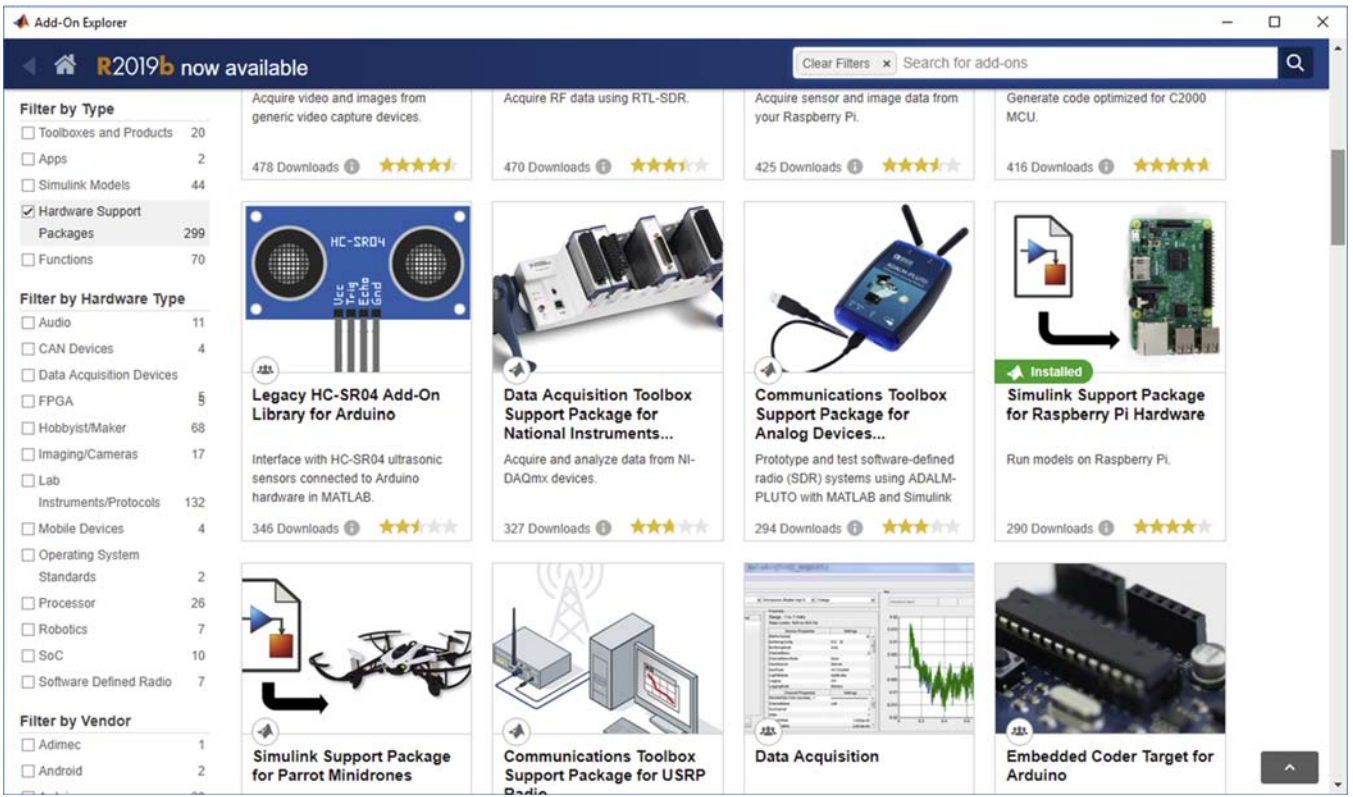


Figure 8.27 Finished installation for Raspberry Pi hardware.

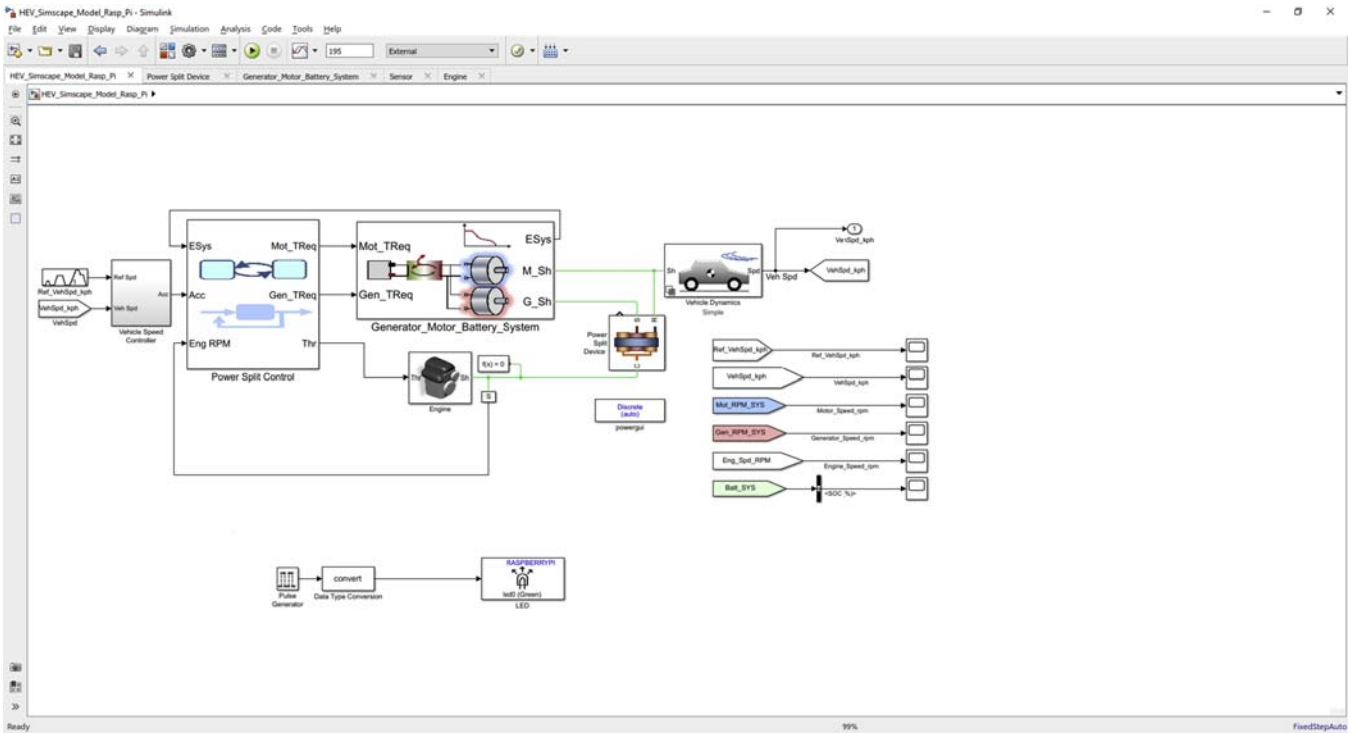


Figure 8.28 HEV Simscape™ configured to run on Raspberry Pi.

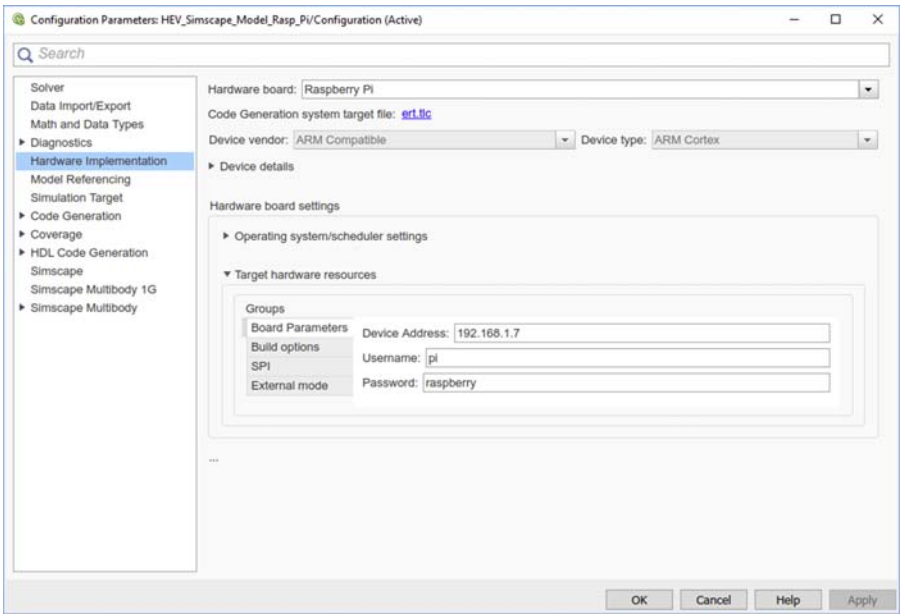


Figure 8.29 Updating the Raspberry Pi IP address and login credentials on the model.

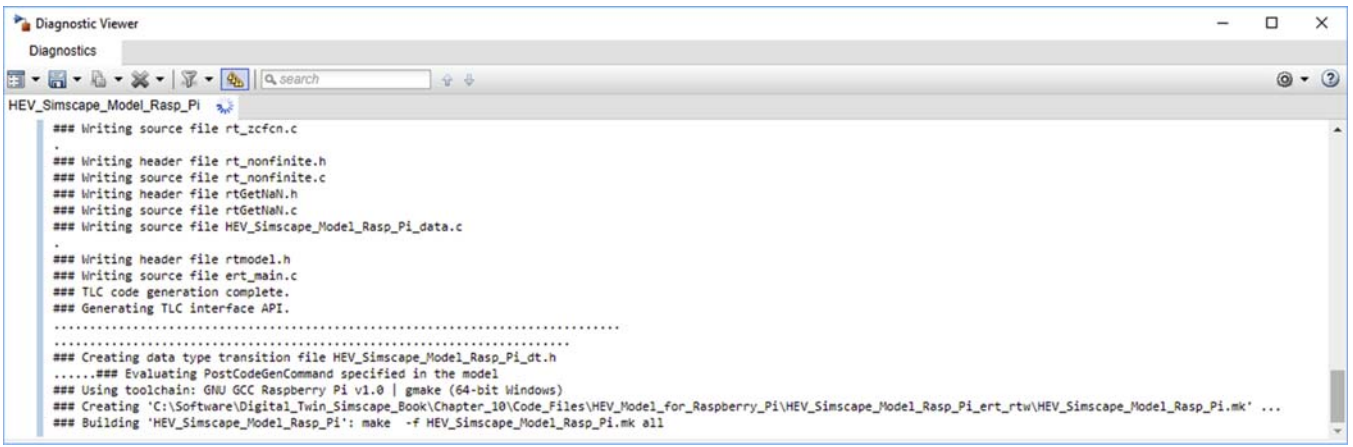


Figure 8.30 Building and downloading HEV Simulink model to Raspberry Pi hardware board.

1. Copy the **MQTTPublish.m** and **MQTTSubscribe.m** from the Chapter 8 attachment folder **Code_Files\HEV_Model_for_Raspberry_Pi_with_MQTT\MQTT_Support_Files** to the Raspberry Pi hardware installation folder **toolbox\realtime\targets\linux\+codertarget\+linux\+blocks**.
2. Copy **MW_MQTT.c** from the Chapter 8 attachment folder **Code_Files\HEV_Model_for_Raspberry_Pi_with_MQTT\MQTT_Support_Files** to **toolbox\realtime\targets\linux\src**.
3. Copy **MW_MQTT.h** from the Chapter 8 attachment folder **Code_Files\HEV_Model_for_Raspberry_Pi_with_MQTT\MQTT_Support_Files** to **toolbox\realtime\targets\linux\include**.

Open the model **HEV_Simscape_Model_Rasp_Pi_with_MQTT.slx** from the Chapter 8 attachment folder under **Code_Files\HEV_Model_for_Raspberry_Pi_with_MQTT**. This model has the MQTT transmit block added to it as shown in Figs. 8.31 and 8.32. Basically all the required signals that we want to send from the hardware to the AWS cloud are connected to a Mux block in the Simulink model along with a free running counter block as shown in Fig. 8.32 and then connected to an MQTT publish block. The publish block uses the topic name **digital_twin_mqtt_topic** for publishing the messages. This MQTT publish block is actually provided by the Raspberry Simulink library, but since MATLAB 2018a did not have the support for MQTT, this block is copied over from 2018b version of MATLAB and the PI hardware support library. The free running counter block is also connected to the MQTT block, and this counter serves as a timer indicator to buffer the data from the hardware for 5 s and then send the buffered data to AWS cloud. So the hardware sends MQTT message with the data every 0.1 s, which is the sample time of the model, and this MQTT message will be received by a Python program that is subscribed to the same MQTT topic **digital_twin_mqtt_topic**, and it buffers the data for 5 s, then formats the data to a JSON structure, and establishes a connection to AWS cloud and sends the message. Fig. 8.33 shows the high-level diagram of the cloud connection setup used in this chapter.

First we will start AWS side steps, we have to get some specific files when we setup AWS, which will then be used with the Python program for AWS−IoT connection. We will use the AWS service called AWS IoT Core in this section, which lets the connected devices securely interact with cloud or other connected devices. For more information about the AWS IoT Core services, follow the link https://aws.amazon.com/iot-core/. Fig. 9.115 shows the setup we are trying to establish in this section, which establishes a connection between ESP32 and AWS IoT Core and transmits data from Raspberry Pi hardware, which is connected to the ESP32.

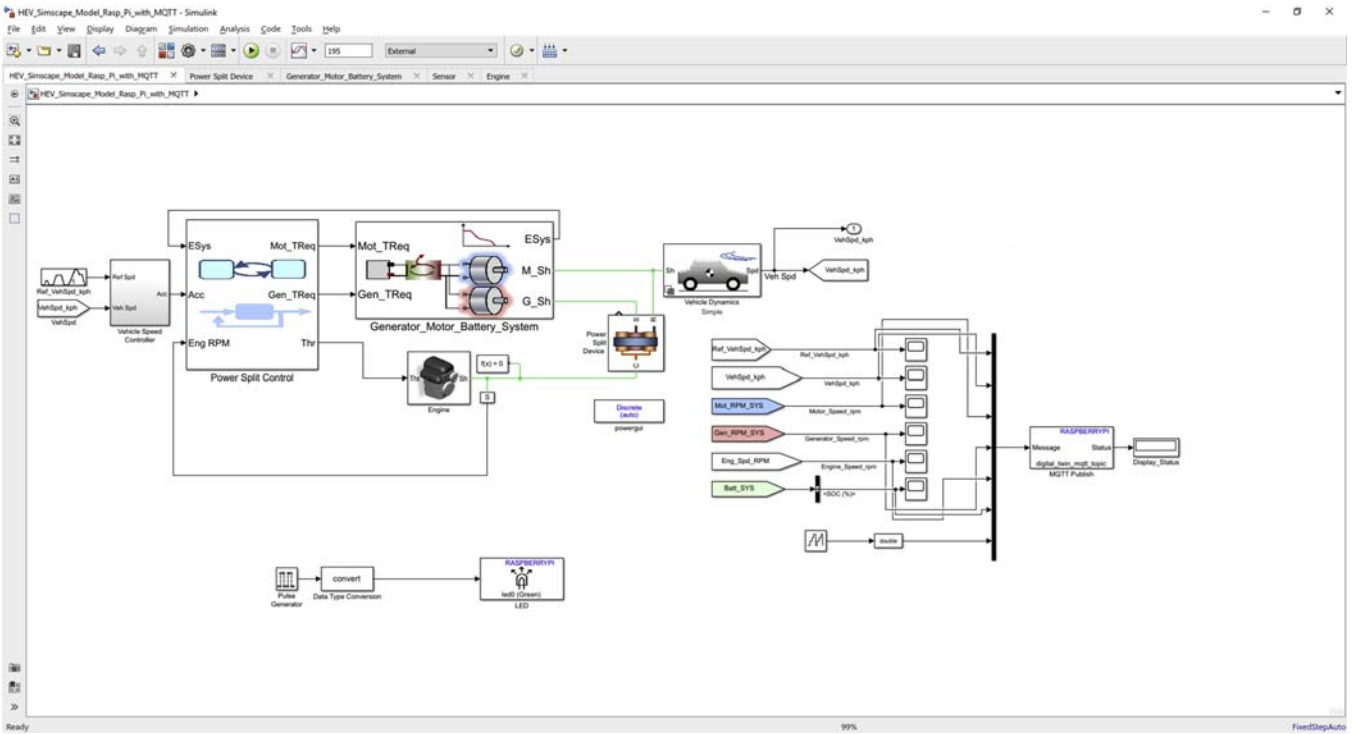


Figure 8.31 Raspberry Pi model with MQTT transmit logic.

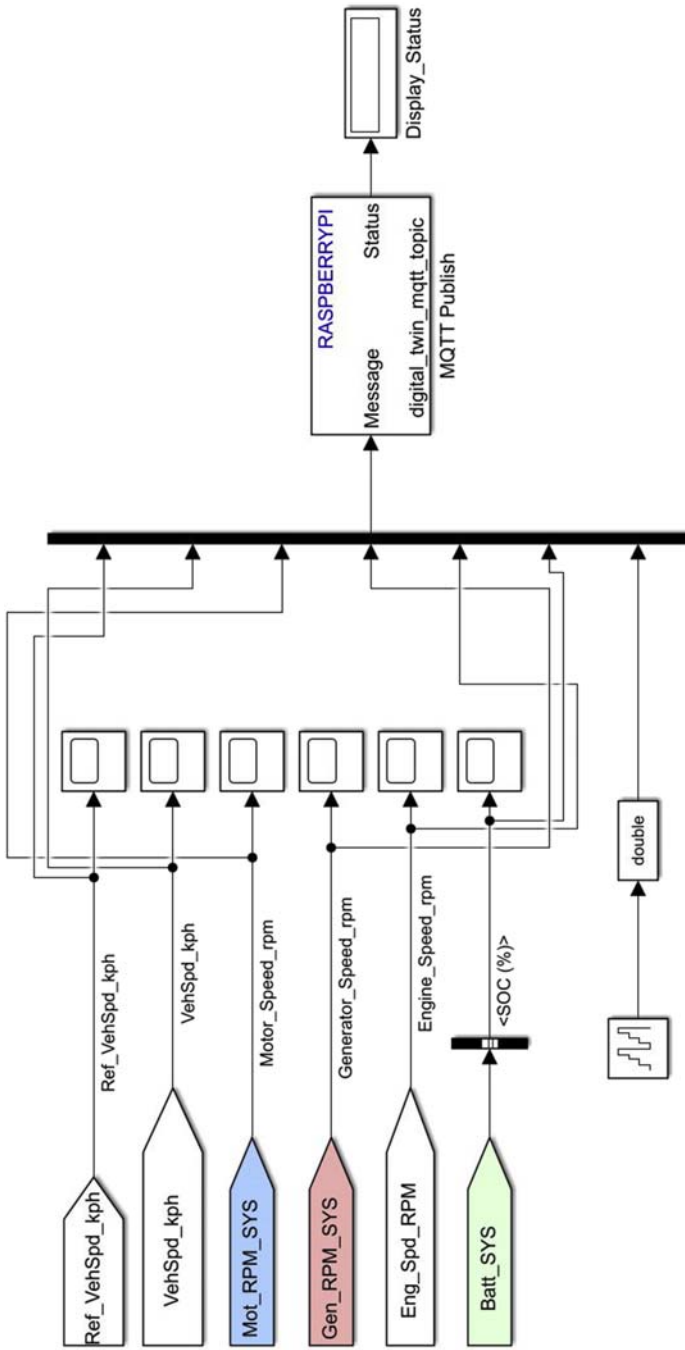


Figure 8.32 Close look at the MQTT transmit block.

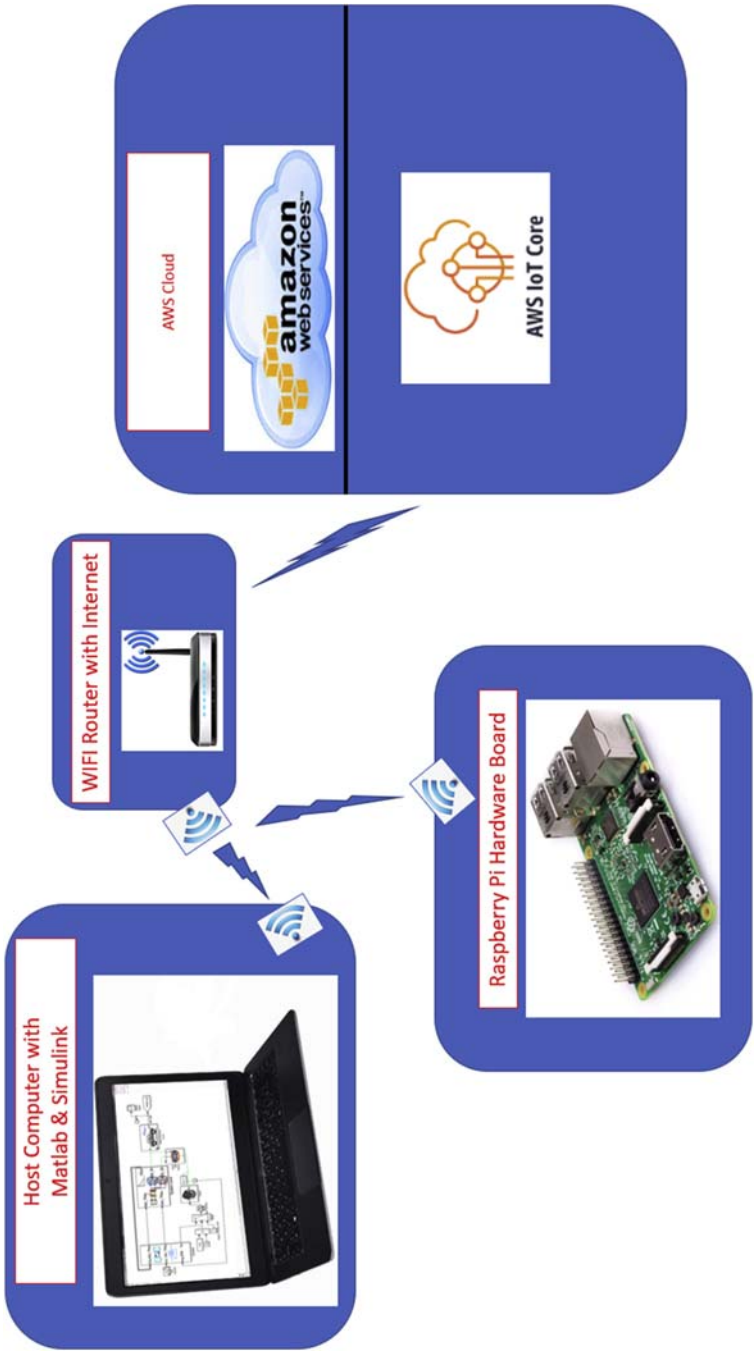


Figure 8.33 High-level diagram of the cloud connection setup.

1. As a first step, User needs to create a AWS Management Console account if you don't already have one. If you already have an Amazon account, you can use the same account information to login. If you don't have an account, please create an account. Follow this link https://aws.amazon.com/console/ on a web browser for logging in, creating a new account, etc. Use the ***Sign in to the Console*** option in the main page as shown in Fig. 8.34 and login using the credentials as shown in Fig. 8.35.
2. On the logged in Console window search for **IoT** and select the **IoT Core** as shown in Fig. 8.36. The AWS IoT window will be loaded, and it will display all the **AWS IoT Things** available in the User account. An **AWS IoT Thing** represents an **IoT** connected device, in this case, it will represent our Raspberry Pi connected to Wi-Fi and is running the HEV model in real time. The Author already has few **IoT Things** created previously on the account, which is the reason why it displays some **IoT Things** already as shown in Fig. 8.37. For security reasons, the already existing **IoT Things** are not shown in the images. We will be following similar graying out for some part of the images for protecting Author's information, but for the new additions that we are doing for this book, we will display it in the images.
3. Before creating a new **IoT Thing**, we need to create a **Policy.** A **Policy** is a JSON formatted document that allows access to the AWS services for the connected devices. We need to create a **Policy** and link it with the **IoT Thing** for the Raspberry Pi to establish a connection with the **AWS IoT Thing.** Go to **Secure >> Policies** on the left side tabbed window of the AWS Console under **AWS IoT** and click on the **Create** button as shown in Fig. 8.38.
4. On the new window enter the Policy name. This case we will name it as **hev_digital_twin_policy.** But User can choose to give a different name if needed. For the **Action** field enter **iot***, **Resource ARN** enter *, and for **Effect** check the **Allow** option and click on the Create button as shown in Fig. 8.39. The newly created Policy will show up under the Policies section as shown in Fig. 8.40.
5. The next step is to create an **AWS IoT Thing.** Goto **Manage >> Things** menu and click on **Create.** From the next window, click on **Create a Single Thing** as shown in Figs. 8.41 and 8.42.
6. Enter the name of the new **AWS IoT Thing** as shown in Fig. 8.43. We name it **hev_digital_twin_thing**, but User can choose a different name. We can leave the rest of the fields as default. Click **Next** at the bottom of the page.
7. We will use a certificate-based authentication for the Raspberry Pi device to connect to the **AWS IoT Thing** for secure connection. So we have to generate and download the certificates and use it with the Python program for the Raspberry Pi while establishing connection with **AWS IoT.** From the IoT Thing creation window, shown in Fig. 8.44, click on the **Create Certificate** option. It will generate a private key, a public key, and a certificate file as shown in Fig. 8.45. Download the certificate for the Thing, public key, private key and also the **Root CA** file to somewhere safely into the computer. We will be using these files later for programming the Raspberry PI to establish secure connection with **AWS IoT.** Next we need to **Activate** the Thing by clicking on the **Activate** button.
8. Next we need to attach the **Policy** that we have created in Step 3 and 4 to this new **IoT Thing.** Click on the Attach Policy option shown in Fig. 8.45, and in the new window select the **digital_twin_policy** and click on Register Thing as shown in Fig. 8.46.

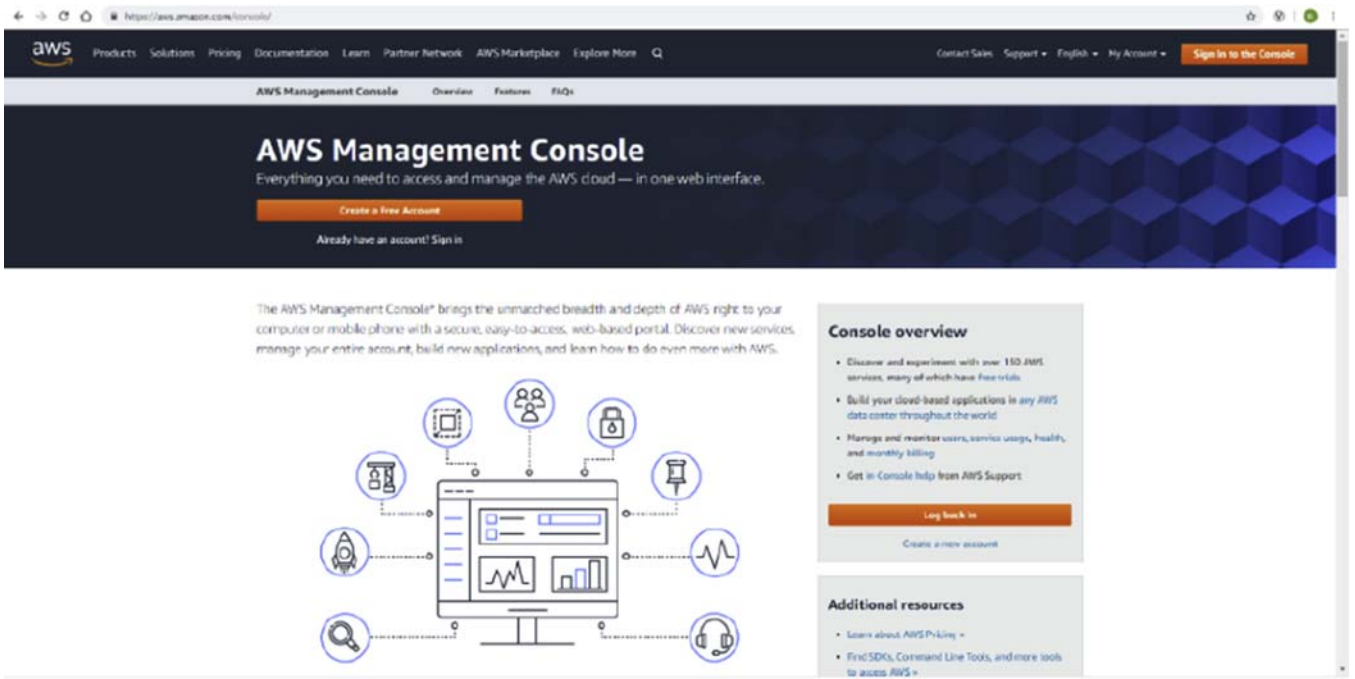


Figure 8.34 AWS console login page.

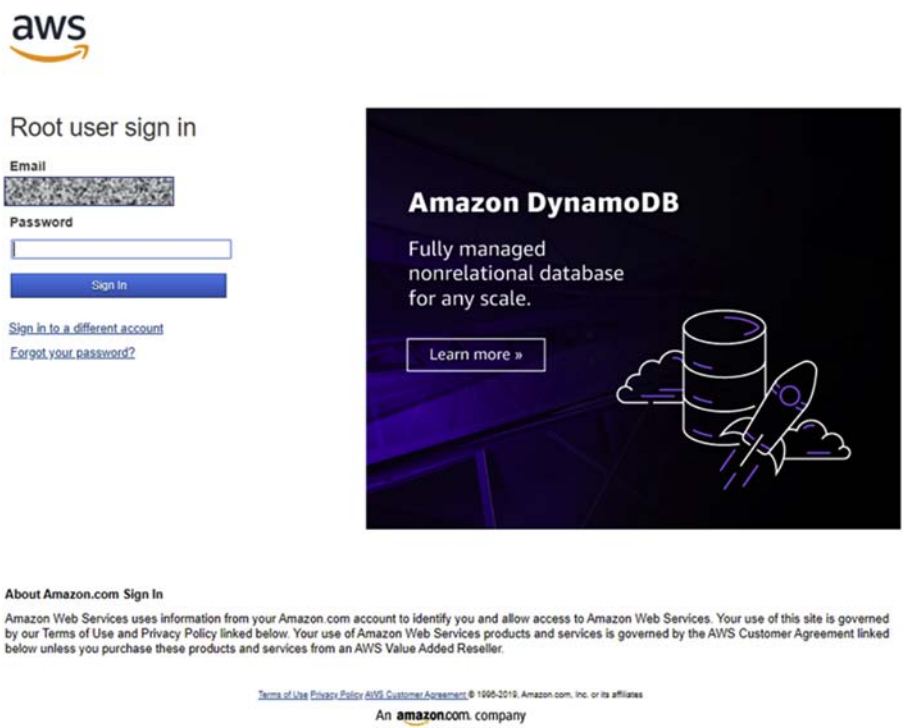


Figure 8.35 AWS console login.

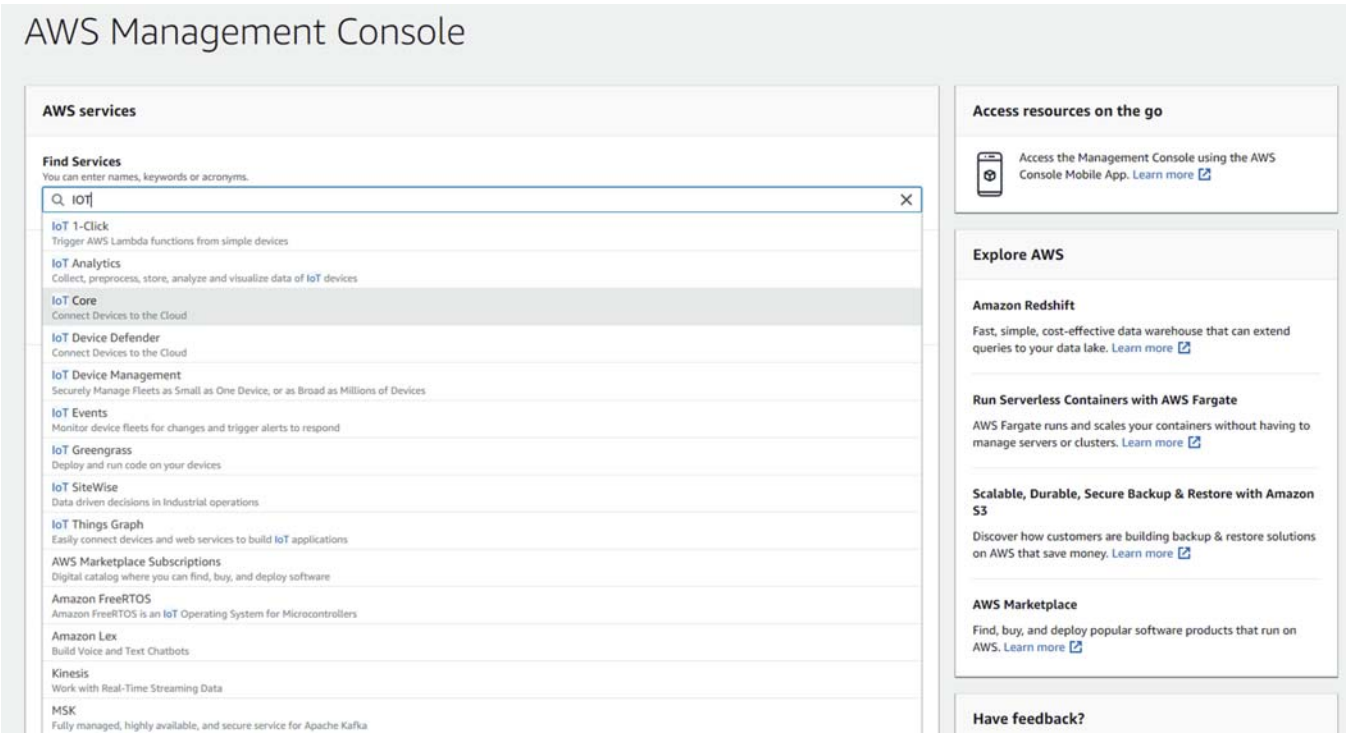


Figure 8.36 Selecting IoT Core services from AWS console.

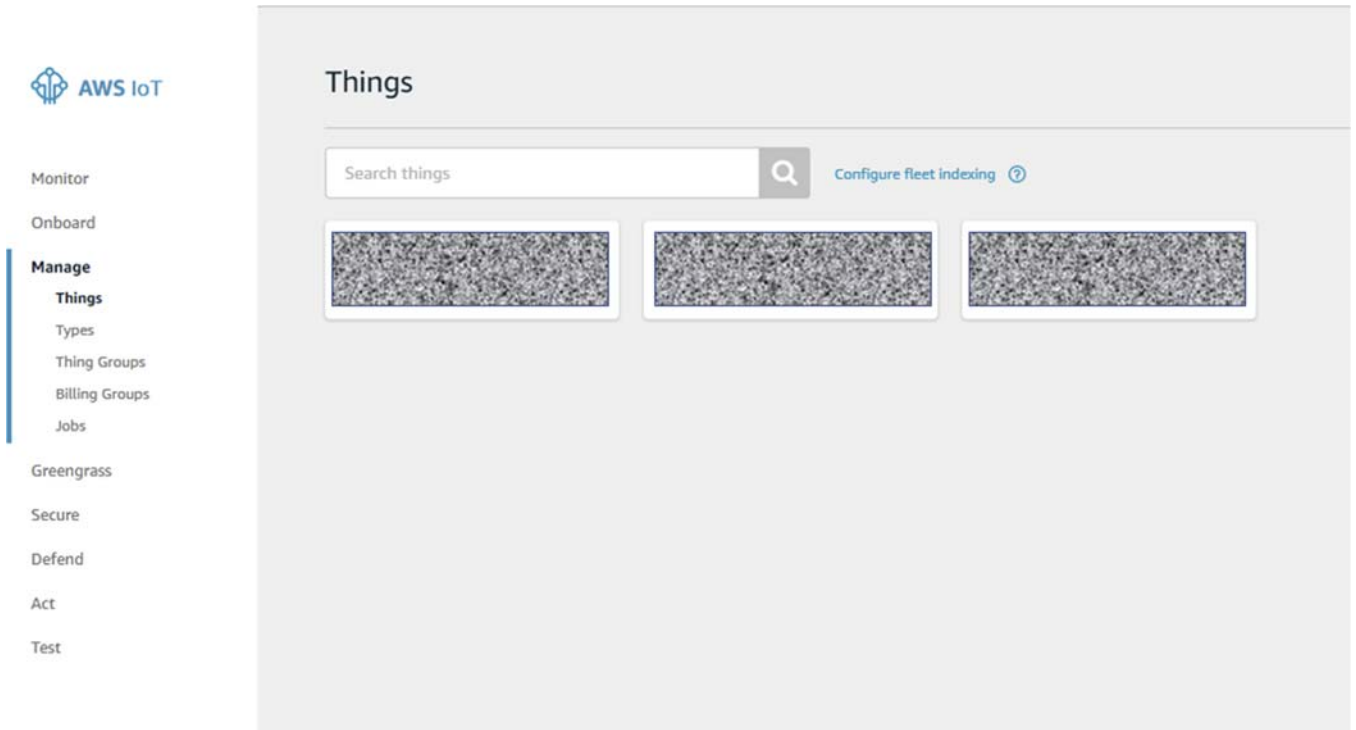


Figure 8.37 AWS IoT Things main page.

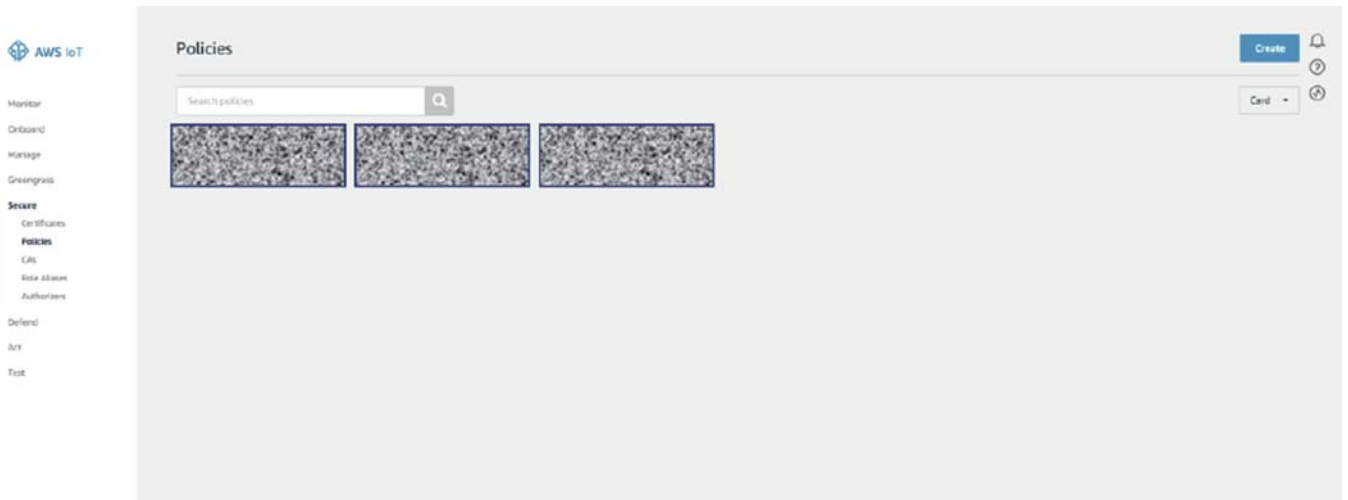


Figure 8.38 Creating a New Policy.

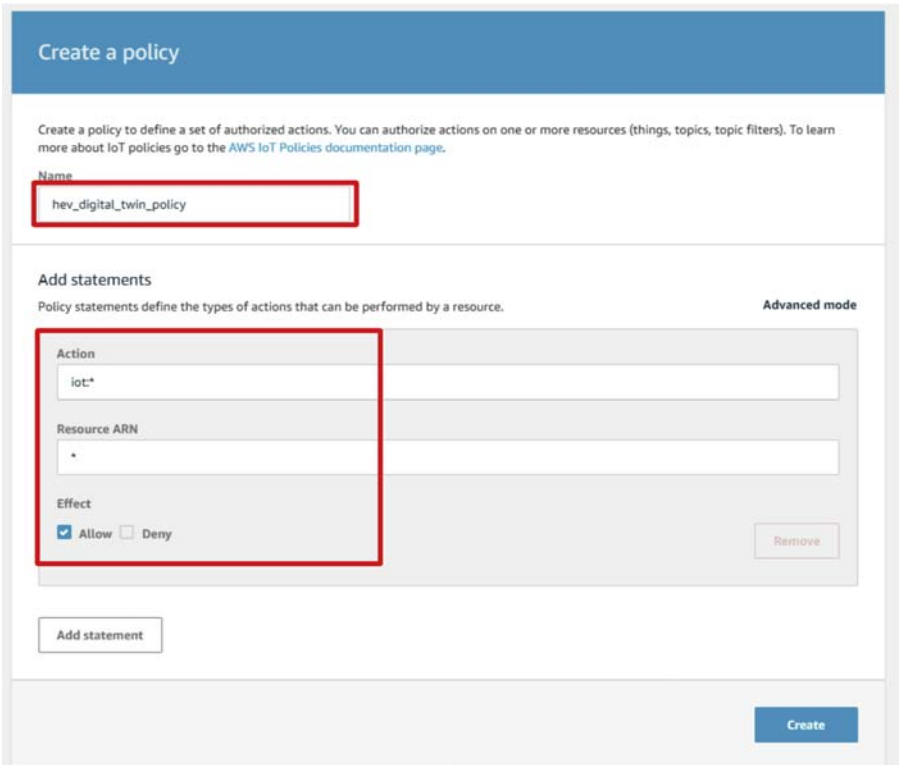


Figure 8.39 Entering New Policy Details.

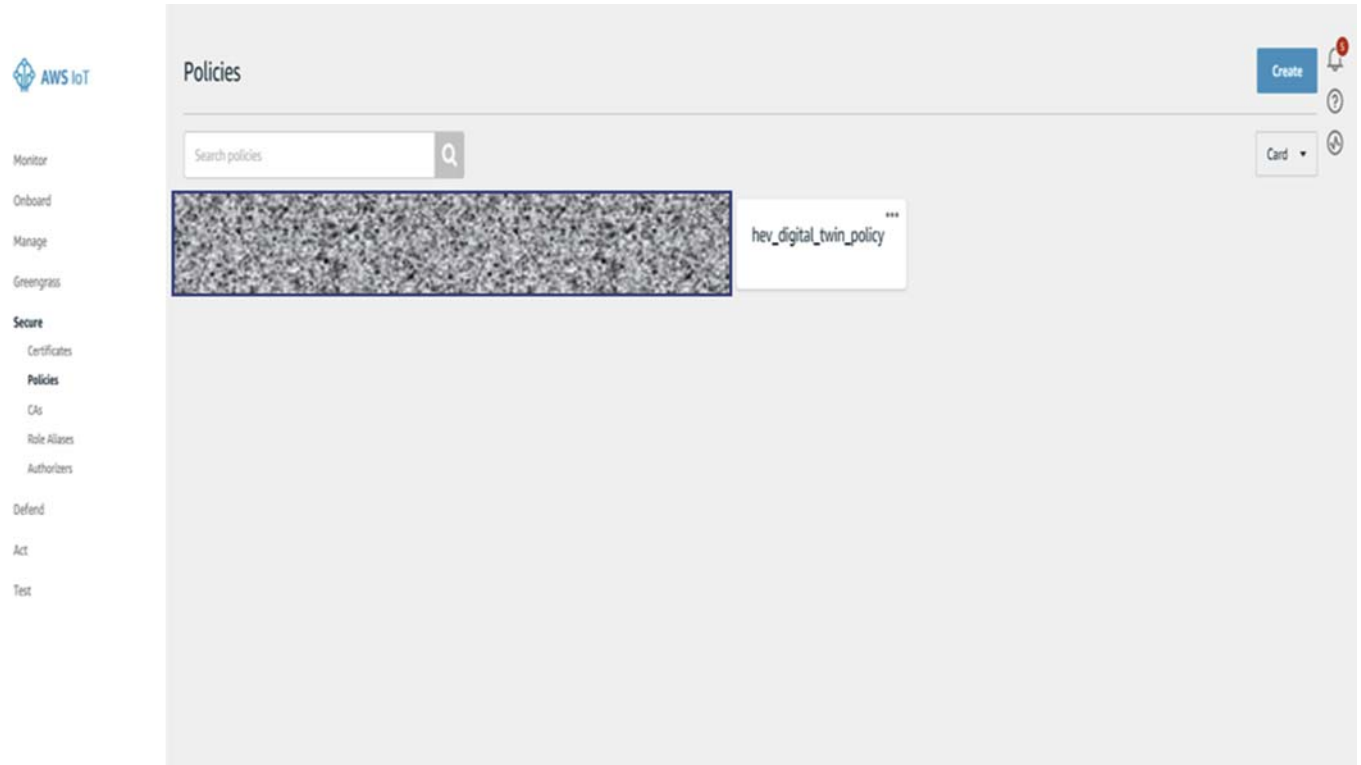


Figure 8.40 Newly created policy.

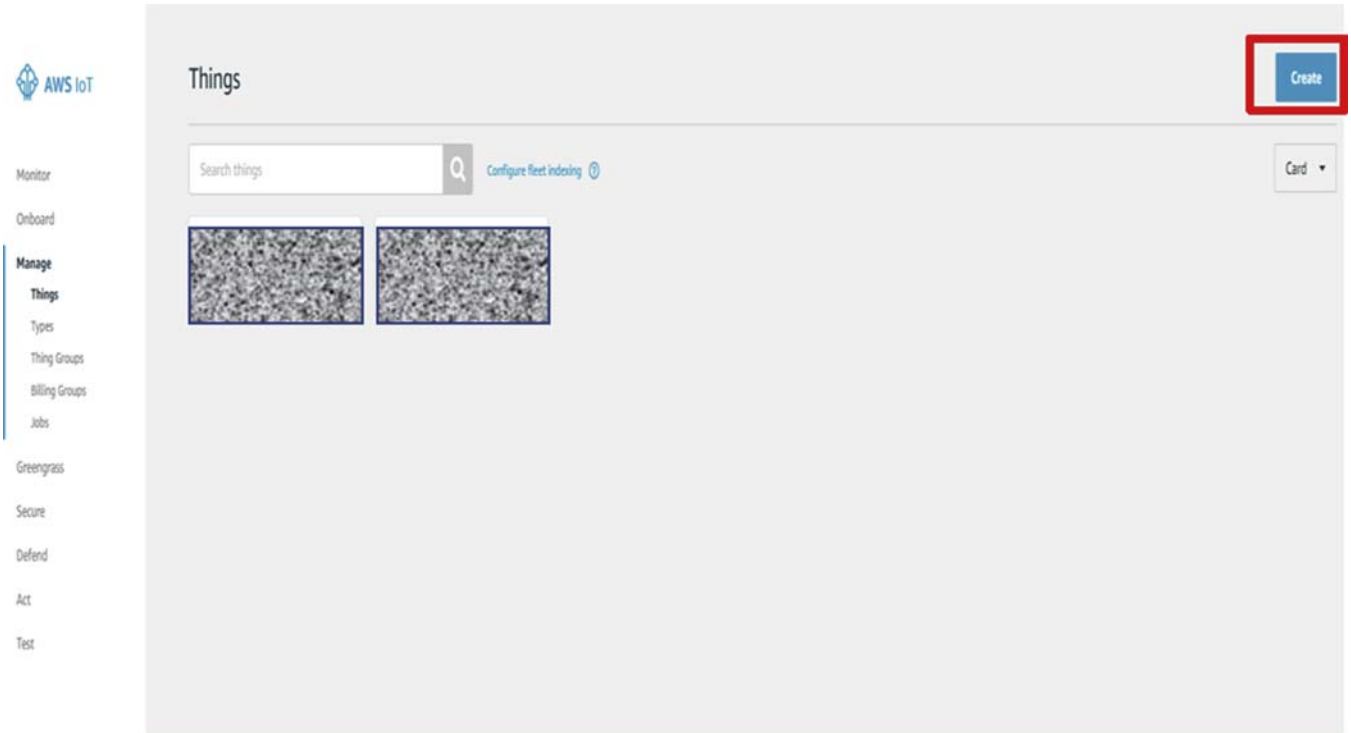


Figure 8.41 Creating an AWS IoT Thing.

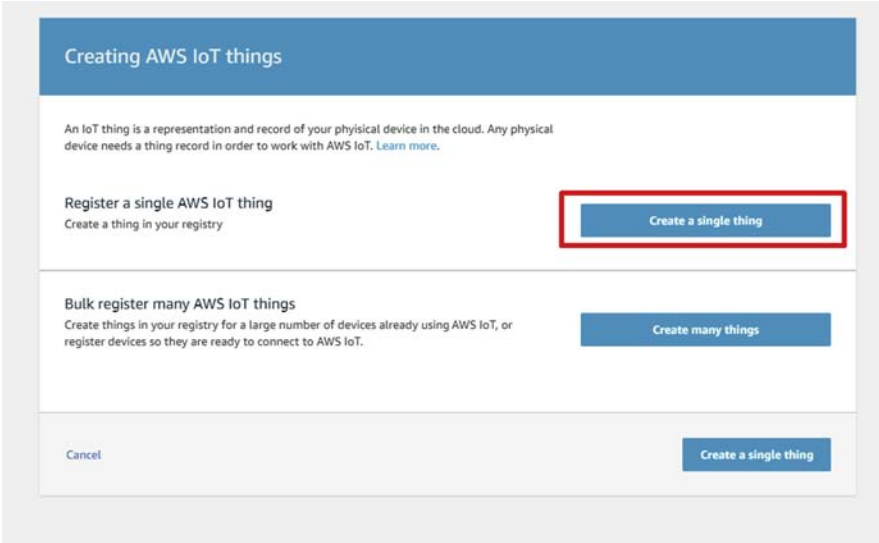


Figure 8.42 Create a single thing.

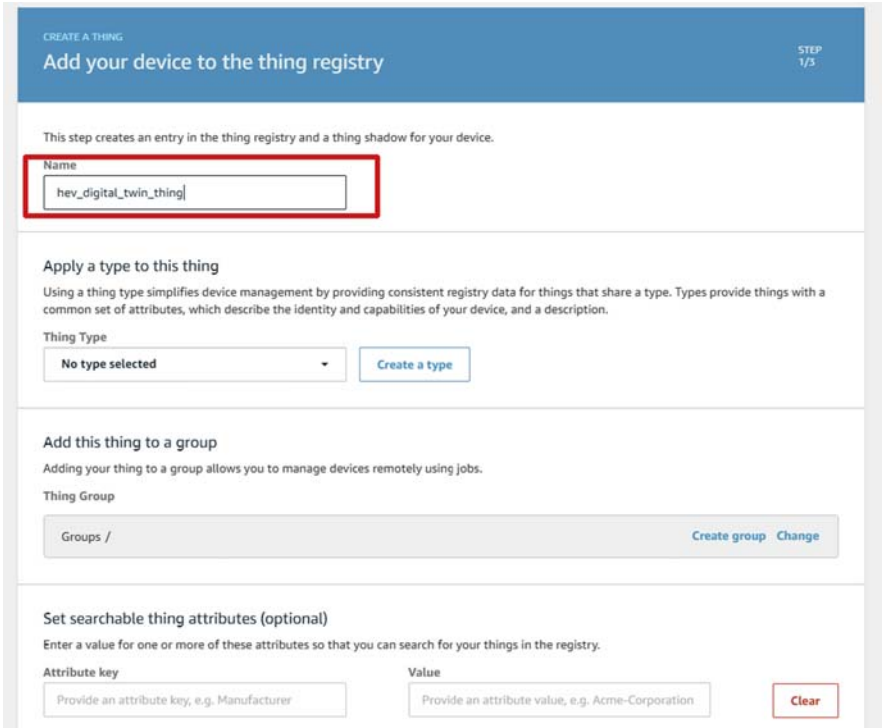


Figure 8.43 Naming the newly created AWS IoT thing.

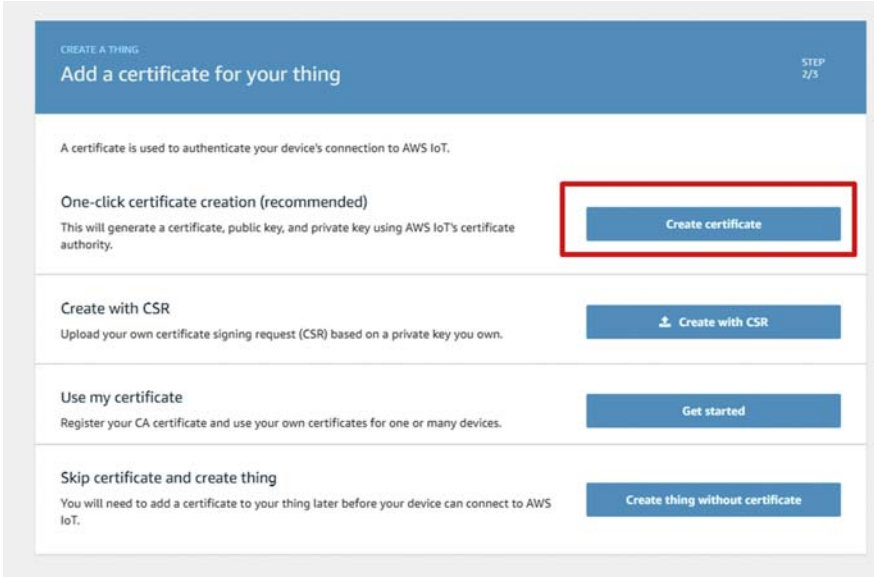


Figure 8.44 IoT Thing certificate creation window.

9. The newly created **IoT Thing** will show up in the **Manage >> Things** section. See Fig. 8.47.
10. Click on the **hev_digital_twin_thing** and go to the Interact menu as shown in Fig. 8.48 and note down the **Rest API** for the **IoT Thing**. For security reasons, the Author's API is not shown in the screenshot, but you need to copy the entire string, which will be used later for

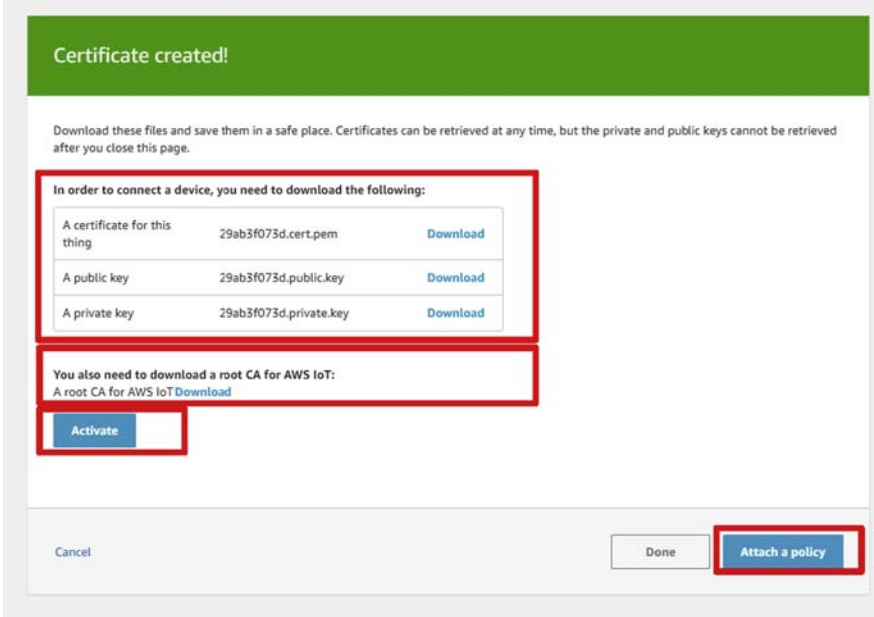


Figure 8.45 Certificates generated for IoT Thing.

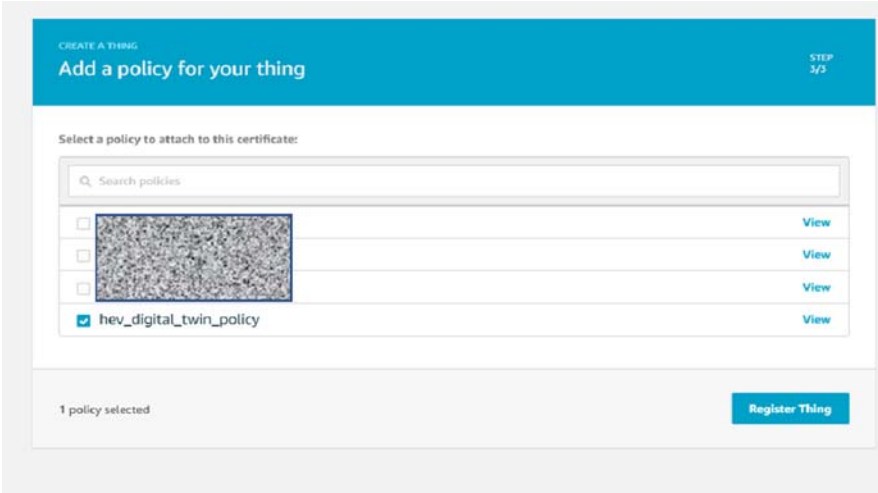


Figure 8.46 Attaching a policy and registering the IoT Thing.

the Raspberry Pi Python programming. Also note down the **IoT Thing** topic name under the MQTT section. MQTT is a fast, secure, and efficient protocol that we use to communicate data between Raspberry Pi and **IoT Thing.** MQTT nodes or end points publish and subscribe to the topics and exchange information between them using the topics.

11. Next step is to install AWS IoT SDK for Python in Raspberry Pi. To connect with AWS IoT services, we will use AWS IoT Python SDK, which is built on top of Paho MQTT client library, which we installed on Raspberry PI earlier in this chapter. For installing Python SDK, download the SDK package from the link below: https://s3.amazonaws.com/aws-iot-device-sdk-python/aws-iot-device-sdk-python-latest.zip.We can download this by logging into Putty Desktop App and go into a specific folder in Raspberry Pi and then type the command ***wget*** https://s3.amazonaws.com/aws-iot-device-sdk-python/aws-iot-device-sdk-python-latest.zip as shown in Fig. 8.49.
12. Unzip the package using the command ***unzip*** and the zip file name as shown in Fig. 8.50.
13. After unzipping, install the SDK using the command ***sudo python setup.py install*** as shown in Fig. 8.51.
14. After the SDK installation is finished, we will need to copy the highlighted folder AWSIoT-PythonSDK to the folder where we will be creating and running the Python code to send data to AWS IoT. In this example, we will run the Python code from the folder ***/home/pi/Digital_Twin_Simscape_Book/Chapter_8/Python_Code_Raspberry_PI_for_AWS_-Connection***. Use the command ***cp -r AWSIoTPythonSDK//home/pi/Digital_Twin_Simscape_Book/Chapter_8/Python_Code_Raspberry_PI_for_AWS_Connection*** to copy the AWSIoTPythonSDK to the above folder. See Fig. 8.52.
15. Next on the host computer, let us create a folder aws_certificates and copy the certificate files downloaded when we created the IoT Thing in Fig. 8.45. In this case, we have renamed the files as shown in Fig. 8.53. We can keep the file names as downloaded from the AWS as well, but just need to enter the correct file names in the Python code.

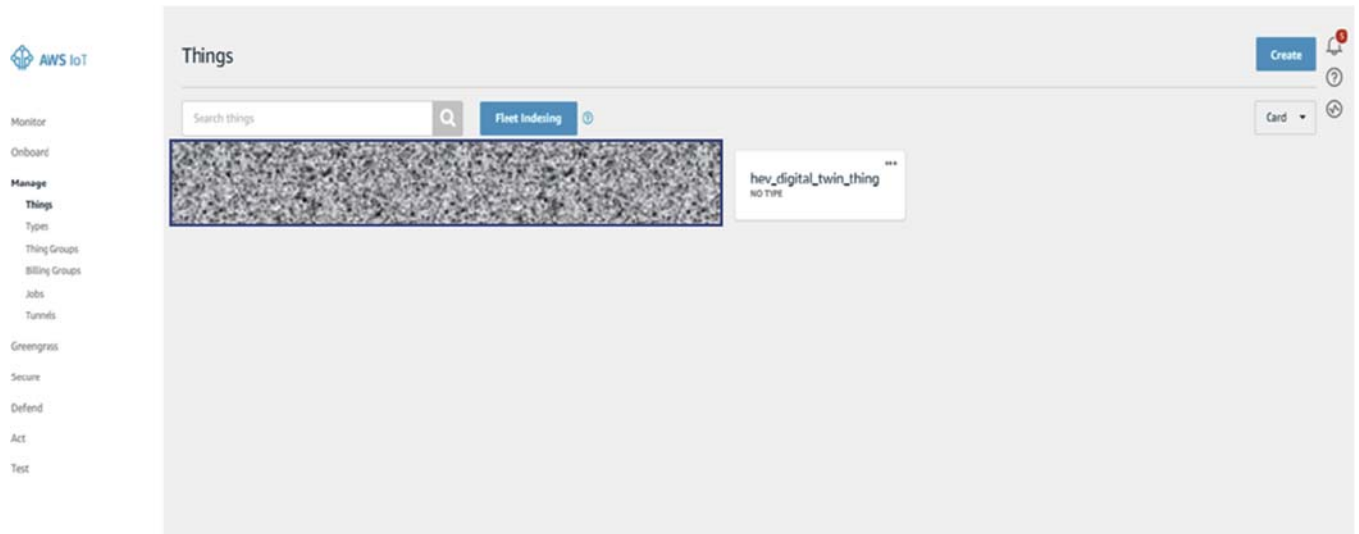


Figure 8.47 Newly created thing.

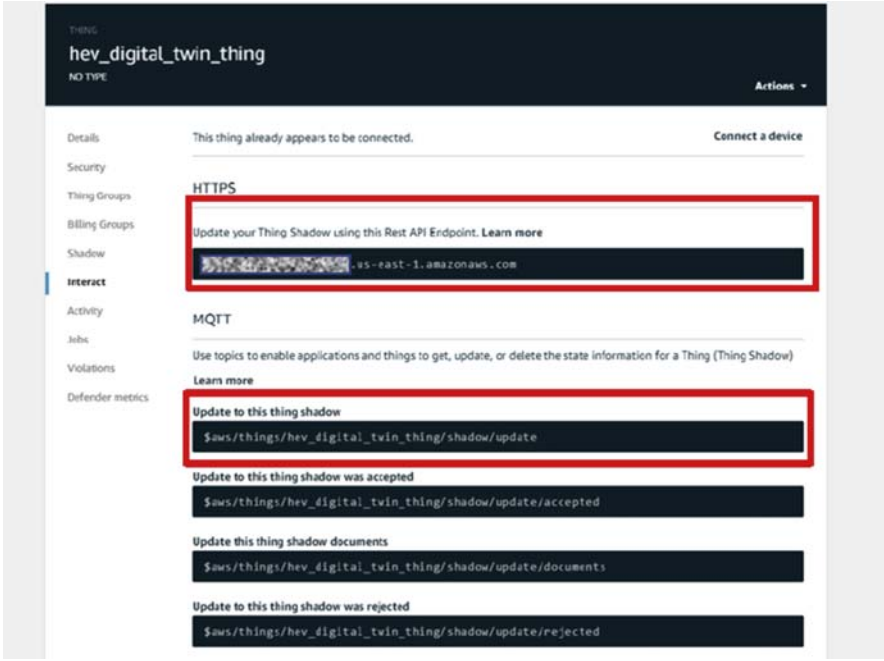


Figure 8.48 IoT Thing interact options.

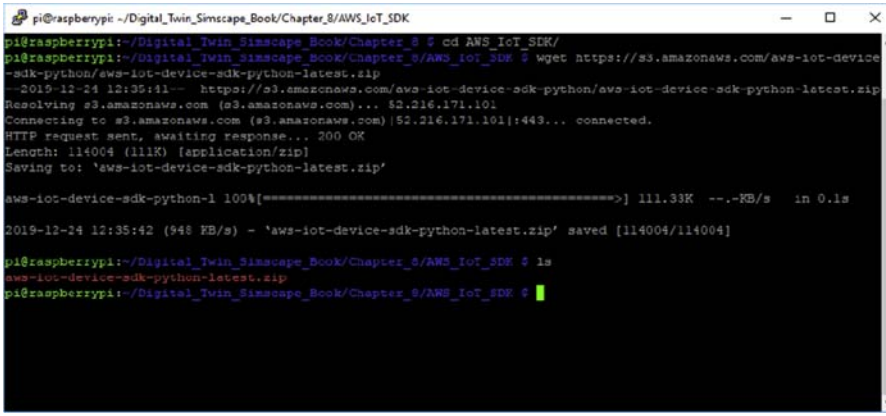


Figure 8.49 Downloading AWS IoT SDK.

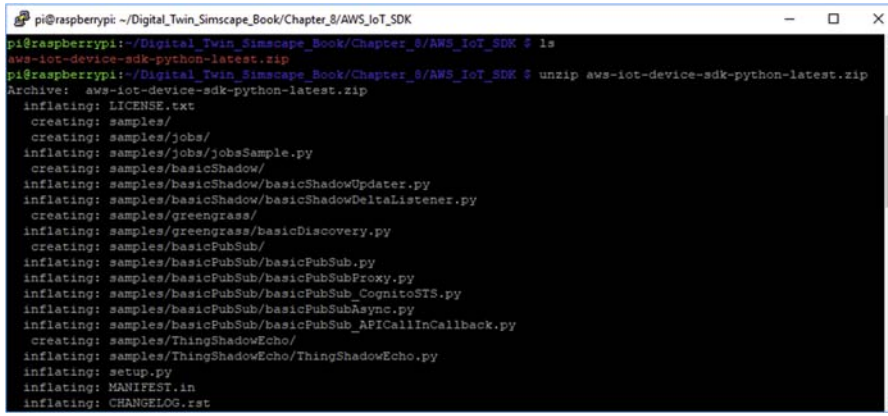


Figure 8.50 Unzip AWS IoT SDK.

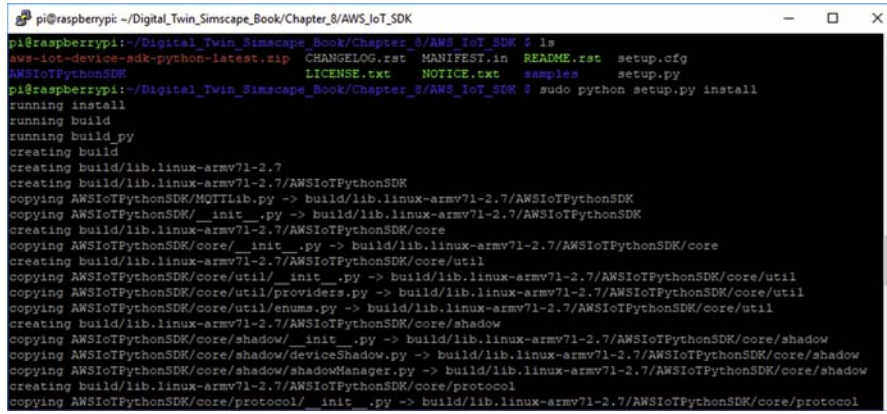


Figure 8.51 Installing AWS IoT Python SDK.

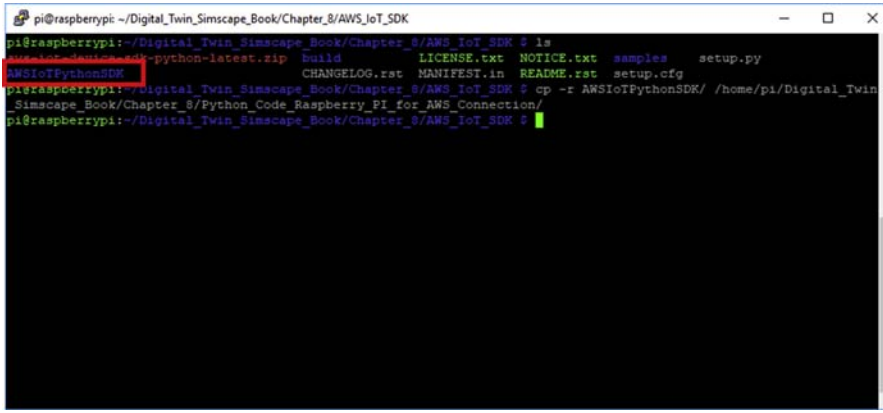


Figure 8.52 Copying the AWSIoTPythonSDK folder to the Python code folder.

16. Next on the host computer, use any programming editor (Authors used Notepad++) and create a new Python file ***Raspberry_Pi_AWS_IOT_Cloud_Connection.py.*** The file name doesn't matter, User can chose any name. On the new Python file, import the required libraries first as shown in Fig. 8.54.
17. In the next section in the Python code, setup the AWS IoT Thing details. Variable ***host*** should contain the string of the REST API for the IoT Thing, which we noted in Step 10. Variable ***certPath*** should hold the path in the Raspberry Pi hardware where we will copy the AWS certificate files. Variable ***clientId*** should hold the Thing name we created, and variable ***topic*** should hold the IoT Thing topic name noted from Step 10. Then initialize the object AWSIoTMQTTClient with the AWS Thing information and point it to the certificate files for the credentials for secure connection. See Figs. 8.55 and 8.56 for more details.
18. The next section of the Python code has the main() function, where it will connect to the local MQTT broker which is running on the Raspberry Pi with IP address 192.168.1.7 and subscribe to the MQTT topic ***digital_twin_mqtt_topic***. Note that this is the same topic to which Simulink HEV model running on the PI is publishing the message with the HEV current state information. So whenever the Simulink model runs and publishes messages to the broker, the Python code which is connected to the same broker will receive the messages. Also this section of the code configures a function to be called ***on_message*** whenever the code receives an MQTT message from the broker. And the main code just waits in a while loop for the messages to be received.
19. In the next section in the Python code, we have the implementation of the callback function on_message whenever the Python code receives an MQTT message from the MQTT broker with the topic ***digital_twin_mqtt_topic***. In a nutshell, this function receives the message with the HEV model state information such as Target and Actual Vehicle speed, Motor, Generator, and Engine speed, and Battery SoC, etc., buffers the data for 5 s, and creates a JSON structure with the buffered data and sends the data to AWS IoT Thing. After that, it clears the buffer and gathers next 5 s data and continues. Detailed comments are added to the code for better understanding. See Fig. 8.57 for more details. The entire code is available under the Chapter_8 attachment ***Code_Files\Python_Code_Raspberry_PI_for_AWS_Connection.***
20. Next step is to transfer the Python code and the AWS certificate files to Raspberry Pi hardware. Users can use any FTP client software to do that. Authors used a software named WinSCP. Login to the Raspberry Pi using WinSCP using the IP address and PI login credentials as shown in Fig. 8.58.
21. After the FTP connection is established, copy the Python file we created and the AWS certificates to the Raspberry Pi folder where we have copied the AWSIoTPythonSDK. So in this example, we are copying files from ***C:\Software\Digital_Twin_Simscape_Book\Chapter_8\Code_Files\Python_Code_Raspberry_PI_for_AWS_Connection*** to ***/home/pi/Digital_Twin_Simscape_Book/Chapter_8/Python_Code_Raspberry_PI_for_AWS_Connection***. See Fig. 8.59 for the copy process details.
22. Now we can run the Python code and the HEV Simulink model together in Raspberry PI and verify that the program we wrote is making the AWS connection and sending the HEV state information to the cloud. Open Putty Desktop App, go to the folder ***/home/pi/Digital_Twin_Simscape_Book/Chapter_8/Python_Code_Raspberry_PI_for_AWS_Connection*** and run the Python program using the command i***python Raspberry_Pi_AWS_IOT_Cloud_Connection.py*** as shown in Fig. 8.60
23. Next step is to run the Simulink model ***HEV_Simscape_Model_Rasp_Pi_with_MQTT.slx*** from ***Code_Files\HEV_Model_for_Raspberry_Pi_with_MQTT*** back again to run the

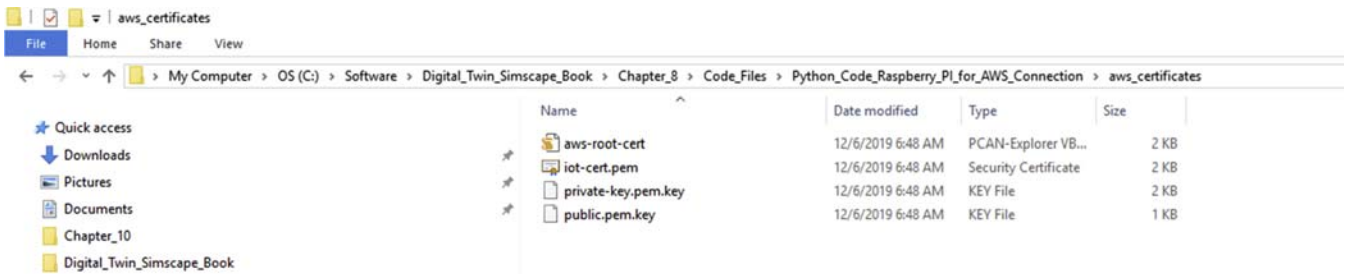


Figure 8.53 AWS certificates.

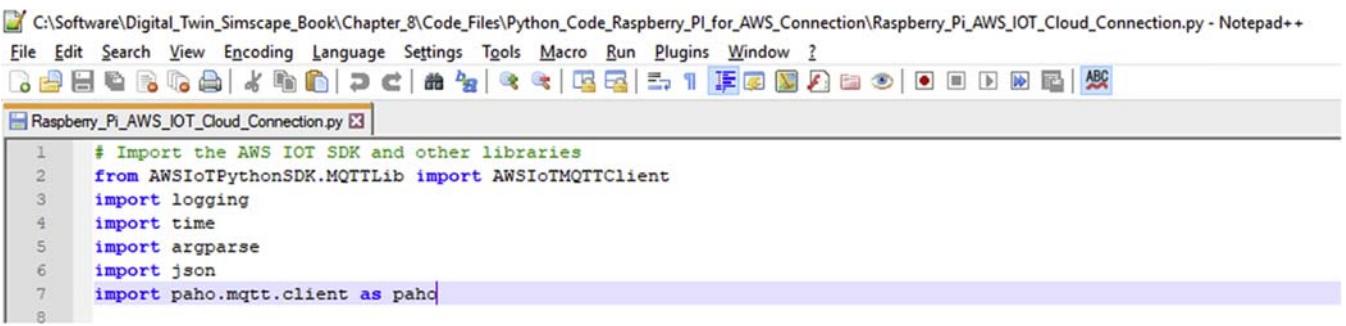


Figure 8.54 Importing libraries.

C:\Software\Digital_Twin_Simscape_Book\Chapter_8\Code_Files\Python_Code_Raspberry_PI_for_AWS_Connection\Raspberry_Pi_AWS_IOT_Cloud_Connection.py - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Raspberry_Pi_AWS_IOT_Cloud_Connection.py

```
# Setup the AWS IoT Host and the Certificate File Paths
host = "[redacted]-east-1.amazonaws.com"
certPath = "/home/pi/Digital_Twin_Simscape_Book/Chapter_8/AWS_IOT_Cloud_Connection/aws_certificates/"
# Setup the IoT Thing Name and Thing Topic Name for publishing data
clientId = "hev_digital_twin_thing"
topic = "$aws/things/hev_digital_twin_thing/shadow/update"

# Initialize Variables
jason_string_for_aws = {}
message_count = 0
prev_msg_data = ''
jason_string_for_aws_custom = '{"state": { "desired" : {'

# Initialize the AWS IoT MQTT Client, by pointing to the Root CA, Private Key and Certificate Files
myAWSIoTMQTTClient = None
myAWSIoTMQTTClient = AWSIoTMQTTClient(clientId)
myAWSIoTMQTTClient.configureEndpoint(host, 8883)
myAWSIoTMQTTClient.configureCredentials("{}aws-root-cert.pem".format(certPath), "{}private-key.pem.key".format(certPath), "{}iot-cert.pem.crt".format(certPath))

# Configure the AWS IoT MQTT Client Connection
myAWSIoTMQTTClient.configureAutoReconnectBackoffTime(1, 32, 20)
myAWSIoTMQTTClient.configureOfflinePublishQueueing(-1)  # Infinite offline Publish queueing
myAWSIoTMQTTClient.configureDrainingFrequency(2)  # Draining: 2 Hz
myAWSIoTMQTTClient.configureConnectDisconnectTimeout(10)  # 10 sec
myAWSIoTMQTTClient.configureMQTTOperationTimeout(5)  # 5 sec
myAWSIoTMQTTClient.connect()
```

Figure 8.55 AWS IoT−specific configuration for secure connection.

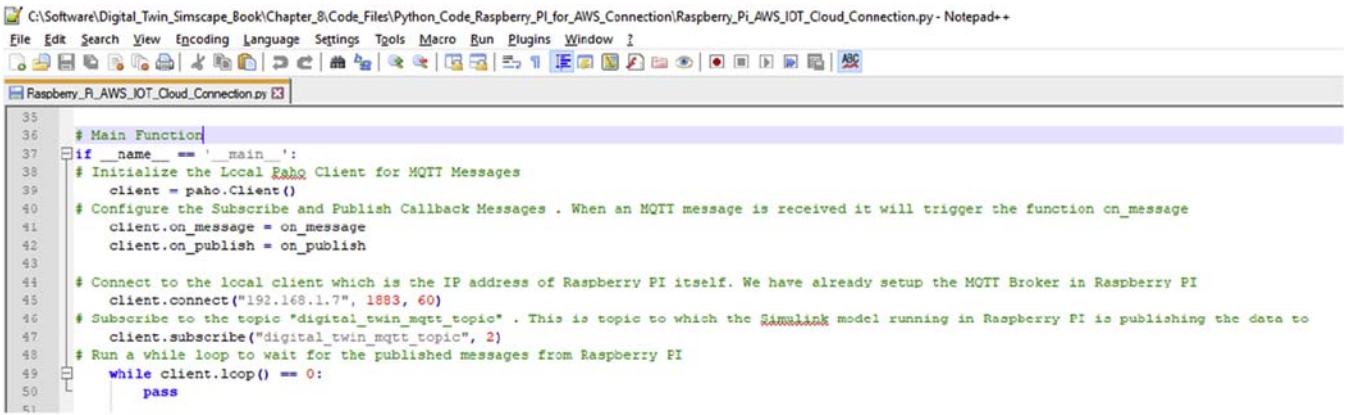


Figure 8.56 Main function to connect to local MQTT broker.

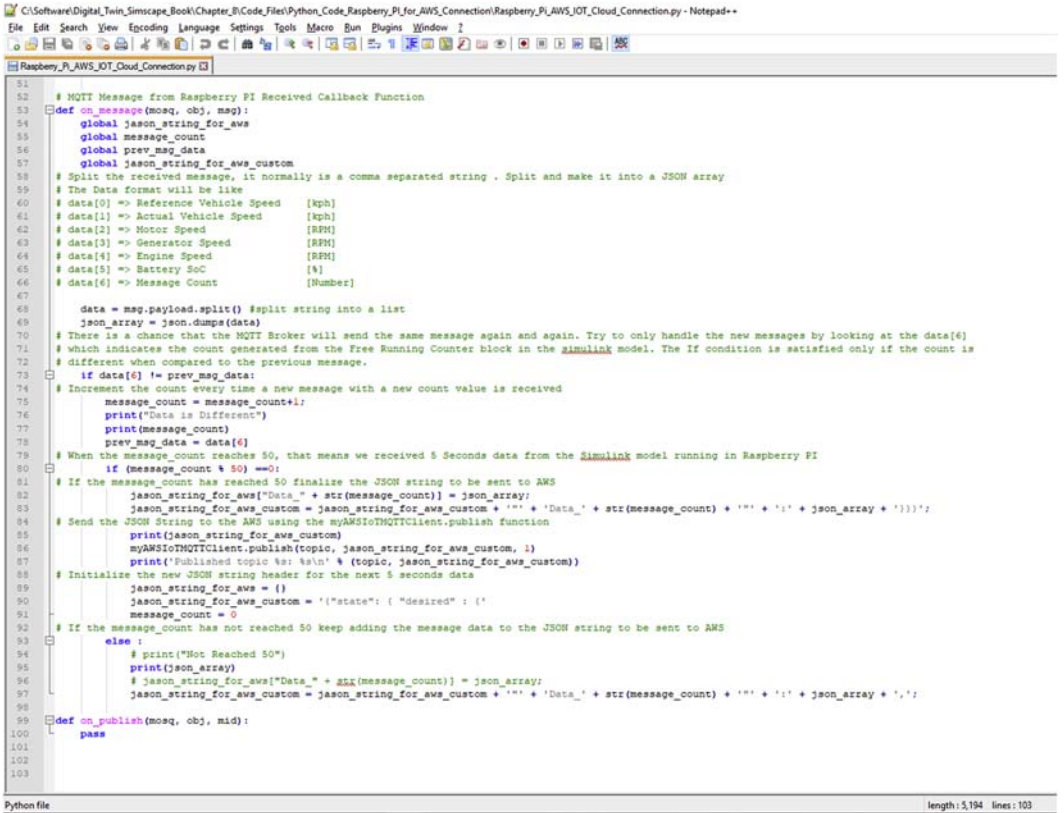


Figure 8.57 Callback function for MQTT message receive events.

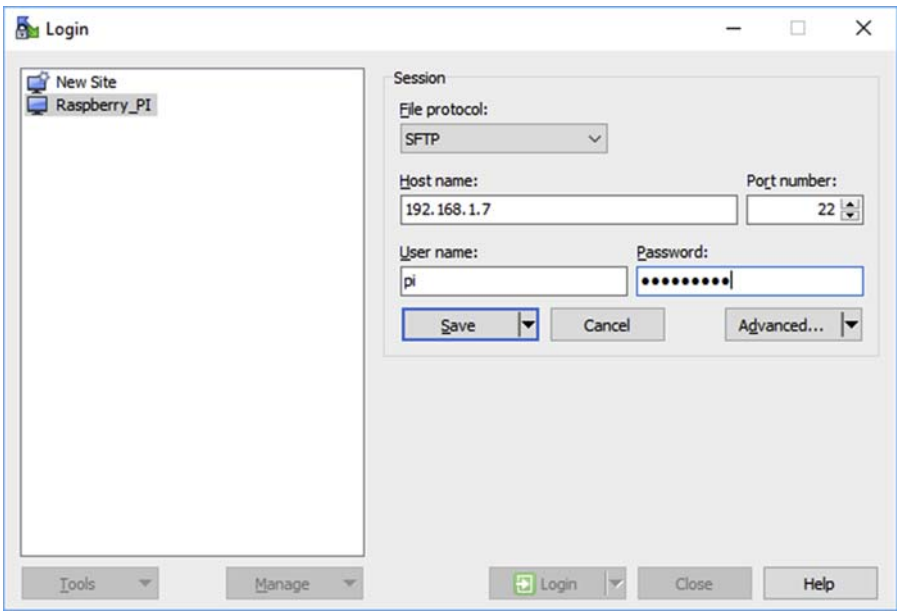


Figure 8.58 WinSCP FTP client to connect to Raspberry Pi.

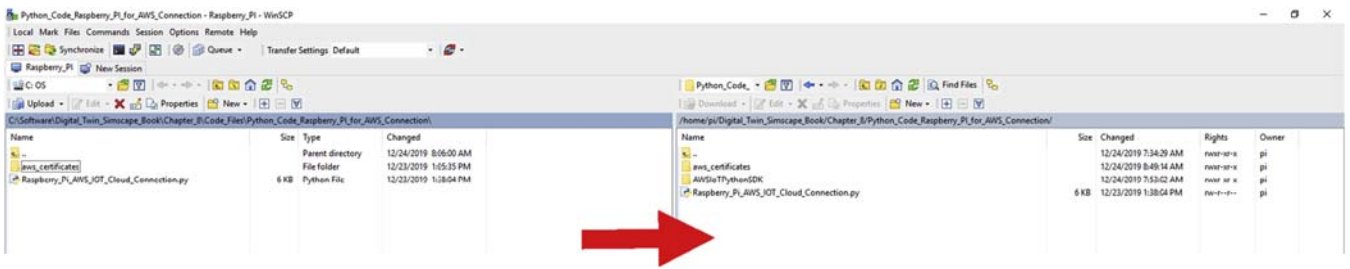


Figure 8.59 Copying Python code and AWS certificates from host computer to Raspberry Pi.

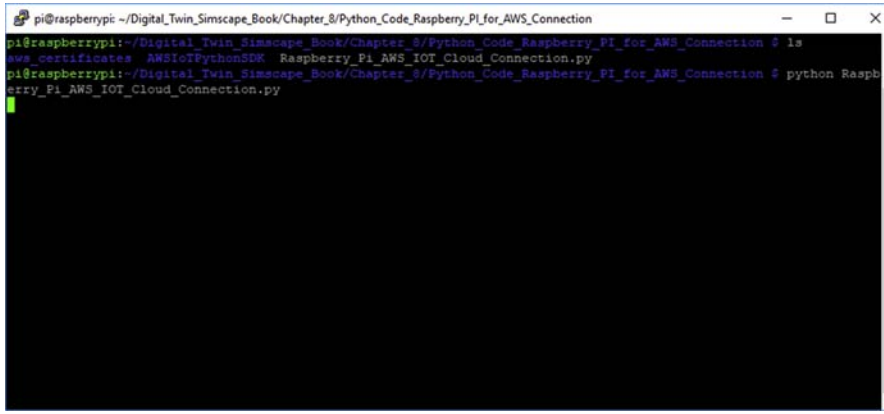


Figure 8.60 Running the Python code.

HEV model on the Raspberry Pi. We can see that as the HEV model starts to run in the PI, on the Putty window where we ran the Python code, it starts to receive the HEV state data as shown in Figs. 8.61 and 8.62.

24. In order to verify that the published message from the Raspberry Pi is reaching AWS IoT, go to the AWS console goto ***IoT Core*** >> ***Things*** >> ***hev_digital_twin_thing*** >> ***Activity*** and it will show the same JSON structure the Python code is publishing in real time. See Fig. 8.63. It can be noted that every 5 s we receive a new JSON structure with the buffered data for the previous 5 s.

8.7 Digital Twin Modeling and calibration

So far we have completed the hardware asset setup with Raspberry Pi and its cloud connectivity to send real time data to AWS cloud. Further we will be proceeding with the cloud side work to deploy the Digital Twin model and perform Off-BD. For the HEV example, we will be using the same Simulink model deployed into the Raspberry Pi hardware to the AWS cloud as well. So no further modeling or calibration efforts are needed other than the model we already have.

8.8 Off-board diagnostics algorithm development for Hybrid Electric Vehicle system

In this section, we will develop an algorithm to diagnose and detect the Throttle failure of the Hybrid Electric Vehicle system and test the diagnostic algorithm with the Digital Twin Model. We will first test the diagnostic algorithm locally on a computer before deploying it on the cloud.

The flow chart shown in Fig. 8.64 will be used to diagnose the Throttle failure conditions where we collect the input, output, and the states data from the HEV model running in Raspberry Pi hardware, run the Digital Twin model of the HEV on the cloud with the same input data to get the expected output and states, calculate the RMSE

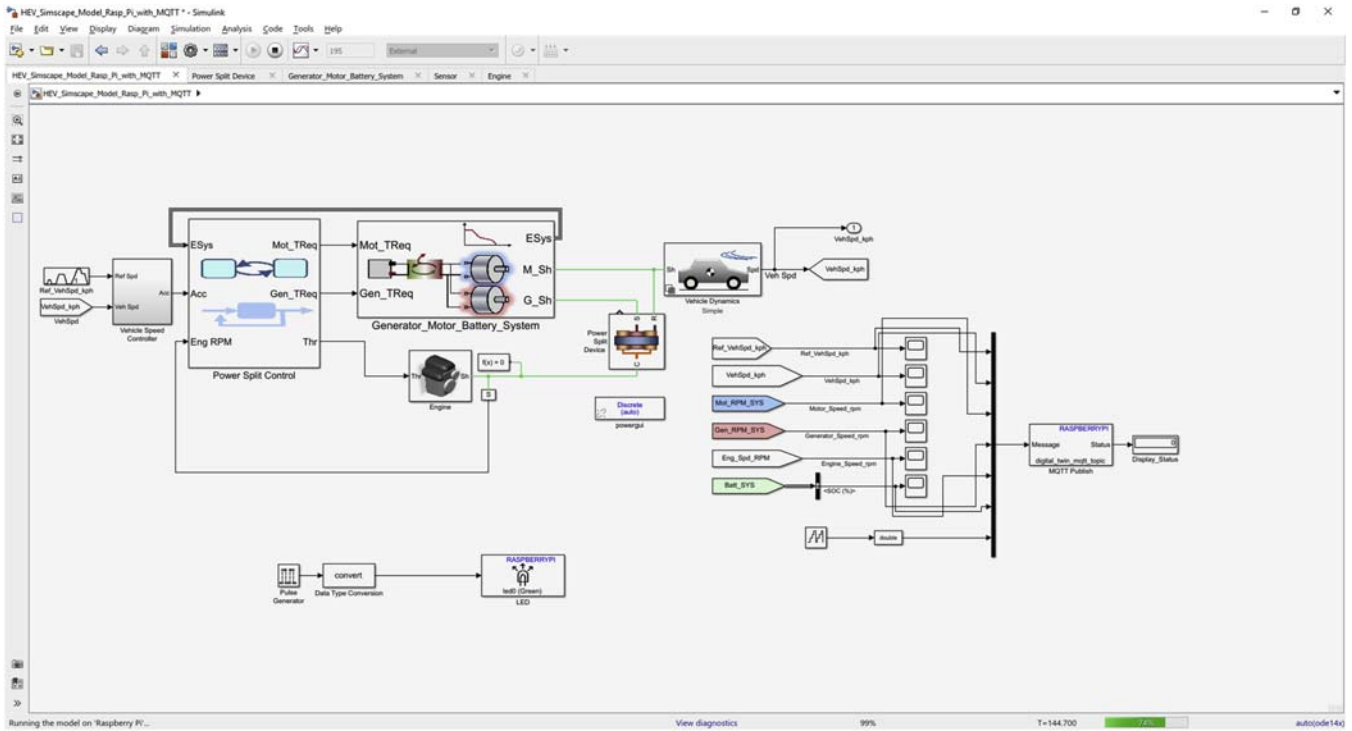


Figure 8.61 HEV Simulink model runs in Raspberry Pi external mode in real time.

Figure 8.62 MQTT message from the HEV model is received by Python code and published to AWS IoT Thing.

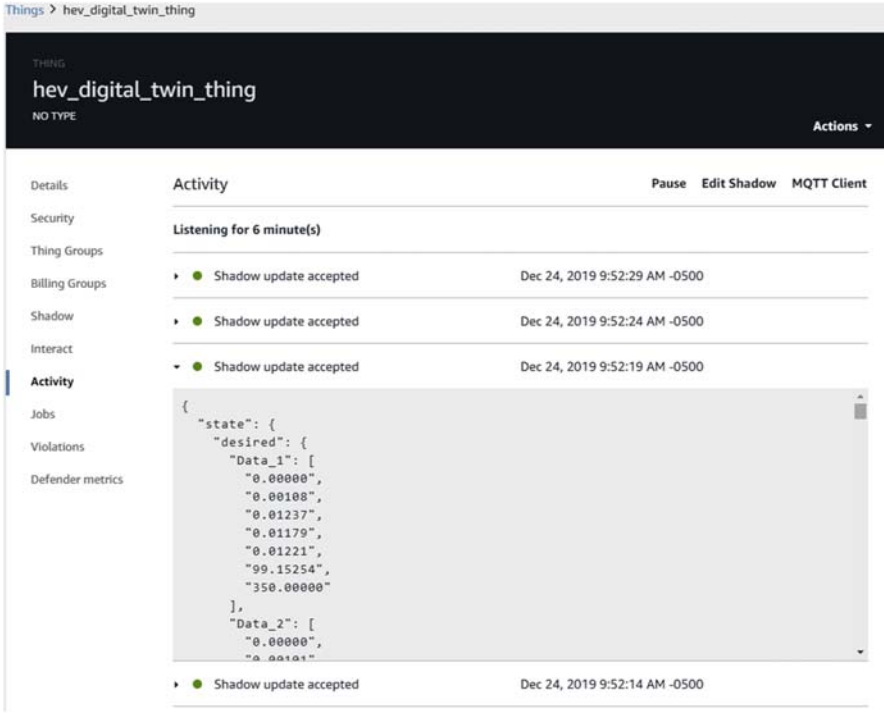


Figure 8.63 AWS IoT Thing receiving the HEV state information from Raspberry Pi hardware.

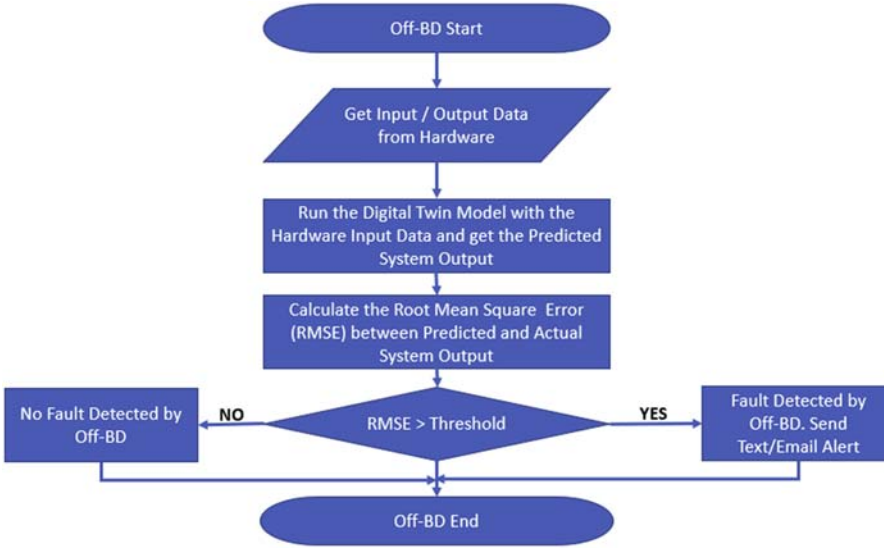


Figure 8.64 Off-BD algorithm flow chart.

between the actual and predicted signals, and compare the RMSE value with a threshold to detect if there is a failure. The E-mail/Text notification will not be tested in this section but will be covered later when we deploy the Off-BD and Digital Twin to the AWS cloud.

We will be running the Off-BD algorithm using Python programming language. One of the main reason why Python is chosen is that we eventually have to deploy this Off-BD algorithm to AWS cloud. We will be using **AWS Lambda** services for running the Off-BD algorithm on the cloud and **AWS Lambda** supports Python. Note that the Digital Twin model we have is developed using MATLAB®, Simulink®, and Simscape™, so in order to call the Digital Twin model from Python, we will have to generate "C" code from the Simscape™ model and compile it into an executable and then call the executable from Python program. One other thing to note is that in the back end the Amazon Web Services (AWS) is running on Linux Operating System. So the compiled executables that we have to make for the Digital Twin Model should also be in Linux so that the executable can be deployed and ran from AWS Linux machines. So we will use Linux operating system just for this section to generate code and compile an executable for the Digital Twin.

Follow the below steps to develop the Off-BD algorithm and test it locally on a machine:

1. The first step is to install Linux on a Machine. The Authors have tested the process with Ubuntu Linux. The installation process for Ubuntu Linux will not be covered in this chapter; there are plenty of online resources available for Linux installation. Authors recommend to install Ubuntu as a Dual-Boot setup if the User already has Windows installed on the machine. Dual-Boot installation allows Users to switch between Windows and Linux when required by just restarting one operating system and selecting others from the boot up menu. One of the working link to install Ubuntu at the time of writing of this chapter is here https://itsfoss.com/install-ubuntu-1404-dual-boot-mode-windows-8-81-uefi/.
2. The rest of the steps in this section is using Linux. So make sure Linux is installed and setup. We will also need to install MATLAB® on Linux to generate code and compile the Simscape™ Digital Twin Model. Once again, since it is outside the scope of the book, we will not go through the MATLAB® installation process in Linux. An instruction link to install MATLAB® on Linux that worked at the time of writing of this chapter is given here https://www.cmu.edu/computing/software/all/matlab/matlabinstall-linux.html.
3. Open MATLAB® in Linux and create a new folder and copy the HEV Model Simscape™ model used earlier and its initialization MAT file as shown in Fig. 8.65. The new folder is ***Digital_Twin_HEV_Model*** created under Chapter_8 folder.
4. Open the Simscape™ HEV model and make sure the System Target File under the **Model >> Configuration Parameters >> Code Generation >> System Target File** is set to "***ert.tlc***." On the model, add input port for the Reference Vehicle Speed and output ports for Actual Vehicle Speed, Motor, Generator and Engine Speed, and Battery SoC, etc., as shown in Fig. 8.66. Then click on the "***Build***" button on the Simulink Model Menu as highlighted in Fig. 8.66. This will now update the model, generate "***C***" source code from the model, create a Makefile to compile the generated code, compile the code, and create an executable application from the Model logic. MATLAB® will show the progress of this process as shown in Fig. 8.67. Since we are trying this on a Linux Machine, it will create a Linux executable application. The "***C***" code will be generated into a folder named **Model_Name_ert_rtw** under the working folder and the name of the executable will be same as

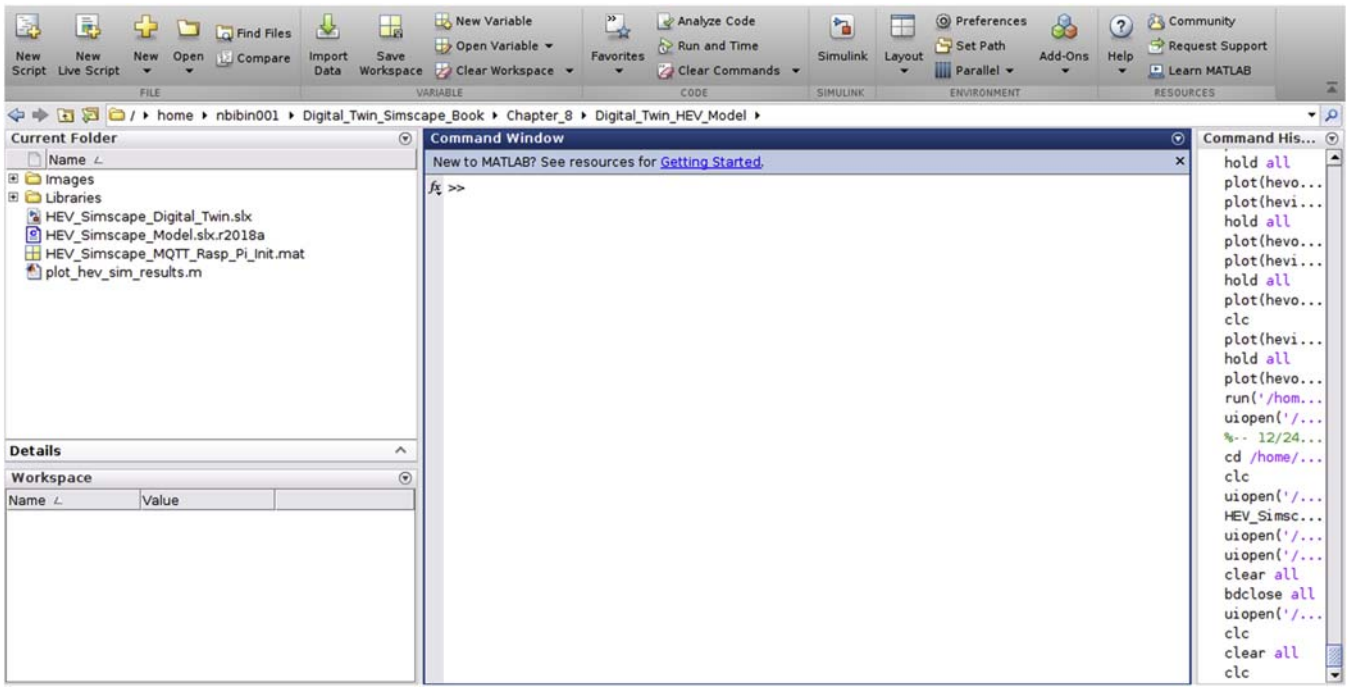


Figure 8.65 Copy Simscape™ HEV model and initialization M file to Linux MATLAB folder.

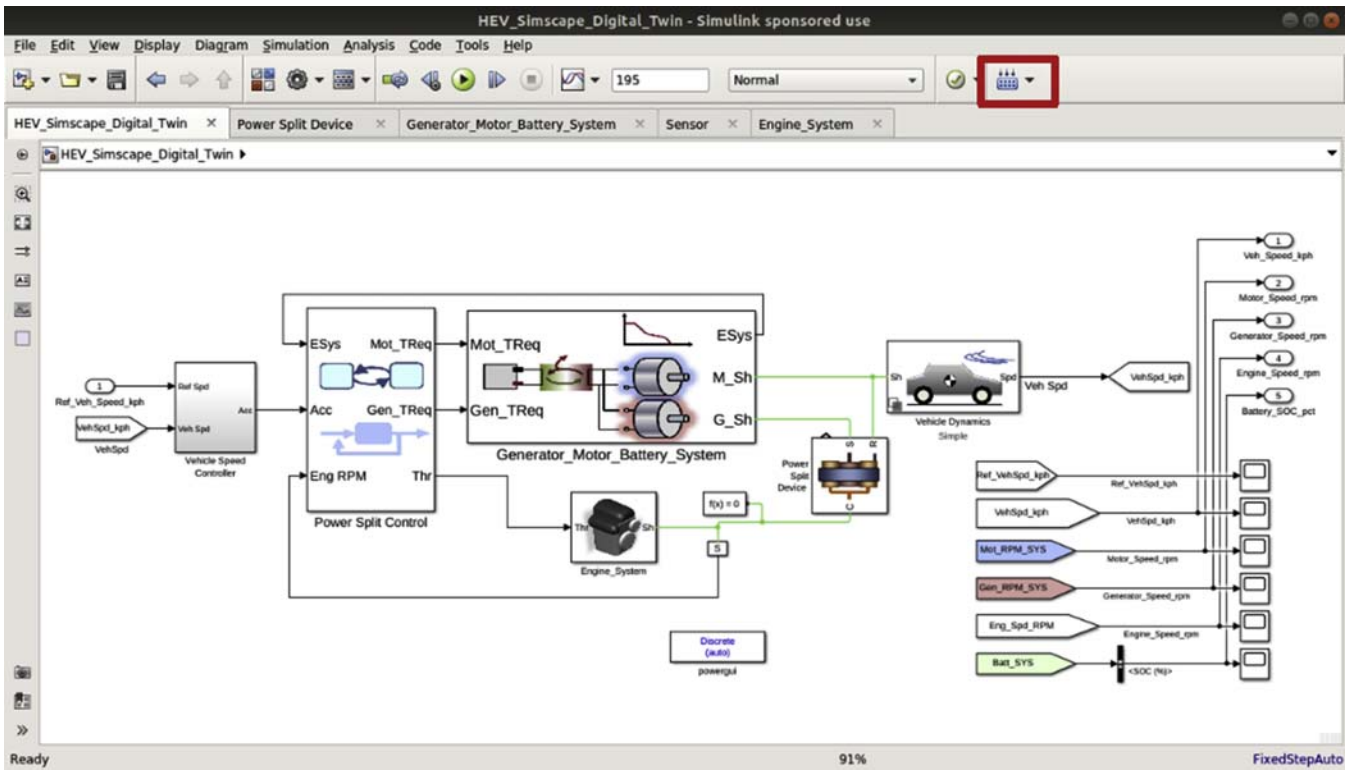


Figure 8.66 HEV model configured for Digital Twin codegen and compiling.

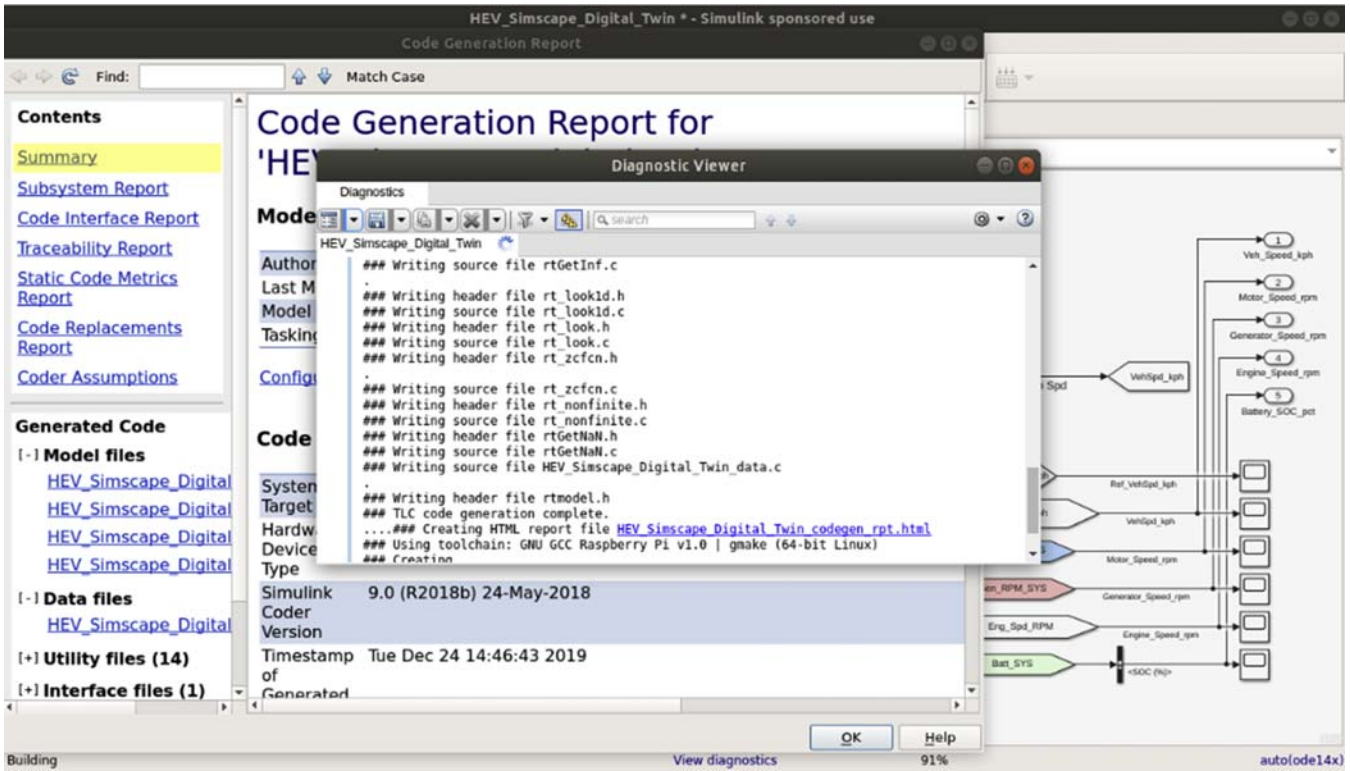


Figure 8.67 Model codegen and build progress.

that of the model name. In this case, our Model Name is **HEV_Simscape_Digital_Twin.slx**, so the generated code will be placed in **HEV_Simscape_Digital_Twin_ert_rtw** and the generated executable application name will be **HEV_Simscape_Digital_Twin**. Fig. 8.68 shows the code generated folder and the executable application in the MATLAB® working folder.

5. Go inside the codegen folder **HEV_Simscape_Digital_Twin_ert_rtw** and open the file ert_main.c as shown in Fig. 8.69 in an editor. Just double click on the file to open it in MATLAB® editor itself. As the User might know, every "***C***" code application requires an entry function named **main()**, which is the entry point function when the application starts to run. The **main()** function is generated by MATLAB® Code Generator into this **ert_main.c** file.
6. The ert_main.c file has two functions mainly in it. One is the **main()** and the other is the **rt_OneStep()** function as shown in Figs. 8.70 and 8.71. As mentioned above, **main()** is the entry point function, and this function initializes the model states if any using the **Model_Name_initialize** function and just enters and stays into a **while loop.** So we can see that even when the application runs, it is not really running our model logic yet in this framework as is now. So we will edit the **main()** function to suit our purpose in later steps mentioned below. The **rt_OneStep()** function calls another function **Model_Name_step** (**HEV_Simscape_Digital_Twin_step()** in this case; this step function contains the actual logic that we have implemented in the Simulink® model. We can see the **Model_step()** function uses global data structures to receive input and return the output. When MATLAB® generates code, the **rt_OneStep()** function call is commented out from the **main()** function, this is mainly because the Users can integrate the **Model_step()** function with their own operating system, embedded target software, or scheduler, etc., and doesn't really have to use the **ert_main.c** in their application software. The way we are going to run this application is we get the input data, we run the application, which will run the **Model_step()** function with one sample of input data, get its output, and then we run the application with the next sample data again.
7. Now we will start editing the ert_main.c to suit our application purpose. We will change the main() function to read an input file "**hev_input.csv**" from a Linux user temporary folder **/tmp**. The **/tmp** folder is selected for the input and output files because**/tmp** is the only folder which is writable from AWS. The **hev_input.csv** file will be automatically created into the **/tmp** folder when we receive data from Hardware system, we will discuss that later when we deploy the Digital Twin Model and Off-BD algorithm into AWS cloud. This input file will contain the actual data collected from Raspberry Pi hardware when it runs the HEV model, one line for each sample of data at every 0.1 s, for 5 s. After the first 5 s, there will be total 50 entries in this file. The data for the next 5 s will be appended to this hev_input.csv file, so after 10 s, there will be 100 entries in the file and so on. For the HEV Digital Twin model testing purpose, we ran the HEV desktop simulation model and save the output workspace into a Mat file and generate the input file hev_input.csv file with the data from simulation output.
8. First in the **ert_main.c**, a new handwritten C function **Parse_CSV_Line()** is added as shown in Fig. 8.72 to read the lines in the **hev_input.csv** file and splits the values between the delimiter comma in each lines. Add this function above the **main()** function. So essentially, this splits and returns the individual signals for Reference Vehicle Speed, Actual Vehicle Speed, Motor, generator Engine Speed, Battery SoC, and the count variable.
9. Next is to add a portion of code to open the **hev_input.csv** file from the path**/tmp/hev_input.csv** and read the actual Reference Vehicle Speed into an array Ref_Veh_Speed_kph [].This code needs to be added inside the main() function as shown in Fig. 8.73.

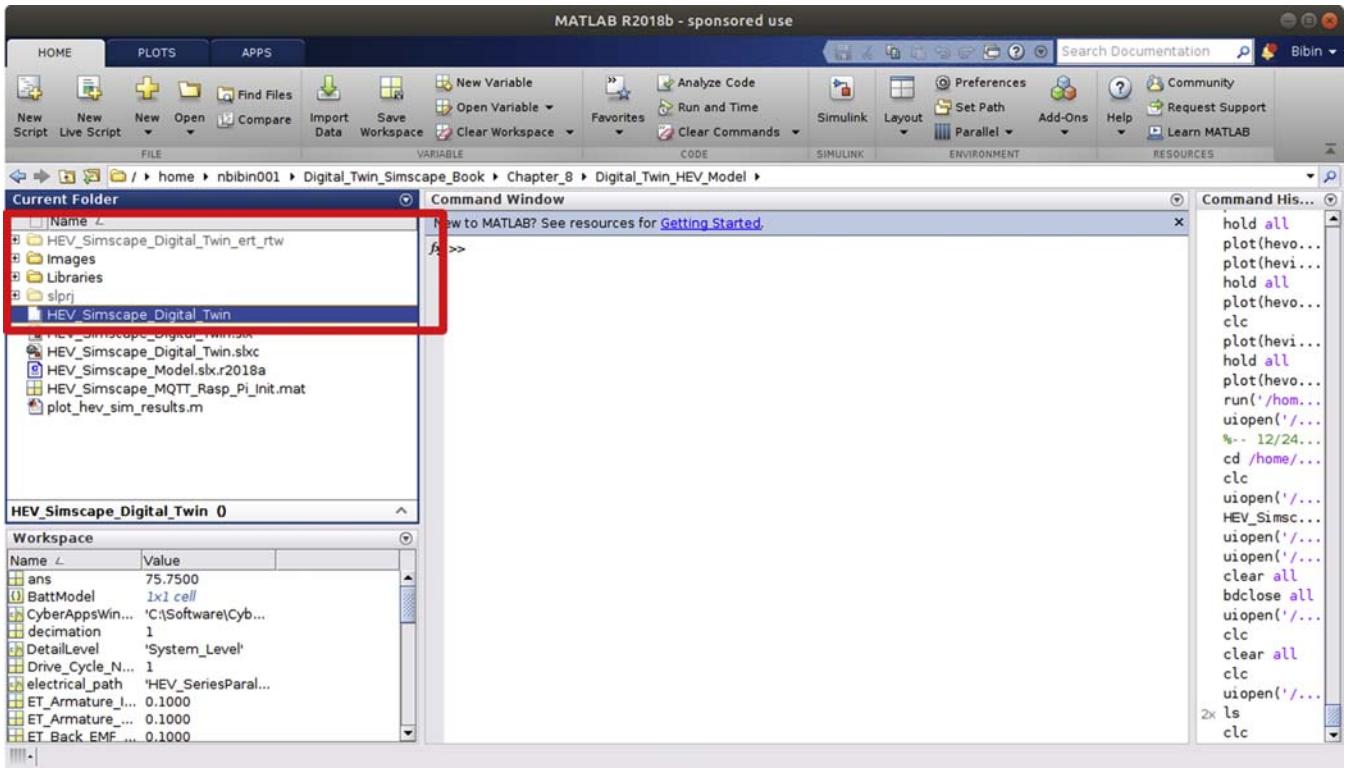


Figure 8.68 Generated code folder and executable application.

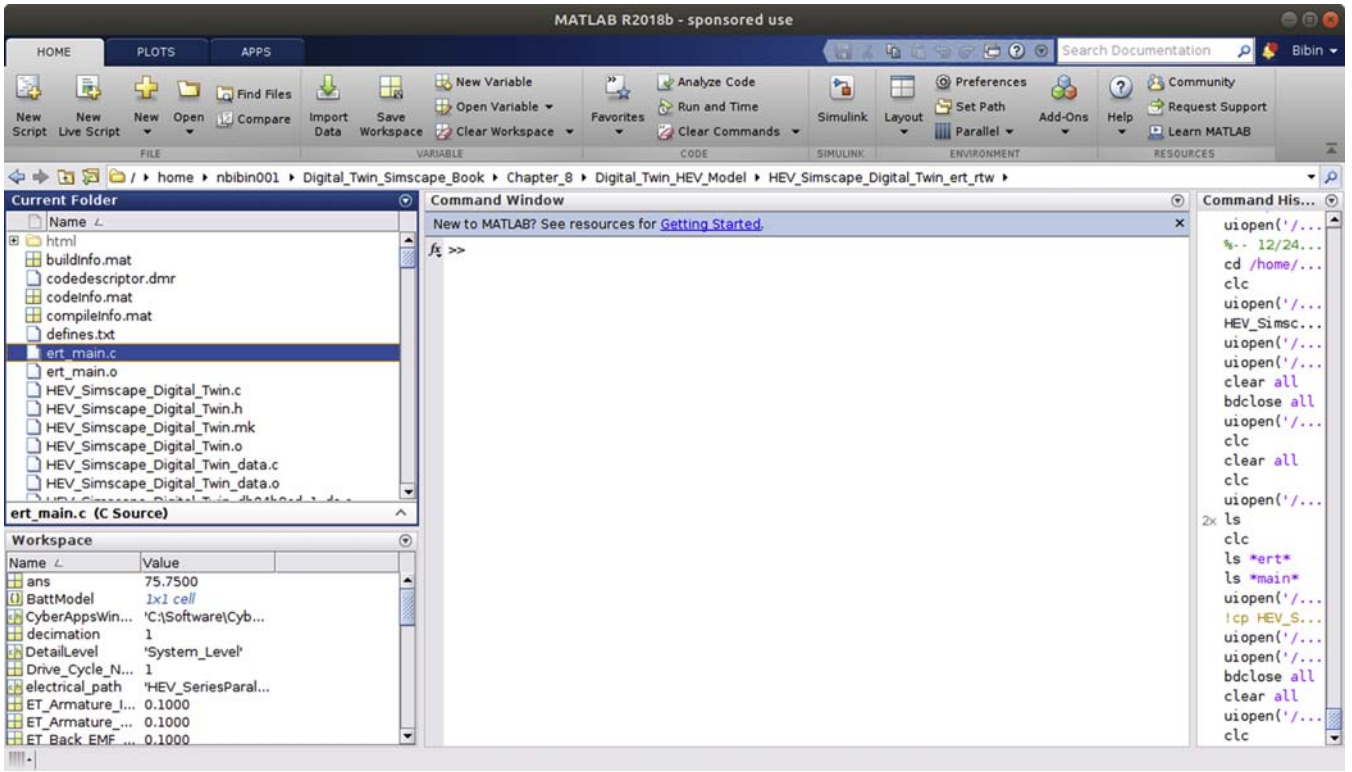


Figure 8.69 Selecting ert_main.c file from codegen folder.

```
 */
int_T main(int_T argc, const char *argv[])
{
  /* Unused arguments */
  (void)(argc);
  (void)(argv);

  /* Initialize model */
  HEV_Simscape_Digital_Twin_initialize();

  /* Simulating the model step behavior (in non real-time) to
   *  simulate model behavior at stop time.
   */
  while ((rtmGetErrorStatus(HEV_Simscape_Digital_Twin_M) == (NULL)) &&
         !rtmGetStopRequested(HEV_Simscape_Digital_Twin_M)) {
    rt_OneStep();
  }

  /* Disable rt_OneStep() here */

  /* Terminate model */
  HEV_Simscape_Digital_Twin_terminate();
  return 0;
}

/*
 * File trailer for generated code.
 *
 * [EOF]
```

Figure 8.70 main() function in ert_main.c.

```
void rt_OneStep(void)
{
  static boolean_T OverrunFlag = false;

  /* Disable interrupts here */

  /* Check for overrun */
  if (OverrunFlag) {
    rtmSetErrorStatus(HEV_Simscape_Digital_Twin_M, "Overrun");
    return;
  }

  OverrunFlag = true;

  /* Save FPU context here (if necessary) */
  /* Re-enable timer or interrupt here */
  /* Set model inputs here */

  /* Step the model for base rate */
  HEV_Simscape_Digital_Twin_step();

  /* Get model outputs here */

  /* Indicate task complete */
  OverrunFlag = false;

  /* Disable interrupts here */
  /* Restore FPU context here (if necessary) */
  /* Enable interrupts here */
}
```

Figure 8.71 rt_OneStep() function in ert_main.c.

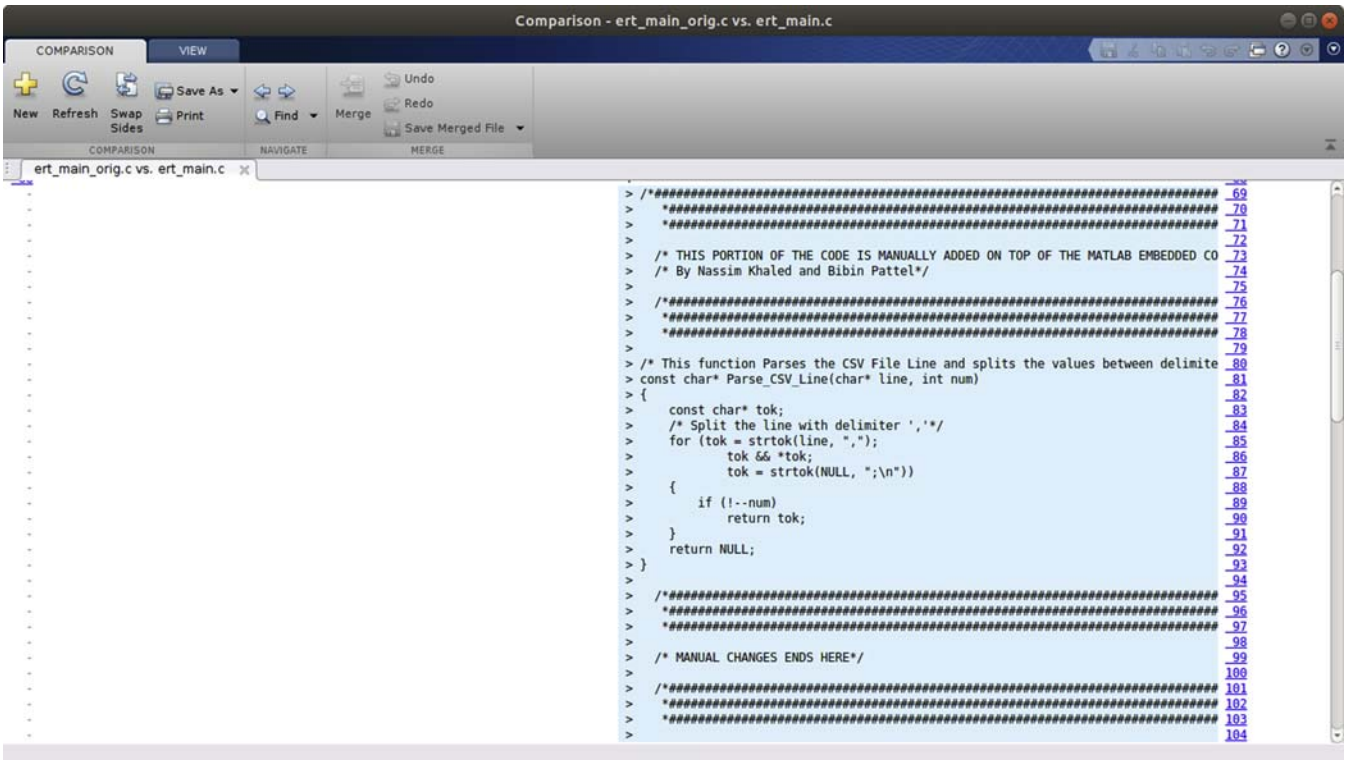


Figure 8.72 Function to parse CSV file lines and get the input file hev_input.csv

10. Next is to take out the infinite **while()** loop function from **main()** function of the **ert_main.c**. The **Model_step()** function will be called explicitly, so we don't need this **while()** loop. Also a loop is ran for the number of lines in the input file to call the step function **HEV_-Simscape_Digital_Twin_step()**. The Input array values are assigned to the global input data structure **HEV_Simscape_Digital_Twin_U.Ref_Veh_Speed_kph**. The outputs will be stored into the global data structure **HEV_Simscape_Digital_Twin_Y**. These structure definitions can be found in the **HEV_Simscape_Digital_Twin.c** file. An output file hev_output.csv will be created in the /tmp folder with the input and outputs collected when running the Digital Twin model. See Fig. 8.74.
11. The changes to the ert_main.c file is completed now. Save and close the file. From the **HEV_Simscape_Digital_Twin_ert_rtw** folder, look for the Makefile with the extension "**HEV_Simscape_Digital_Twin.mk**." Copy this file to a new file named Makefile using the Linux command ***!cp HEV_Simscape_Digital_Twin.mk Makefile*** from MATLAB command prompt. We can try the same from a Linux command window as well without the "**!**" at the beginning. "**!**" tells the MATLAB command window that what follows is an operating system command. Since we have changed the ert_main.c file, let us just delete the previously created object file ert_mian.o using the rm command ***!rm ert_main.o***. Now rebuild the executable application using the command **!make -f Makefile**. Since only the ert_-main.c file is changed and all the other files are same, the **make** command only recompiled **ert_mian.c**. All of the commands tried in this step and their output are shown in Fig. 8.75. The recompiled executable application **HEV_Simscape_Digital_Twin** is shown in Fig. 8.76.
12. Let us just make a quick MATLAB®-based program to test the executable application. The MATLAB® program will take the HEV desktop simulation data collected by running the simulation on the host computer, as if it is coming from the actual hardware in real time, crate the "**hev_input.csv**" file copy it to **/tmp** folder, call the executable application which will run with the input Reference Vehicle Speed data from the **hev_input.csv**, and create the "**hev_output.csv**" file under **/tmp** folder. The MATLAB® program will then look at the "**hev_output.csv**" file and get the predicted signal values, and to calculate RMSE between the actual and predicted data, make plots for the each of the signals, plot the error between them, and also calculate and report the RMSE between the actual and predicted values. See the MATLAB® program below. The Mat file HEV_Simscape_Model_Sim_Results.mat is created by saving the MATLAB workspace after running the model simulation in the host computer. These data are used to create the **"hev_input.csv"** file.

```
% Book Title: Digital Twin Developent and Deployment On Cloud Using Matlab
% Simscape
% Chapter 8
% Authors: Nassim Khaled and Bibin Pattel
% Last Modified 12/14/2019
%%
% Create input file from the simulation data
% Load the Simulation Data
clc
clear all
load HEV_Simscape_Model_Sim_Results.mat
```

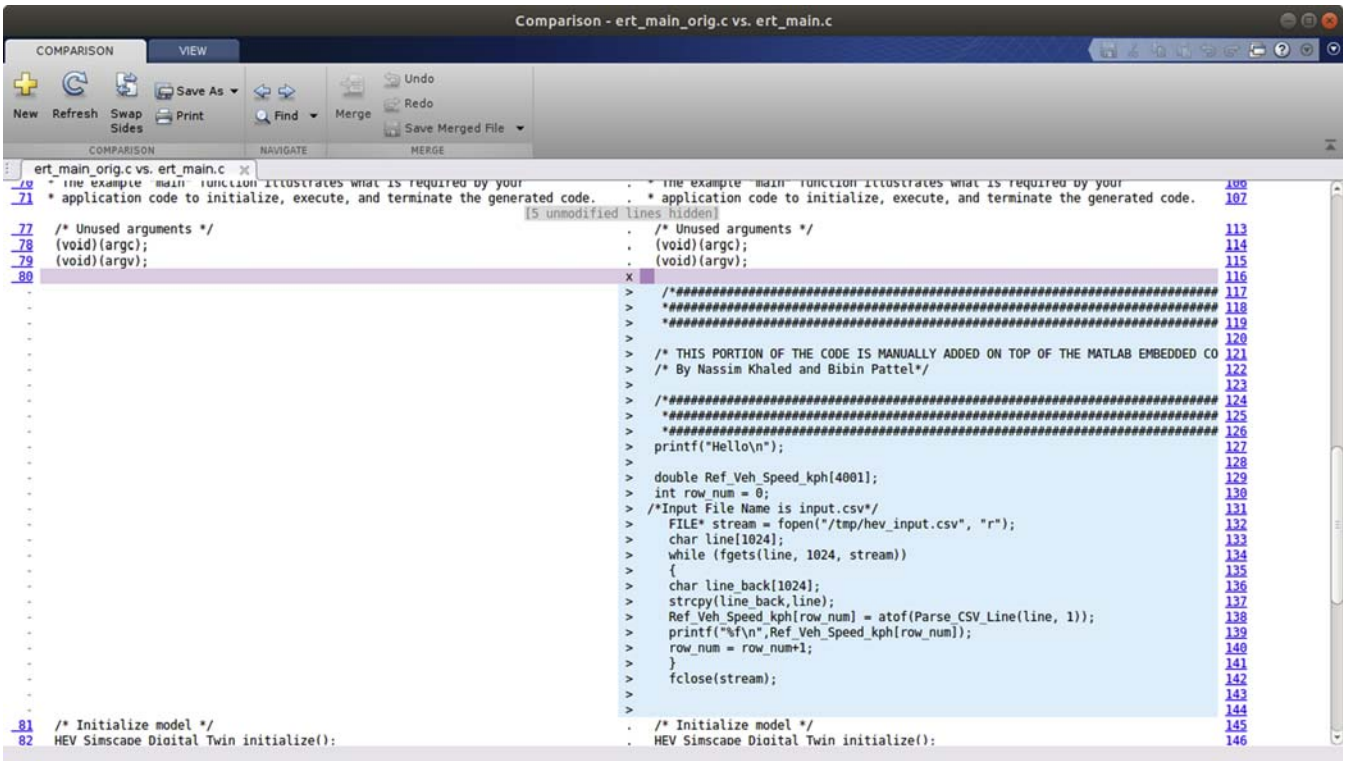


Figure 8.73 Reading input.csv and storing into Input array.

```
 while ((rtmGetErrorStatus(HEV_Simscape_Digital_Twin_M) == (NULL)) &&
       !rtmGetStopRequested(HEV_Simscape_Digital_Twin_M)) {
  rt_OneStep();
}

/* Disable rt_OneStep() here */
```

```
 int_T ii;
 FILE * output_fp;
   output_fp = fopen ("/tmp/hev_output.csv", "w+");
 for (ii= 0;ii<row_num;ii++)
 {
     HEV_Simscape_Digital_Twin_U.Ref_Veh_Speed_kph = Ref_Veh_Speed_kph[ii];
     HEV_Simscape_Digital_Twin_step();
     fprintf(output_fp, "%f,%f,%f,%f,%f,%f,%f\n",HEV_Simscape_Digital_Twin_U.Re
 }
/*Close output.csv file pointer*/
   fclose(output_fp);

 /* Disable rt_OneStep() here */
```

Figure 8.74 Removing while() loop from ert_mian.c and calling the model step function.

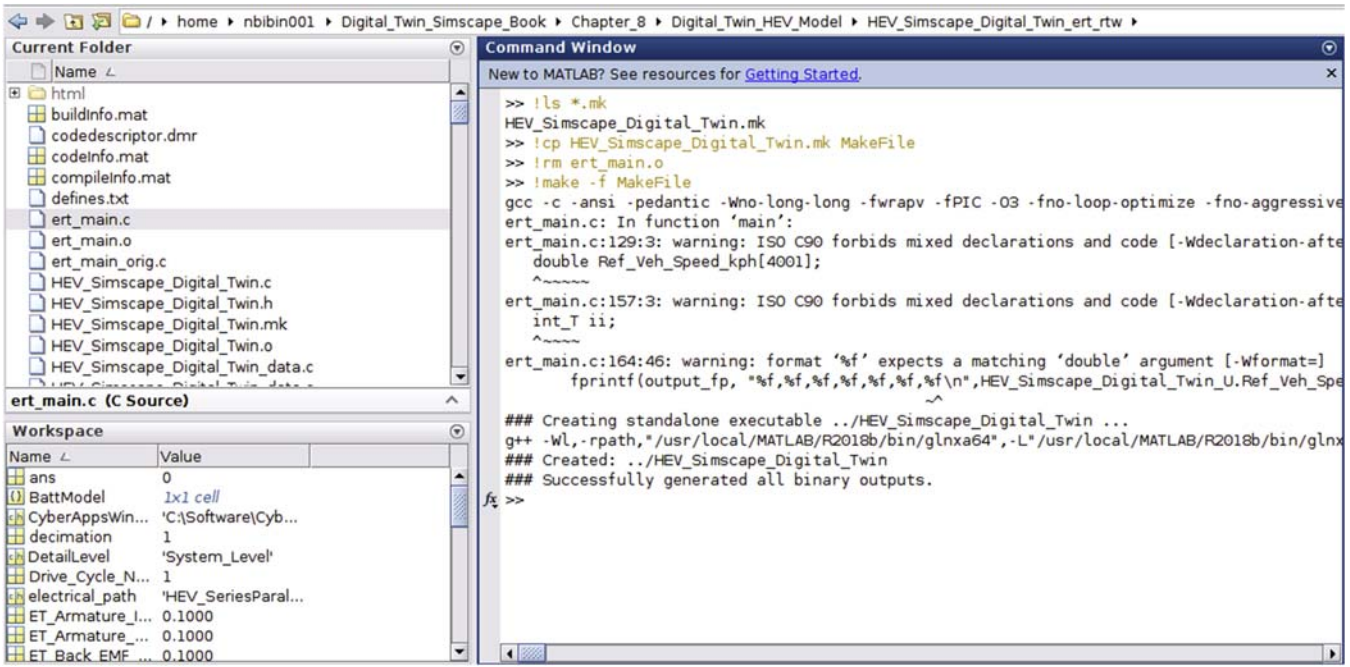


Figure 8.75 Recompiling the application with the updated ert_mian.c.

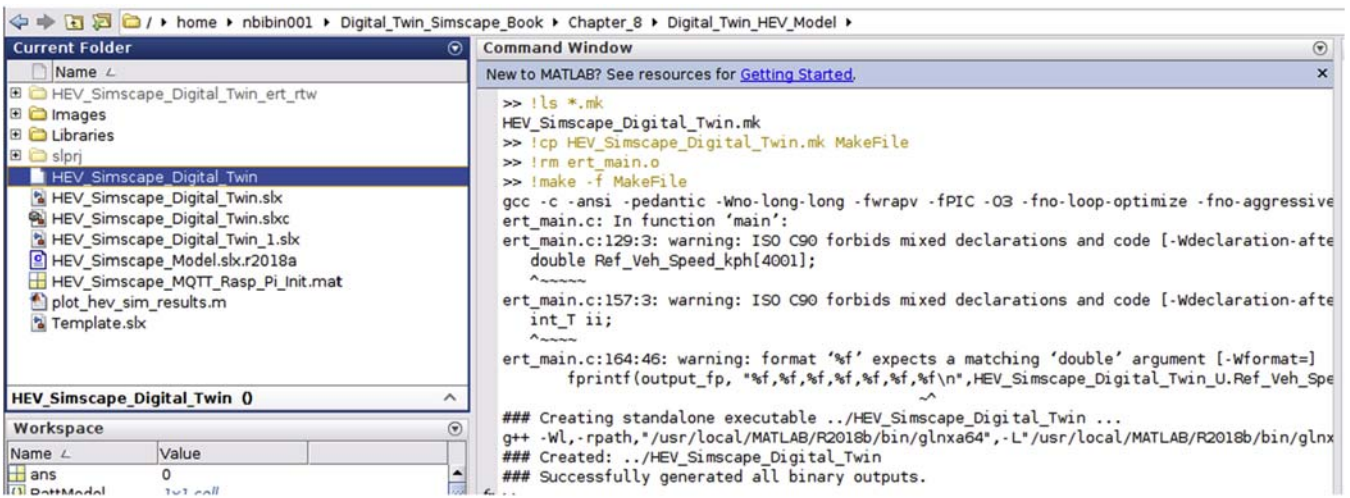
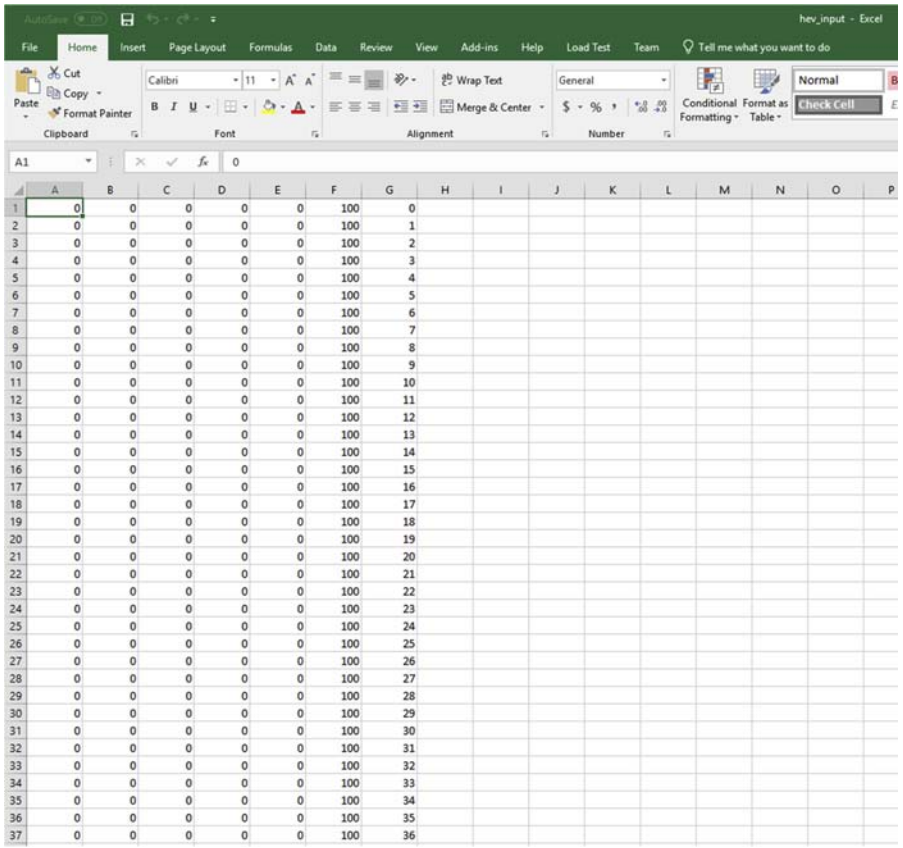


Figure 8.76 Recompiled executable Application.

```
fid = fopen('hev_input.csv','w+');
for i =1:1951
fprintf(fid,'%s,%s,%s,%s,%s,%s,%s\n',
num2str(Ref_VehSpd_kph(i,2)),num2str(VehSpd_kph(i,2)),
        num2str(Motor_Speed_rpm(i,2)),
num2str(Generator_Speed_rpm(i,2)),num2str(Engine_Speed_rpm(i,2)),
        num2str(Battery_SOC_pct(i,2)),num2str(i-1));
end
fclose(fid);
% Copy the hev_input.csv file to the /tmp folder
!cp hev_input.csv /tmp
pause(1)
% Run the Digital Twin HEV Compiled Model
!./HEV_Simscape_Digital_Twin
pause(1)
% Digital Twin Model creates the output file hev_output.csv.Copy that
file
% to the current folder
!cp /tmp/hev_output.csv .
pause(1);
%%
actual_data = csvread('hev_input.csv');
model_predicted_data = csvread('hev_output.csv');
% Loop through to plot and calculate the RMSE
% i = 1 for Reference Vehicle Speed
% i = 2 for Actual Vehicle Speed
% i = 3 for Motorator Speed
% i = 5 for Engir Speed
% i = 4 for Genene Speed
% i = 6 for Battery SoC
title1 = {'Reference Vehicle Speed','Actual Vehicle Speed','Motor
Speed','Generator Speed','Engine Speed','Battery SoC'};
title2 = {'Reference Vehicle Speed Error','Actual Vehicle Speed
Error','Motor Speed Error','Generator Speed Error','Engine Speed
Error','Battery SoC Error'};
for i =1:6
  % Plot the individual signals Actual Vs Predicted
    figure(1);
    subplot(320 +i)
    plot(actual_data(:,i),'linewidth',2);
    hold all
    plot(model_predicted_data(:,i),'--','linewidth',2);
    title(title1{i},'Fontsize',18);
```

```
    % Plot the Corresponding Signal Errors Actual - Predicted
      figure(2);
      Error = actual_data(:,i) - model_predicted_data(:,i);
      subplot(320+i)
      plot(Error,'linewidth',2);
      title(title2{i},'Fontsize',18);
    % Calculate the Root Mean Squared for Error for Each Signal
      Squared_Error = Error.^2;
      Mean_Squared_Error = mean(Squared_Error);
      Root_Mean_Squared_Error(i) = sqrt(Mean_Squared_Error);
end
figure(1)
legend('Actual Data','Predicted Data');
figure(2)
legend('Error Between Actual Data and Predicted Data');
Root_Mean_Squared_Error
```

13. Fig. 8.77 shows the "hev_input.csv" file created from the MATLAB® test script above. First column is the Reference Vehicle Speed, second column is the Actual Vehicle Speed, third, fourth, and fifth columns are the Motor, Generator, and Engine Speeds, respectively, sixth column is the Battery SoC, and last column contains a count variable.
14. Figs. 8.78 and 8.79 show the plots generated from the MATLAB® test script comparing the actual versus Digital Twin predicted signals of HEV, and also the calculated error. Fig. 8.80 shows the RMSE values between actual and Digital Twin predicted values. It can be seen that though the actual and Digital Twin predicted signals are almost matching everywhere, but for the Generator Speed and Engine Speed some deviations are noted, which caused the RMSE for those to signals to be higher than expected. Ideally, we expect the RMSE value to be close to zero because we are running the same model with same data. Authors could not really explain why the mismatch occurs in few points.
15. Now let us repeat the Step 12 with a desktop simulation model in which the Throttle going to the Engine subsystem is connected through a Gain block of value 0, which essentially makes the Engine system see a throttle value of 0 always. This is the way we introduced throttle failure conditions in this example. There could be many different ways though. See Fig. 8.81 for the introduced Throttle Failure. After running the desktop host computer simulation, save the Mat file and load it in the above MATLAB program and run it.
16. Now when the MATLAB code runs, the input Reference Speed is same for both the desktop simulation model and the Digital Twin model, but for the desktop simulation model, the throttle to the Engine system is made 0, whereas Digital Twin model is not aware of this, so it will be make a prediction as if there is no throttle fault, and we will compare the actual and predicted signals and calculate the RMSE of the signals. Check, Figs. 8.82–8.84 for the results. It can be noted that even with the Throttle failure, the vehicle speed reference is still met with the power from the Battery and Motor. The Engine, Generator, and the Battery SoC signals are now a lot different than the expected values, which is reflected in the RMSE value calculations as well. So by comparing the RMSE values to a certain threshold, we can detect the Throttle failure condition.



	A	B	C	D	E	F	G
1	0	0	0	0	0	100	0
2	0	0	0	0	0	100	1
3	0	0	0	0	0	100	2
4	0	0	0	0	0	100	3
5	0	0	0	0	0	100	4
6	0	0	0	0	0	100	5
7	0	0	0	0	0	100	6
8	0	0	0	0	0	100	7
9	0	0	0	0	0	100	8
10	0	0	0	0	0	100	9
11	0	0	0	0	0	100	10
12	0	0	0	0	0	100	11
13	0	0	0	0	0	100	12
14	0	0	0	0	0	100	13
15	0	0	0	0	0	100	14
16	0	0	0	0	0	100	15
17	0	0	0	0	0	100	16
18	0	0	0	0	0	100	17
19	0	0	0	0	0	100	18
20	0	0	0	0	0	100	19
21	0	0	0	0	0	100	20
22	0	0	0	0	0	100	21
23	0	0	0	0	0	100	22
24	0	0	0	0	0	100	23
25	0	0	0	0	0	100	24
26	0	0	0	0	0	100	25
27	0	0	0	0	0	100	26
28	0	0	0	0	0	100	27
29	0	0	0	0	0	100	28
30	0	0	0	0	0	100	29
31	0	0	0	0	0	100	30
32	0	0	0	0	0	100	31
33	0	0	0	0	0	100	32
34	0	0	0	0	0	100	33
35	0	0	0	0	0	100	34
36	0	0	0	0	0	100	35
37	0	0	0	0	0	100	36

Figure 8.77 hev_input.csv file created from the MATLAB test script.

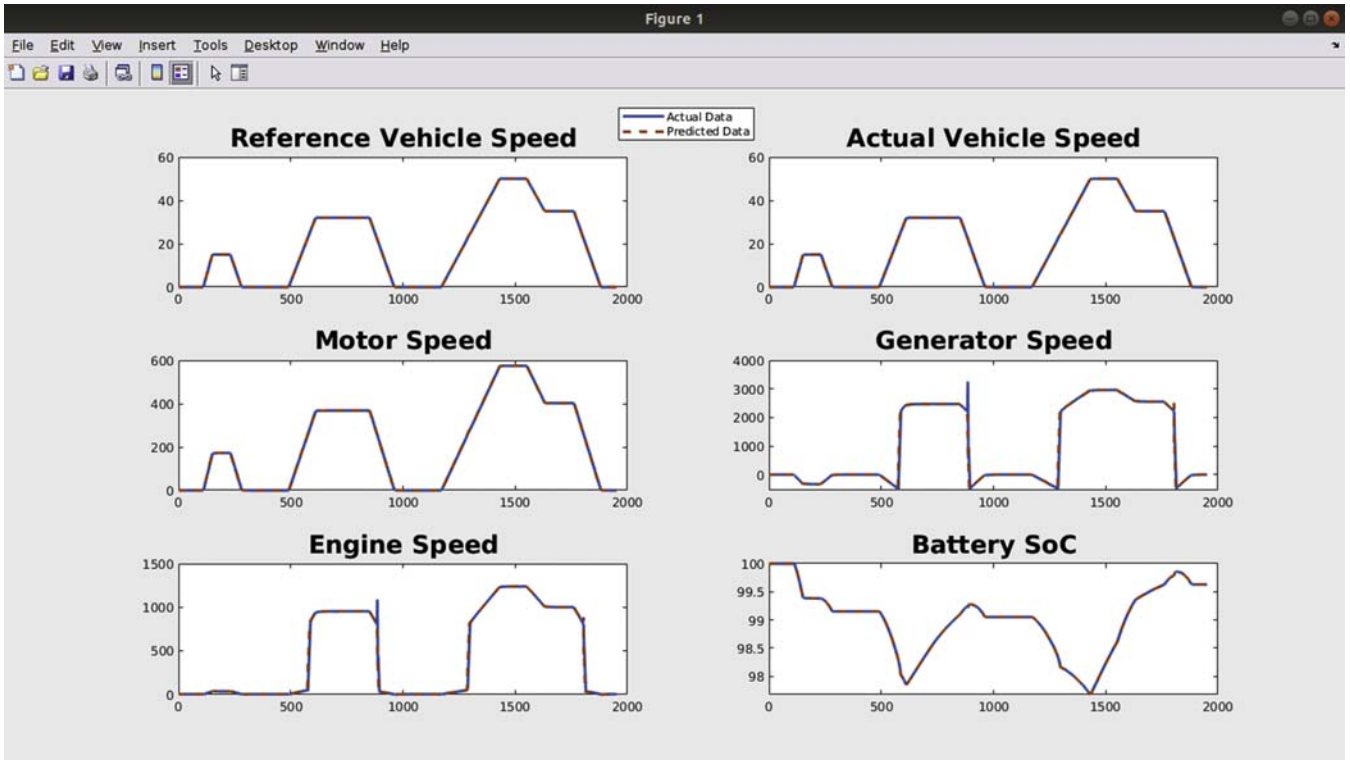


Figure 8.78 Comparing actual versus Digital Twin predicted values for HEV.

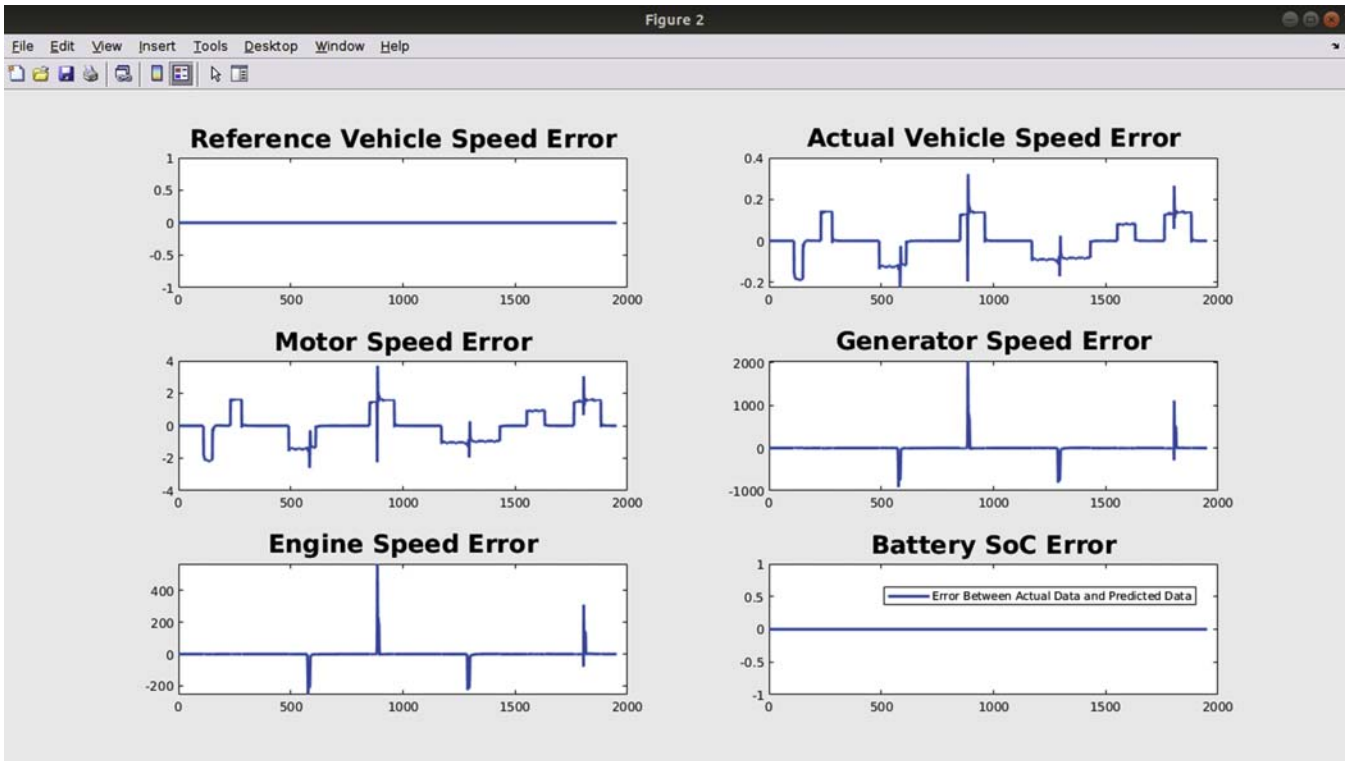


Figure 8.79 Error between actual and Digital Twin predicted values of HEV.

```
>> Root_Mean_Squared_Error

Root_Mean_Squared_Error =

         0    0.0749    0.8612  110.0032   30.6989    0.9114
```

Figure 8.80 RMSE values of the corresponding six signals.

8.9 Deploying digital twin Hybrid Electric Vehicle system model to cloud

In this section, we will deploy the compiled executable of the HEV Digital Twin model and the diagnostic algorithm to AWS cloud. We will be configuring and using three AWS in this section, Simple Notification Service (SNS), and AWS Lambda Functions and Amazon S3 Bucket. SNS is used to send Text/E-mail messages to the subscribers, and the subscriber details can be configured. We can create topics for SNS and add subscriber information such as cell number, E-mail ID. etc., so that the subscribers will be notified when the topic event is triggered. AWS Lambda is an event-driven serverless computing platform that runs a specific code, which we can develop and deploy. This code will run in response to a specific event, and in our case, the event is triggered by the AWS IoT Core when it receives data from the Raspberry Pi. The Amazon S3 Bucket is used as a means to store the data received from the Raspberry Pi Hardware. Because the HEV system simulation/running depends highly on the initial conditions or states, we need to reinitialize the model with the exact same states after every 5 s when we get the data from the hardware. This can be done, but adds a lot of complexity, so instead what we do is we just keep a history of all the data received from the hardware, and every time the Digital Twin model is ran with the entire input right from the beginning. So this approach takes away the state initialization problem. The very first initial condition is same for the hardware and the Digital Twin model, so for the same inputs the hardware and the Digital Twin model should behave the same if there are no fault conditions present. So we use Amazon S3 bucket to store the previous data, as the Lambda function doesn't have memory or there is no way to store data. Please check the AWS documentation for more details about these services. Fig. 8.85 shows the high level diagram, where the AWS IoT Core will be triggering a Lambda function written in Python, which will run the Digital Twin model, run the Off-BD algorithm, and make a diagnostic decision and trigger the SNS service to notify the subscribers about the status of Off-BD algorithm decision. The Lambda function interacts with Amazon S3 Bucket for storing and retrieving data for every run. We will first get started with the SNS configuration; please follow the below steps:

1. On the AWS Management Console, search for SNS and select the SNS as shown in Fig. 8.86.
2. Click on the ***Create Topic*** button. See Fig. 8.87.

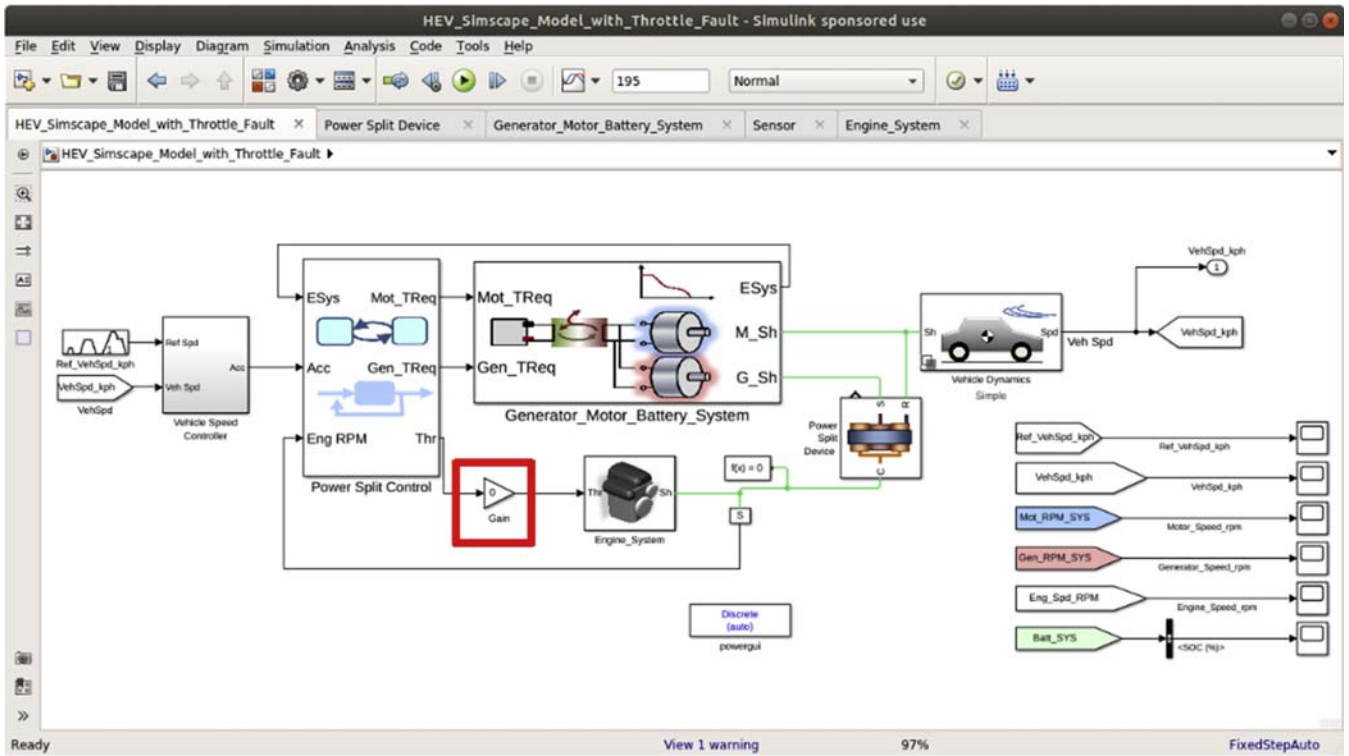


Figure 8.81 Host computer simulation model with throttle failure introduced.

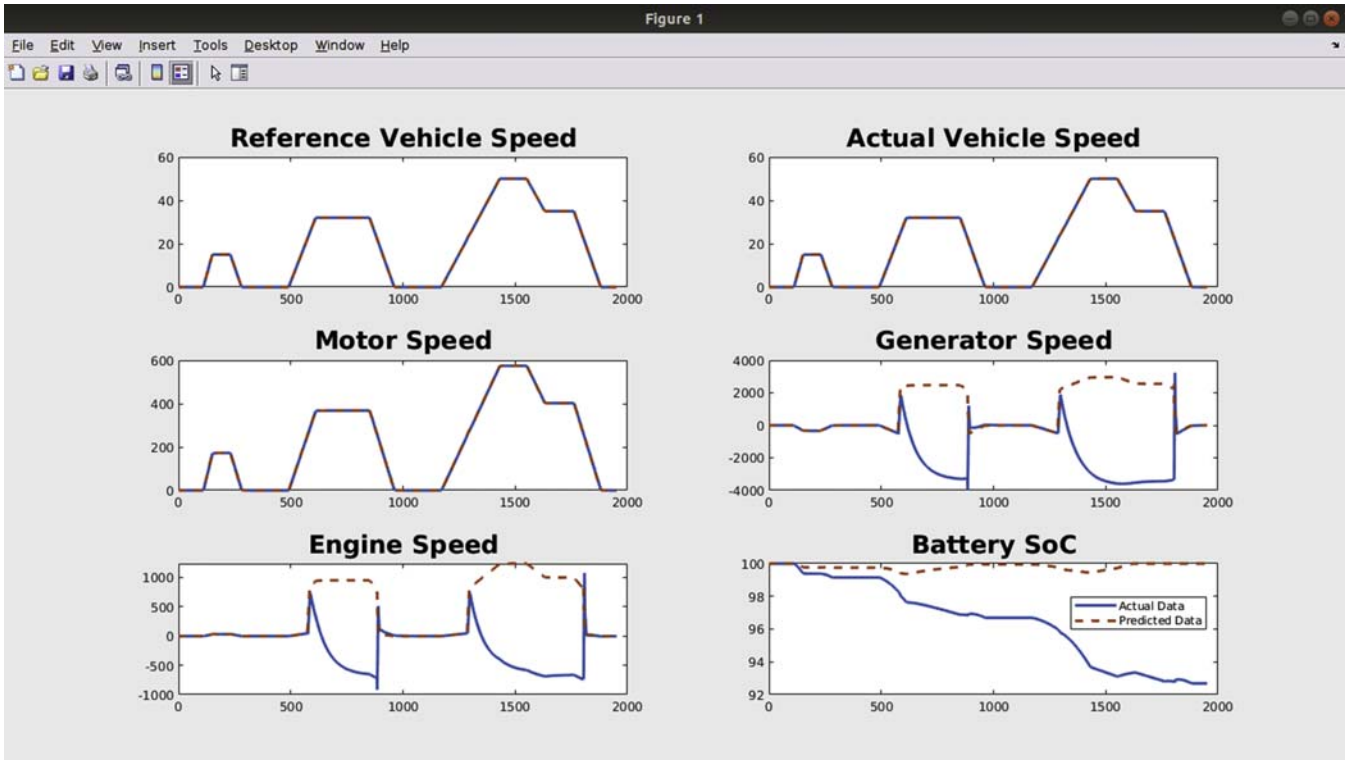


Figure 8.82 Comparing actual versus Digital Twin predicted values for HEV with throttle failure.

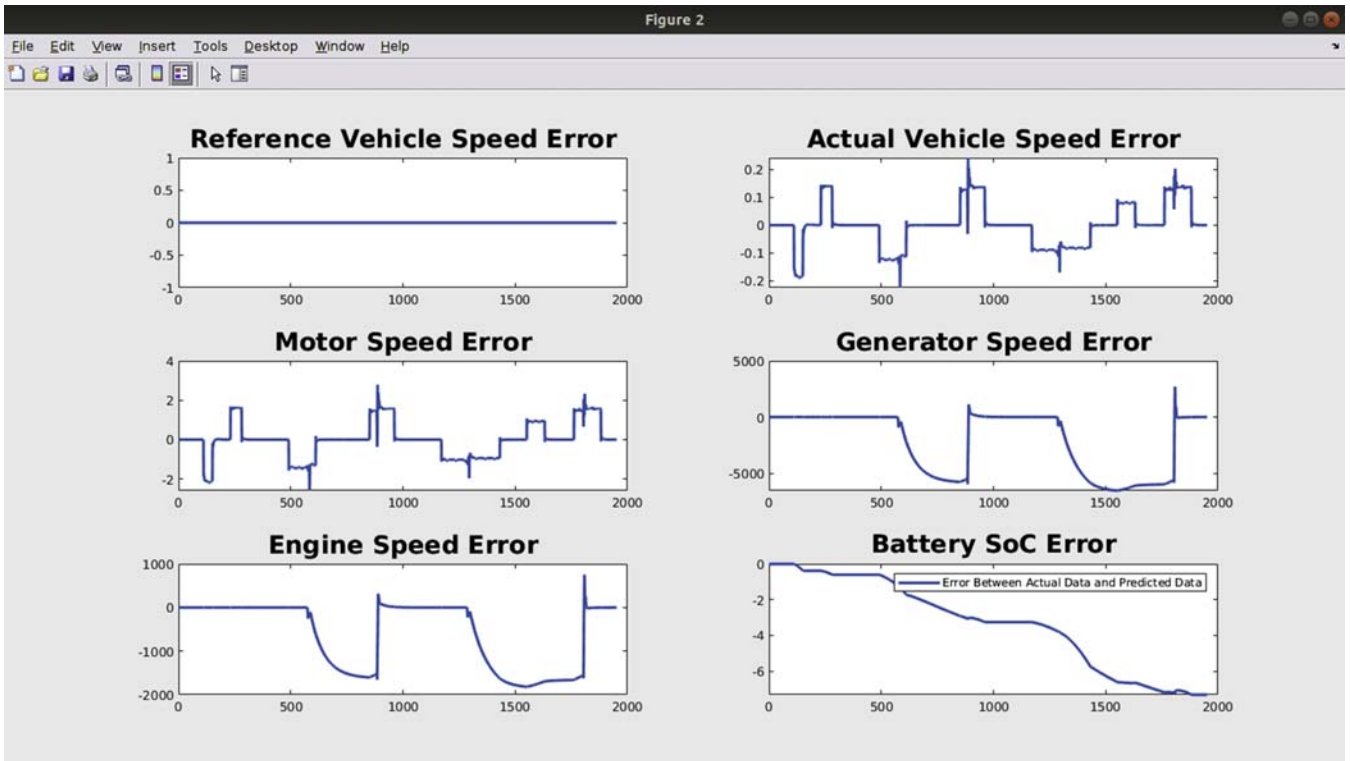


Figure 8.83 Error between actual and Digital Twin predicted values of HEV with throttle failure.

```
>> Root_Mean_Squared_Error

Root_Mean_Squared_Error =

   1.0e+03 *

         0    0.0001    0.0009    3.4804    0.9667    0.0041
```

Figure 8.84 RMSE values of the corresponding six signals with throttle failure.

3. Give a Name to the topic on the Create topic form. We used **digital_twin_topic** in this example. Also give a Display name, which will be displayed in the SMS and E-mail. Click on the **Create topic** button to finish creating the new SNS topic as shown in Fig. 8.88.
4. The newly created topic is shown in Fig. 8.89. Now we need to create a Text and E-mail subscriptions for this topic. Click on the Create subscription button as shown in Fig. 8.89. Note down the topic ARN here, we will be using it later in the Lambda function to interact with the SNS.
5. On the new ***Create subscription*** form, enter first the SMS protocol for Text subscription and enter the Cell number starting with + and Country Code and click the ***Create subscription*** button. For example **+1<10 Digit Cell Number>** for the United States. After the SMS subscription is created, use the same Create subscription button for creating the E-mail subscription from the Drop down menu and enter the E-mail ID to which we want to get the notification. Check Figs. 8.90–8.92 for details for creating Text and E-Mail subscriptions.
6. Though we created the subscriptions, it can be seen that under the **Subscriptions** menu the status shows Pending confirmation as shown in Fig. 8.93. When we created the subscriptions, AWS will automatically send an email to the email address given in the subscription with a link to confirm the subscription. Check the email and confirm the subscription by clicking on the link.
7. Once you click on the link in the E-mail, it will display the Subscription Confirmation message window. And under the Subscriptions menu in AWS SNS, we can see both the SMS and E-mail subscriptions show up with the Confirmed status. See Figs. 8.94 and 8.95. We can test these newly created topics are working by selecting the checkbox against the Topic ID and clicking the ***Publish message*** button highlighted in Fig. 8.95. Enter the Message body in the new form and publish it; based on the Topic ID we selected, it will send Text or E-mail accordingly.
8. Now let us configure the AWS Lambda Functions to run the Digital Twin Model and Off-BD algorithm and also notify the user using the SNS service configured in above steps. Search for lambda in the AWS Management Console and select the Lambda services. See Fig. 8.96.
9. On the AWS Lambda Console, click on the ***Create function*** button as shown in Fig. 8.97.
10. We will be developing and deploying the Lambda function in Python programming language. There are other options also available like Node js, etc., and depending upon the User's expertise, different language options can be selected in the ***Create function*** form. Give a name to the new Lambda function, we chose the name as hev_digital_twin and select the ***RunTime*** as **Python 3.7** and click on the ***Create function*** button as shown in Figs. 8.98 and 8.99.
11. AWS will create a new default template function as shown in Fig. 9.174.

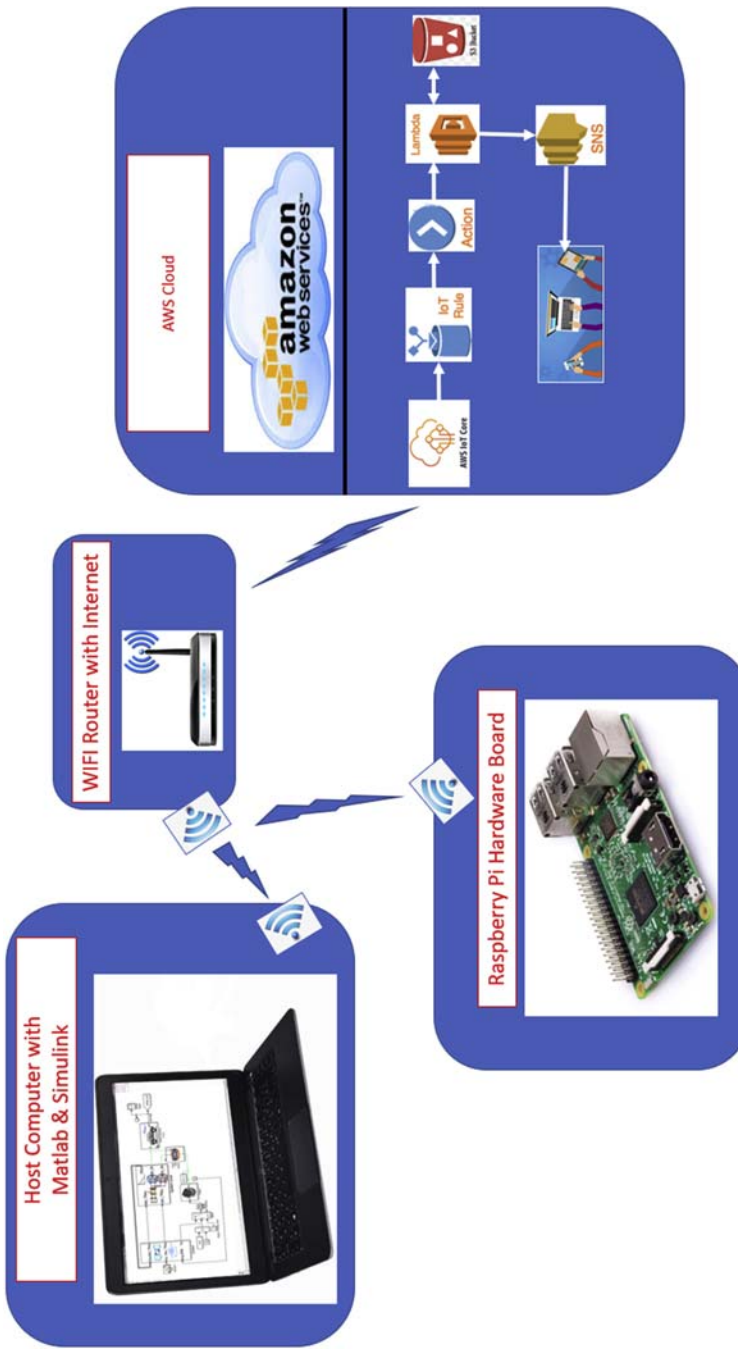


Figure 8.85 Digital Twin deployment high level diagram.

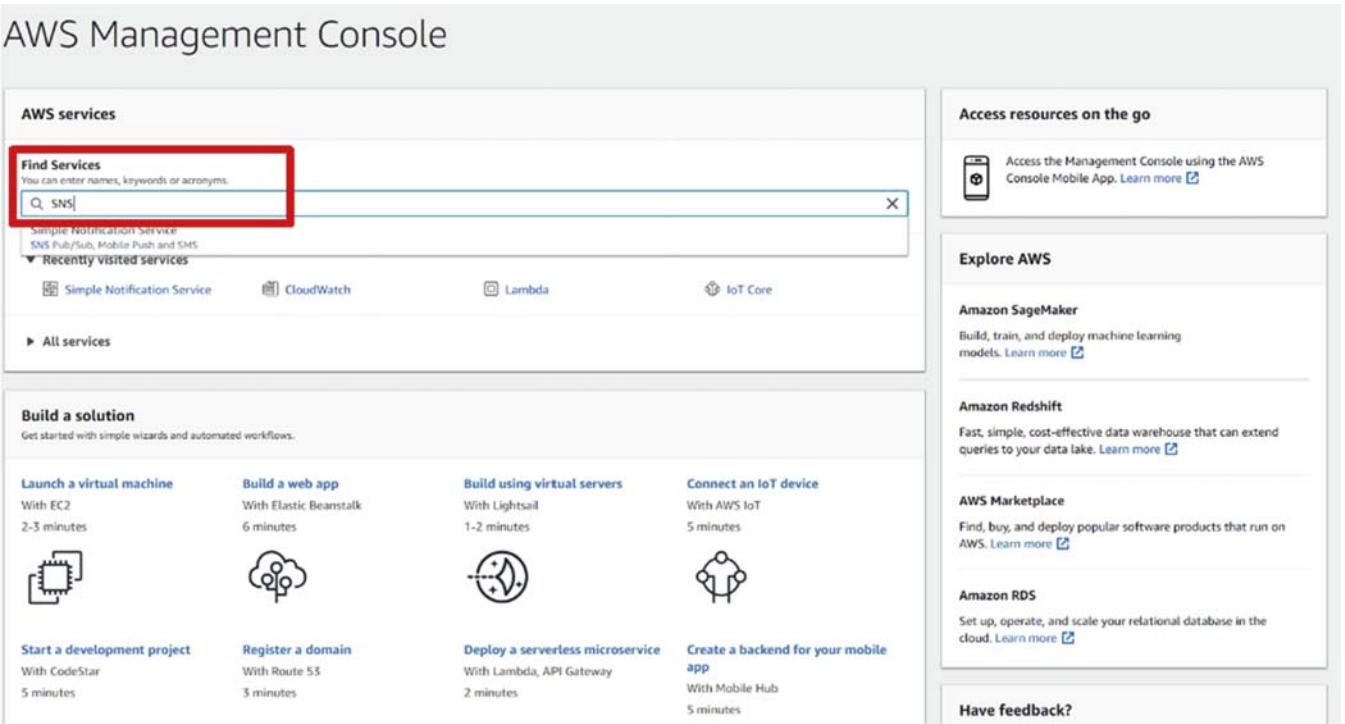


Figure 8.86 Launching Simple Notification Service (SNS).

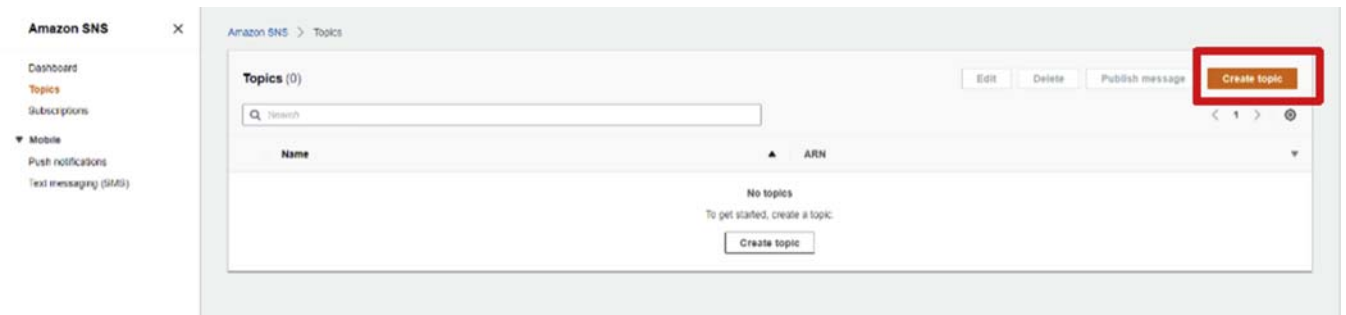


Figure 8.87 Creating an SNS topic.

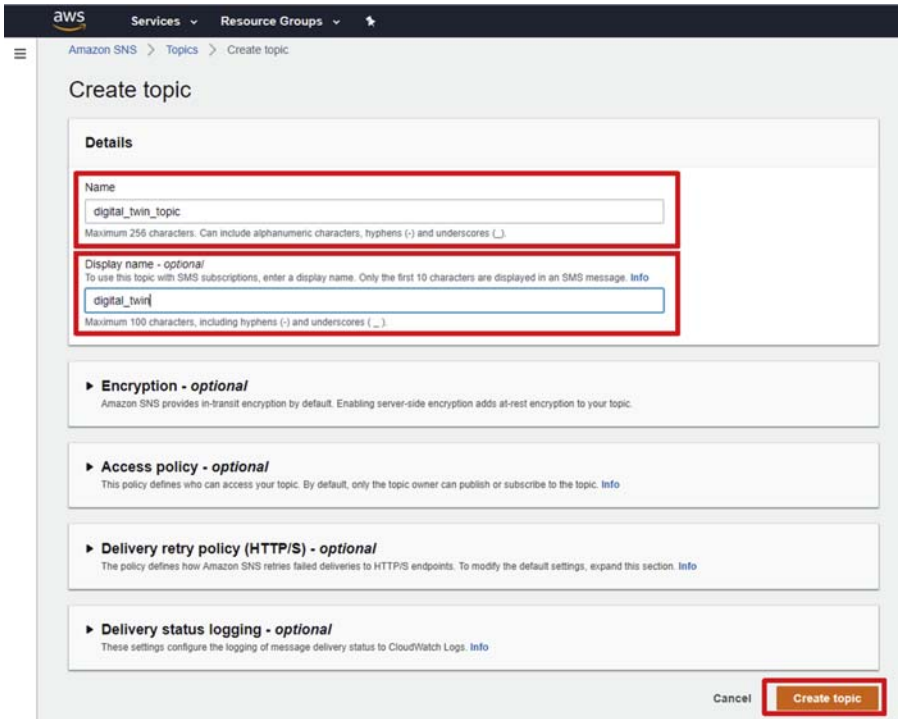


Figure 8.88 New SNS topic details.

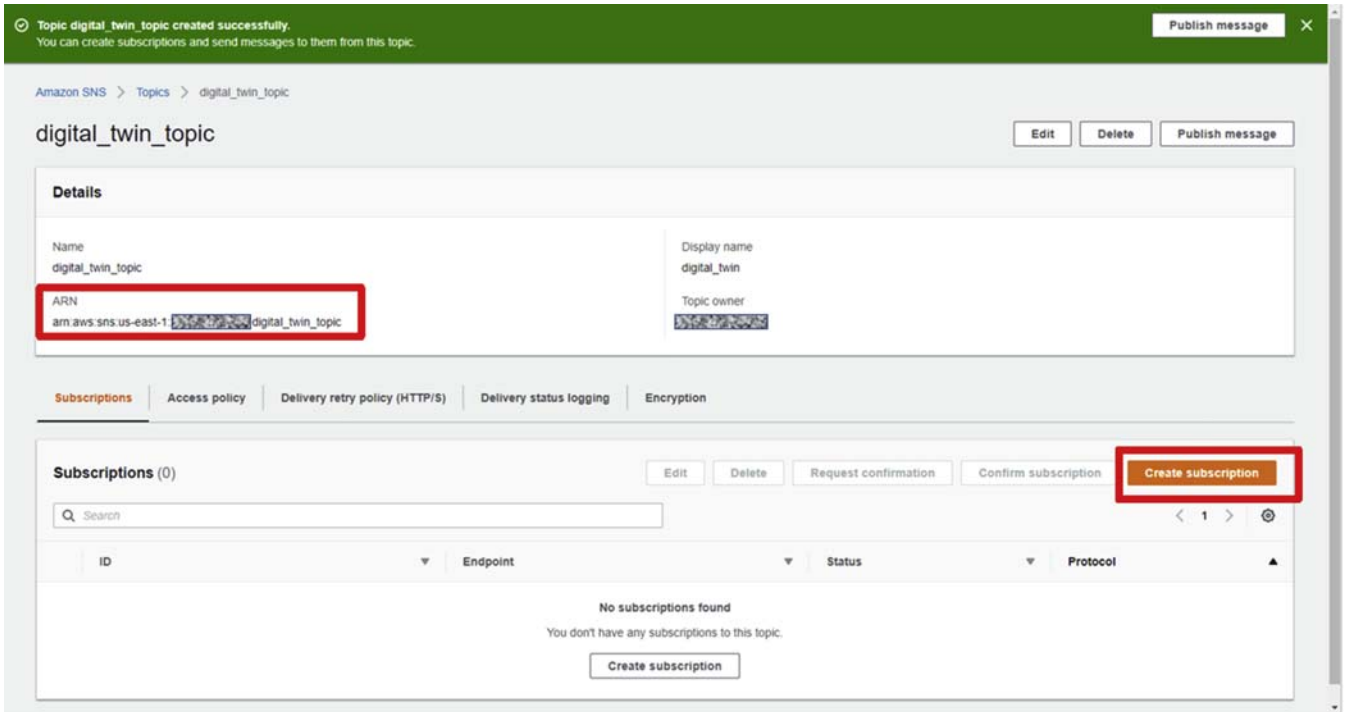


Figure 8.89 Newly created SNS topic.

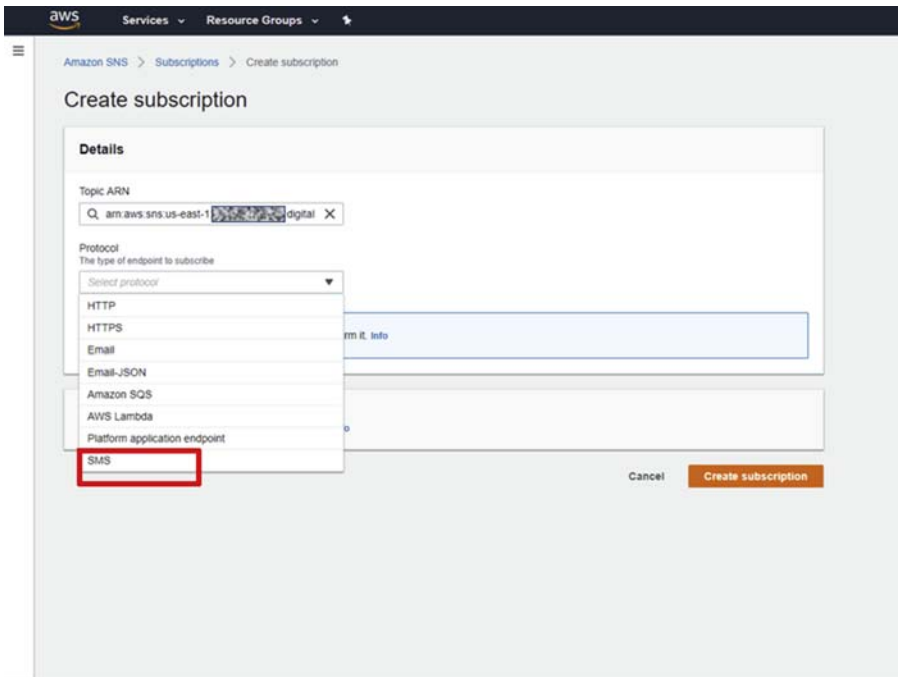


Figure 8.90 Creating subscription for SNS topic.

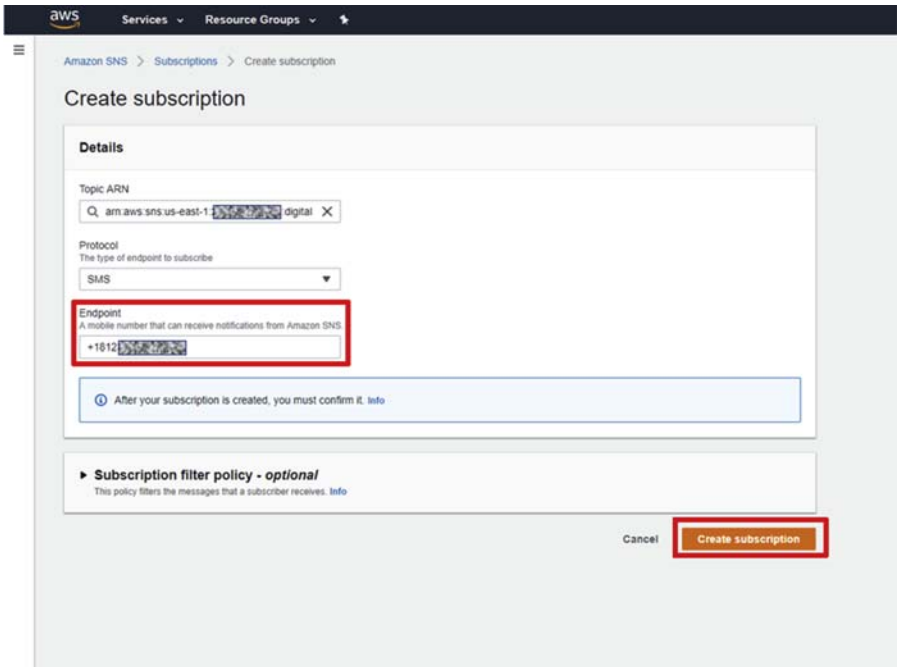


Figure 8.91 Creating text/SMS subscription.

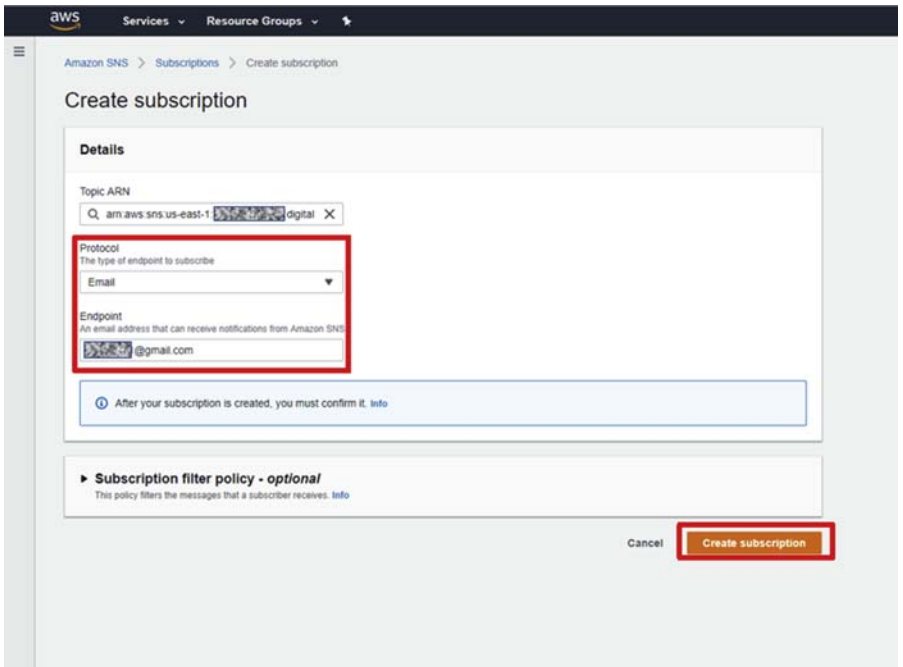


Figure 8.92 Creating E-mail subscription.

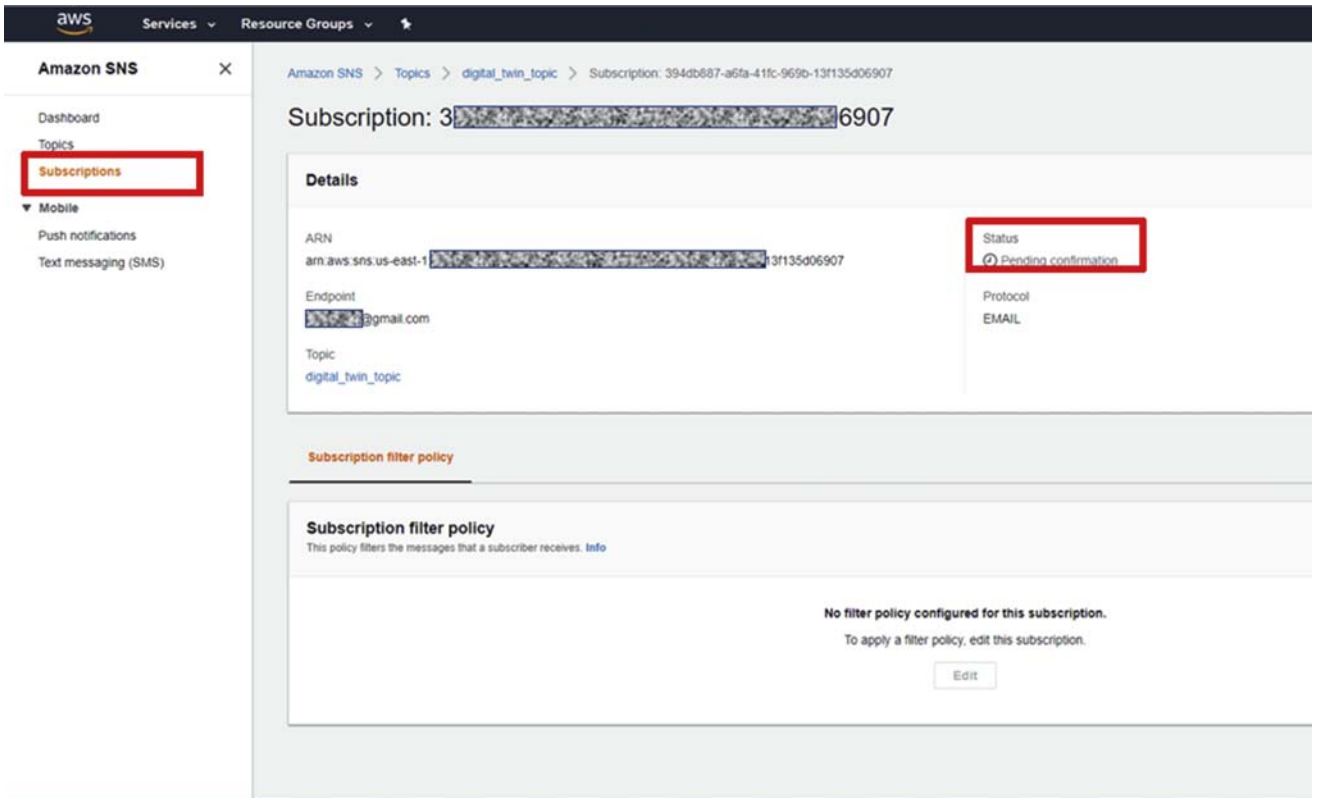


Figure 8.93 Pending subscription.

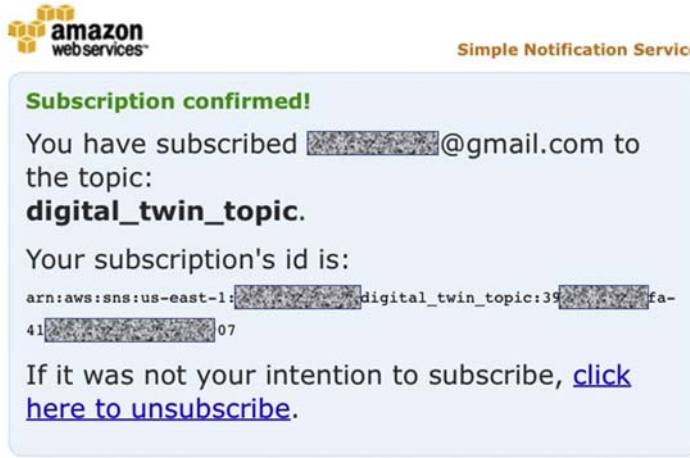


Figure 8.94 Subscription confirmation.

12. Toward the bottom of the new Lambda function window, there is an option to give the role that defines the permissions for the Lambda function. Drop down the menu and select "**Create a new role from AWS policy templates**" as shown in Fig. 8.100. What we want to do is allow the Lambda function to publish messages to the SNS topics that we created and configured in the above steps. The permission has to be explicitly given to the lambda function; otherwise, it will not be able to interact with the SNS service.
13. The new **Create role** form will show up, which involves a four-step process. On the first step, select the ***AWS service*** option and click on the ***Next: permissions*** button as shown in Fig. 8.101.
14. In the second step as shown in Figs. 8.102 and 8.103, search for SNS and S3 in the ***Filter policies*** search bar and select the ***AmazonSNSFullAccess*** and ***AmazonS3FullAccess*** policies and click on the ***Next: Tags*** button.
15. Next Adding the Tags is an optional step. We can skip it.
16. The final step is the Review, see Fig. 8.104. On the Review window, give a Role name, we chose **hev*_digital_twin_role*** in this case. We can give any name to the Role. It can be seen that the policies ***AmazonSNSFullAccess*** and ***AmazonS3FullAccess*** we selected in the previous step is attached to this role. The idea is we will attach this role to the Lambda function, which will give the Lambda function access to the SNS and S3 Bucket services and publish Text/E-mails to the topics and store and retrieve data as needed. Finish the role creation by clicking on the ***Create role*** button.
17. The new Role will be created and shown under the AWS service Identity and Access Management or called as IAM. The new Role that we created is shown under the IAM Console as shown in Figs. 8.105 and 8.106.
18. After the new Execution Role is created go back to the AWS Lambda function console, under the lambda function we created earlier, select the "***Use an existing role***" option and browse and select the new role we created named ***hev_digital_twin_role*.** If the User chose to give a different name for the role in the above step, chose the role name accordingly. See Fig. 8.107 for details.
19. Save the Lambda function by clicking on the ***Save*** button at the top as shown in Fig. 8.108.
20. Next step is to add a trigger for the Lambda function, to decide when will the Lambda function run. Click on the Add trigger button shown in Fig. 8.109. We will be triggering the Lambda function whenever the IoT Thing receives data from the Raspberry Pi Hardware.

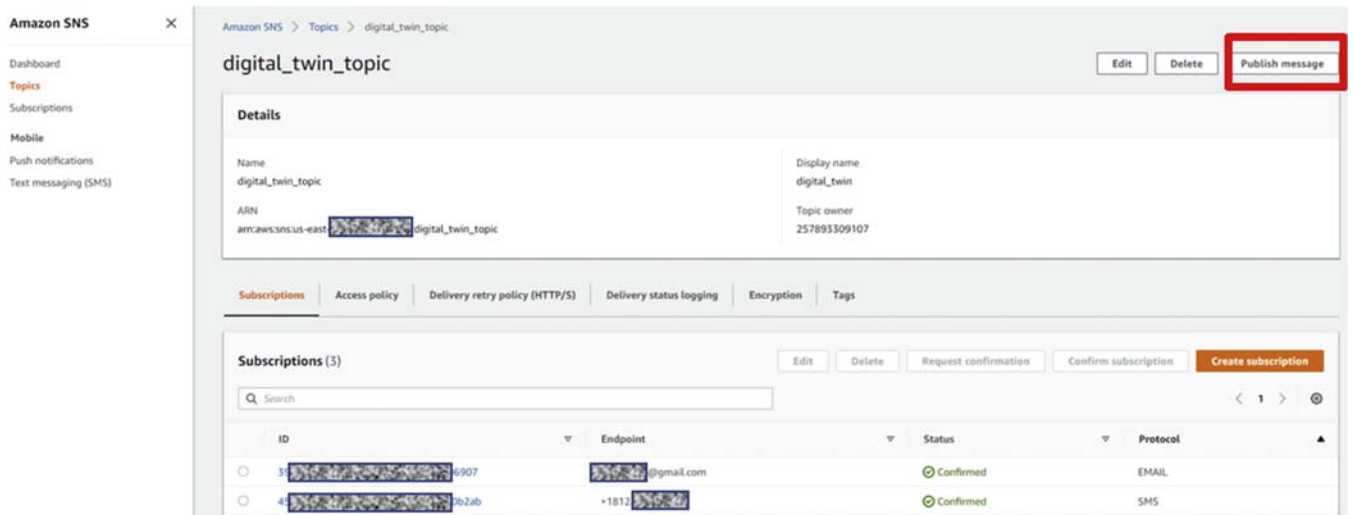


Figure 8.95 Confirmed subscriptions.

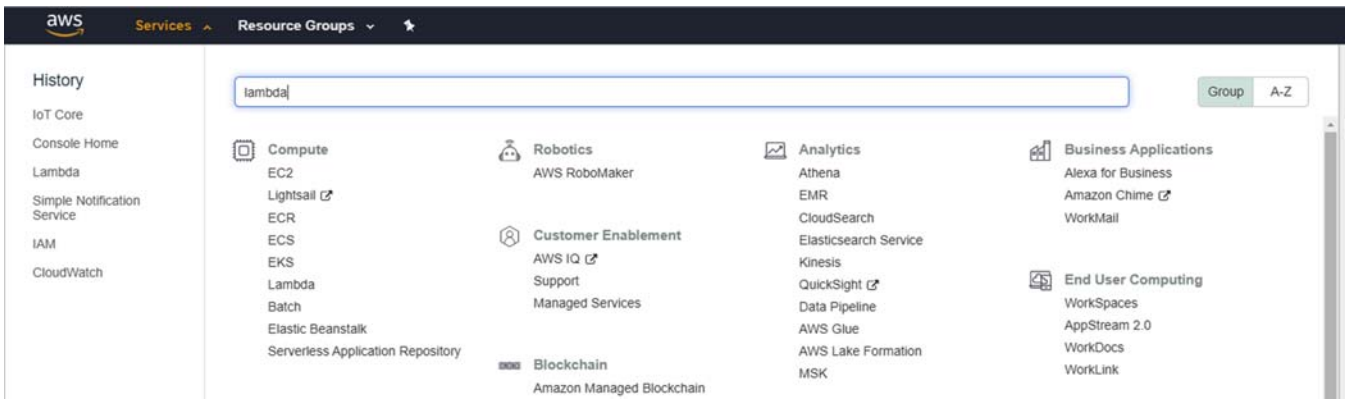


Figure 8.96 Launching AWS Lambda functions.

Figure 8.97 Creating Lambda function.

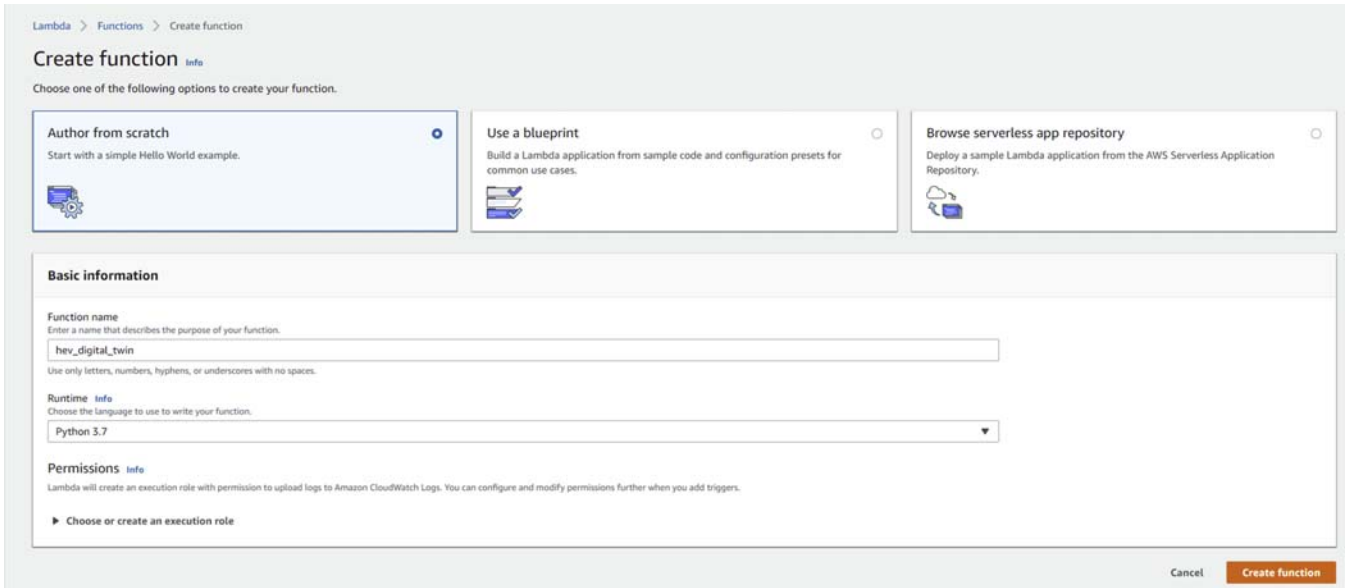


Figure 8.98 New Lambda function details.

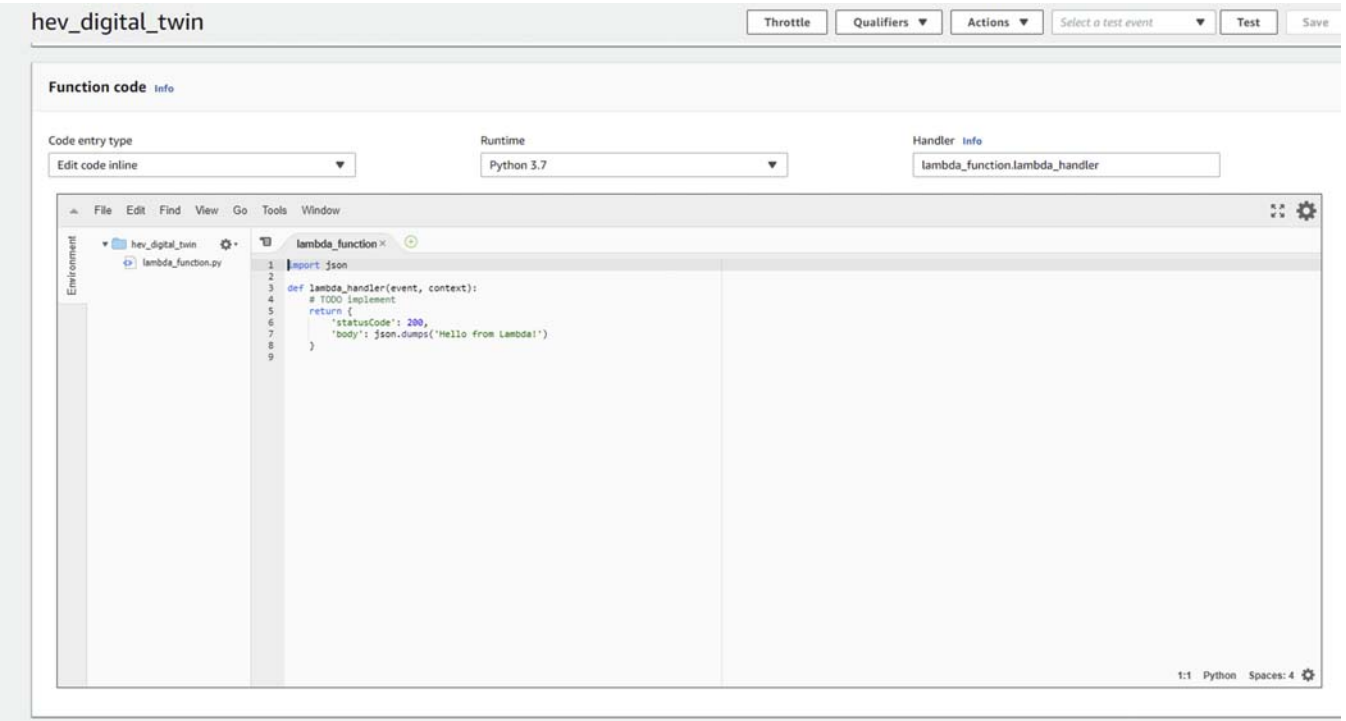


Figure 8.99 Default Lambda function body.

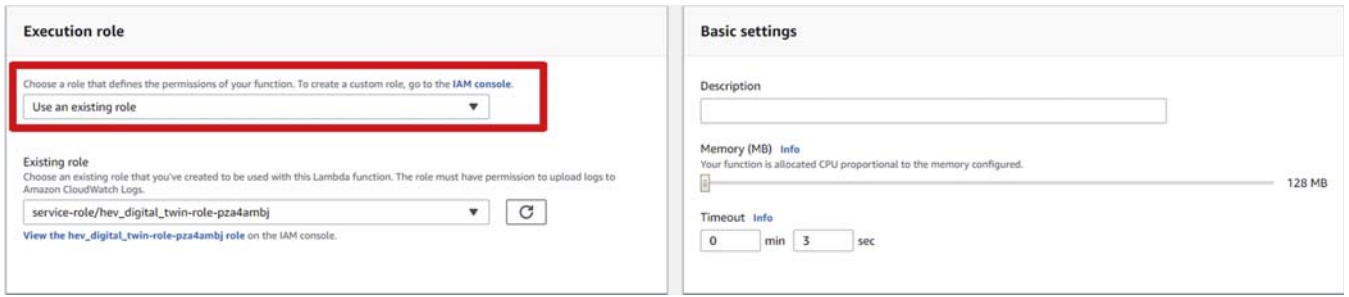


Figure 8.100 Creating an Execution Role.

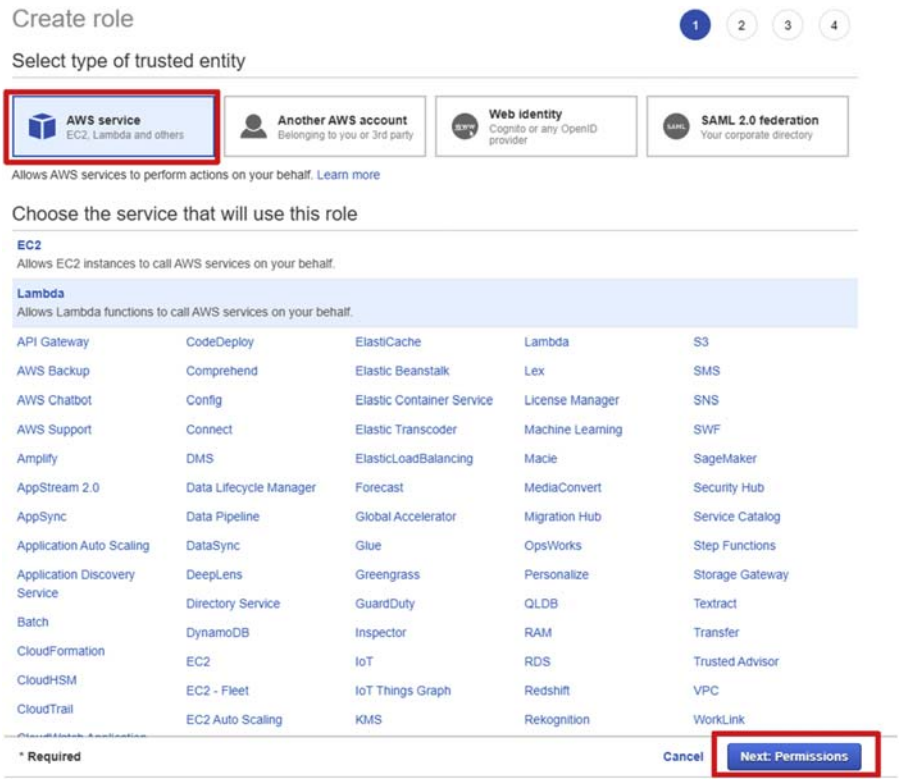


Figure 8.101 Create a new role.

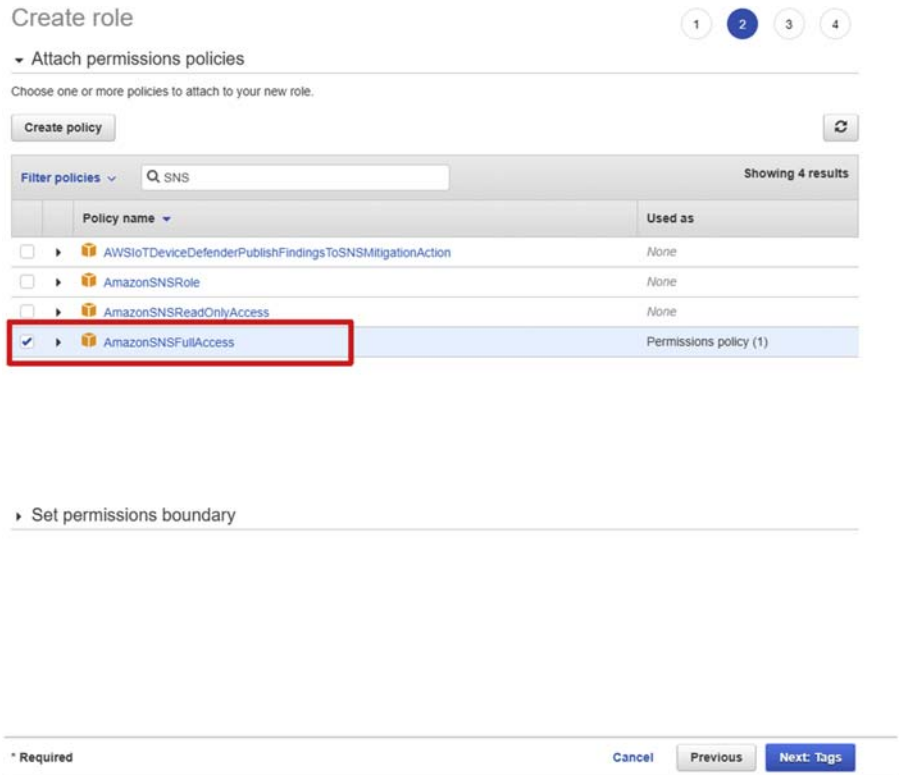


Figure 8.102 Attach SNS permission policy to the new role.

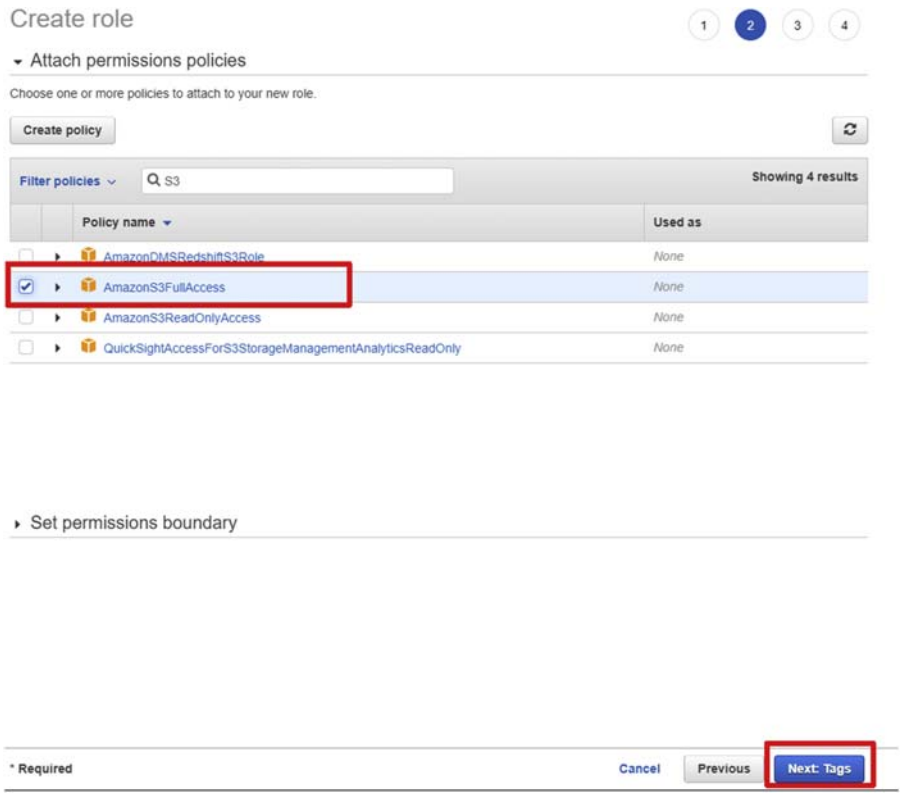


Figure 8.103 Attach S3 bucket permission policy to the new role.

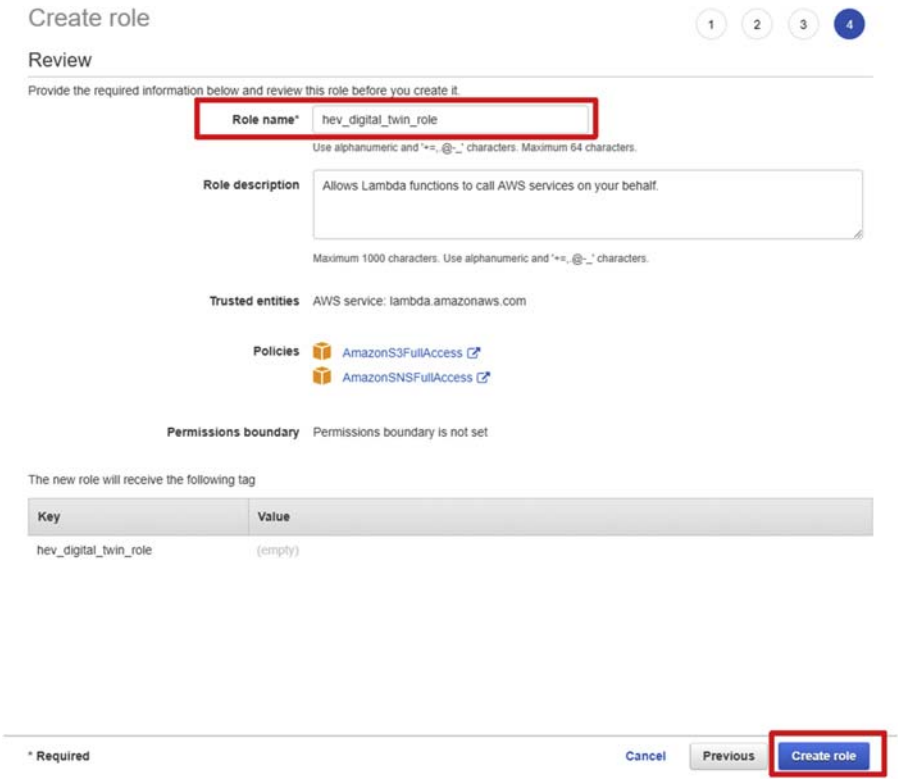


Figure 8.104 Naming the new role.

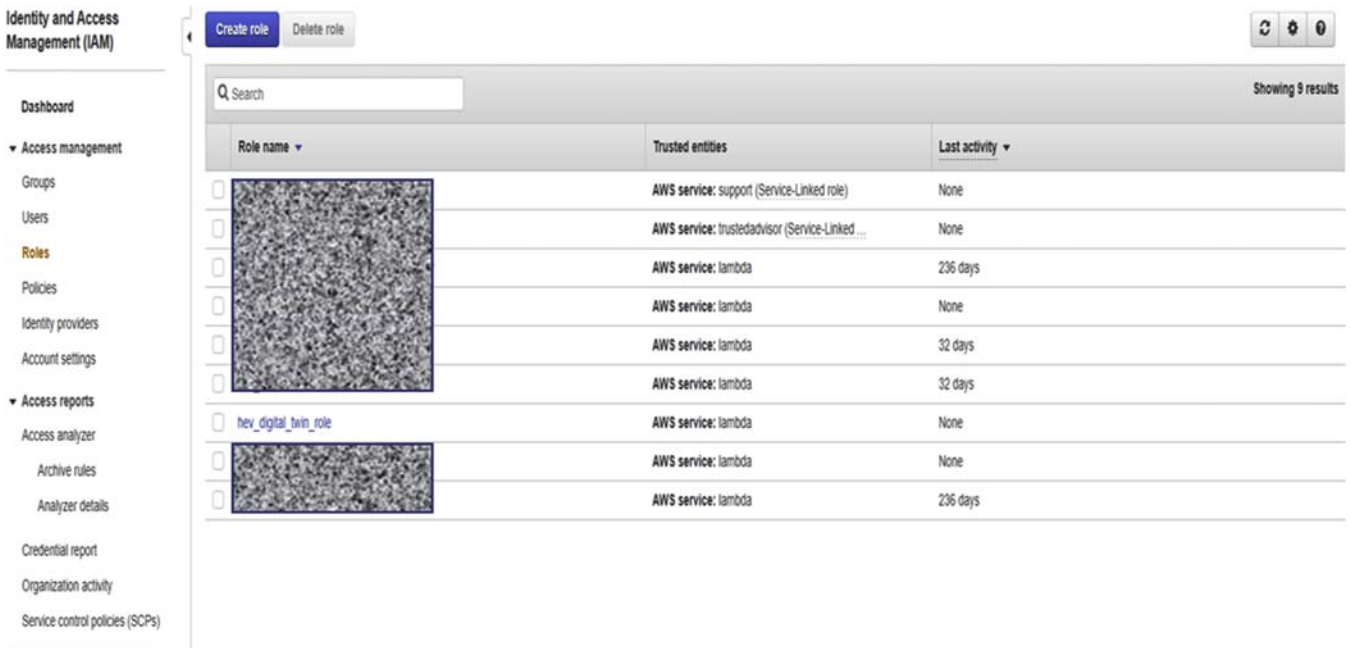


Figure 8.105 New role created.

Figure 8.106 New role created showing the SNS and S3 policies attached.

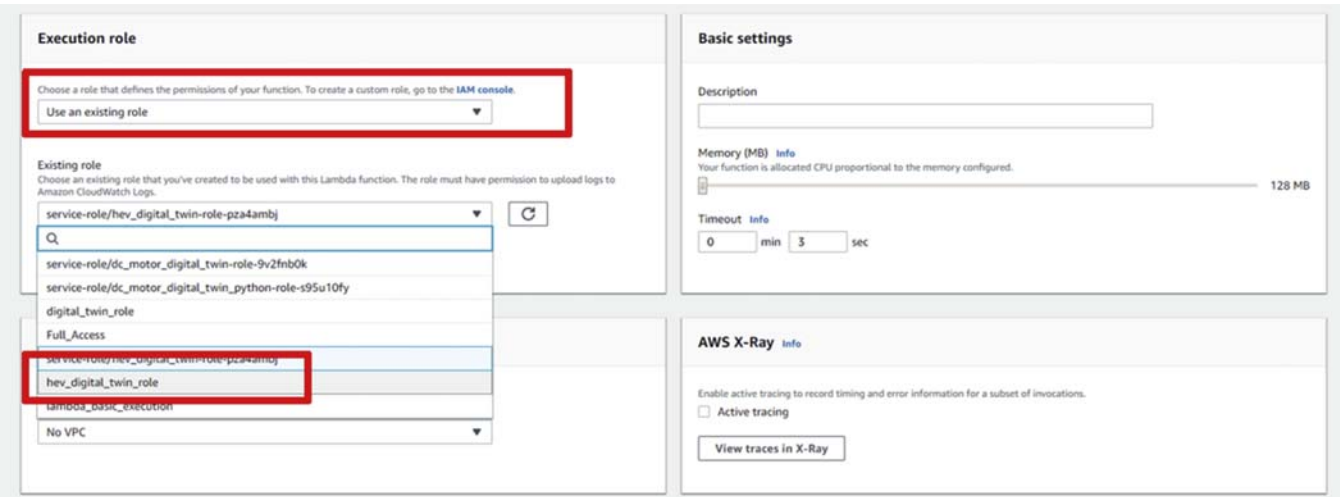


Figure 8.107 Selecting the new Execution Role in the Lambda function.

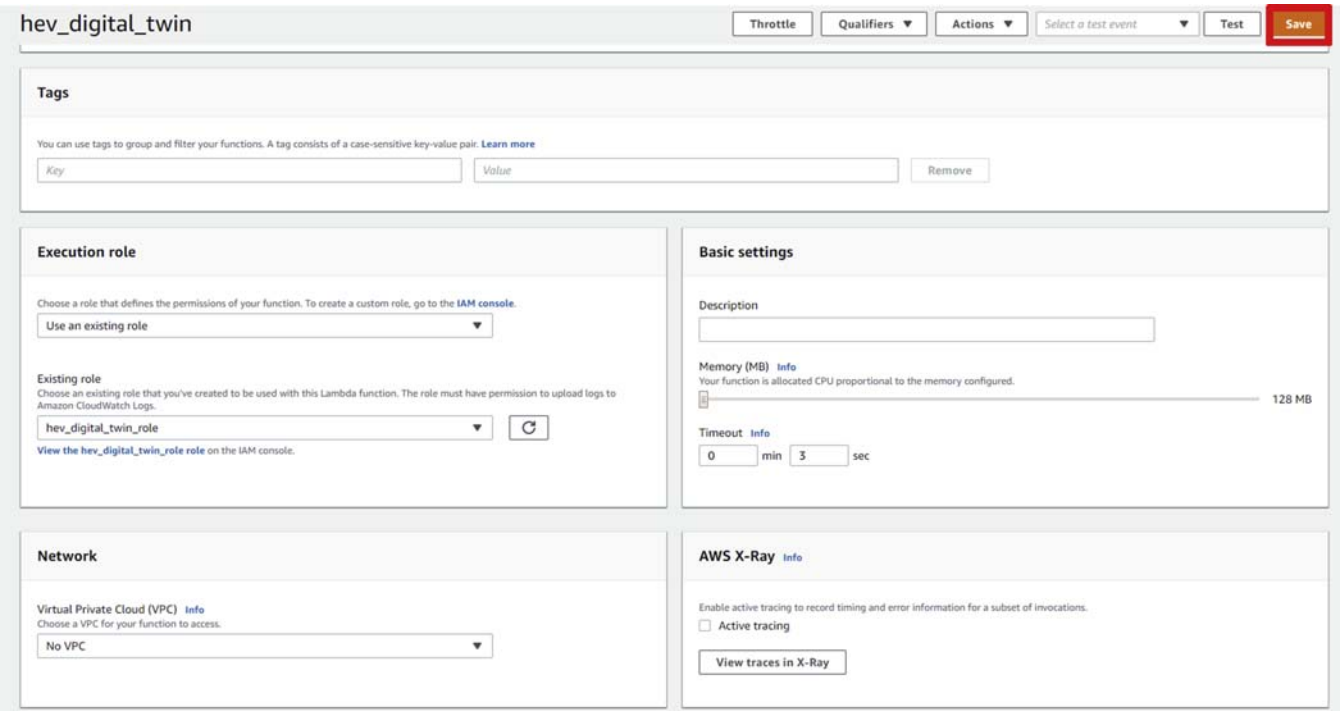


Figure 8.108 Save Lambda function with updated Execution Role.

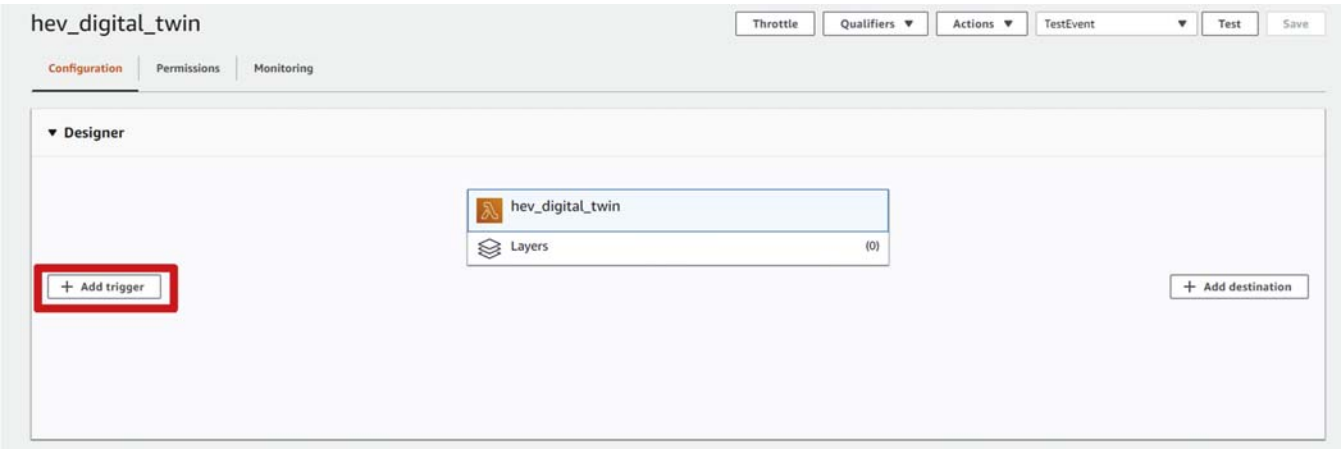


Figure 8.109 Adding trigger to Lambda function.

21. The Add trigger window will be opened as shown in Figs. 8.110 and 8.111. Select the AWS IoT from the dropdown as the trigger source. We need to create a Custom IoT Rule and use it for the trigger. An IoT routes the IoT Thing updates to a specific AWS service. In this case, we are going to create a rule to trigger the AWS Lambda service. Check the box "***Custome IoT rule***" and select the option "***Create a new rule***." Give a name to the new rule, in this case we used ***hev_digital_twin_rule***. The ***Rule description*** is optional. In the ***Rule query statement***, enter the string **SELECT * FROM '$aws/things/hev_digital_twin_thing/shadow/update/accepted'**. This query statement will pull all values from the IoT Thing update and pass it to the AWS Lambda. Also check the "***Enable trigger***" option and click on the ***Add*** button.
22. The Lambda function will now show the AWS IoT as the Trigger source as shown in Fig. 8.110.
23. Next step is developing the Lambda function. At this point, we only have a skeleton template Lambda function, so we will have to expand on that. For convenience of testing the Lambda function along with the Digital Twin compiled executable and to package the Lambda function and Digital Twin executable to deploy to AWS, we have developed the Lambda function in Linux. From the Ubuntu Linux (which we used for compiling the Digital Twin model earlier), open an editor and name the file as **lambda_function.py**. At the top of the Python script, add all the packages we will be using in this program. So here the package **json** is used to deal with the incoming json string from AWS IoT, **os** is used to call the compiled Digital Twin executable, **boto3** is used to interact with the AWS SNS, AWS S3 Bucket, **csv** for creating csv file with incoming data, and **math** is for some math calculations for **Off-BD** algorithm. See Fig. 8.112.
24. Next we will expand the Lambda handler function. We can see from Fig. 8.99 that when we create the AWS Lambda function it creates a default handler function named **lambda_handler.** The input argument **event** contains the JSON string that we sent from the Raspberry Pi Hardware module with the HEV state data while the model runs on the hardware. As shown in Fig. 8.113, we will parse the JSON string and store it into a comma separated input.csv file.
25. A logic is added to identify whether the Lambda function is getting triggered for first time during a particular run of an HEV cycle in the hardware. If it is triggered for the first time, the data received by the Lambda function will be treated as the very first 5 s data from the hardware, and if it is not the first time, Lambda function is triggered in the HEV cycle run, then we have to combine the last 5 s data with all the previously received data for running the HEV Digital Twin. So for the very first time we run Digital Twin with first 5 s data and store all the data information from the hardware into Amazon S3 storage bucket, when the next 5 s data are received from the hardware we first append the previously stored inputs with the newly received inputs and then run the Digital Twin model. So the Digital Twin model runs in a way that it is always starting from the same initial condition and states when the Hardware starts to run. This approach takes out the hassle of reinitializing the states of the Digital Twin HEV to the state it was before 5 s when the new data arrives. It is possible, but it can be more challenging. As shown in Fig. 8.114, the input_data[6] in the received data will contain the free running counter information from the HEV

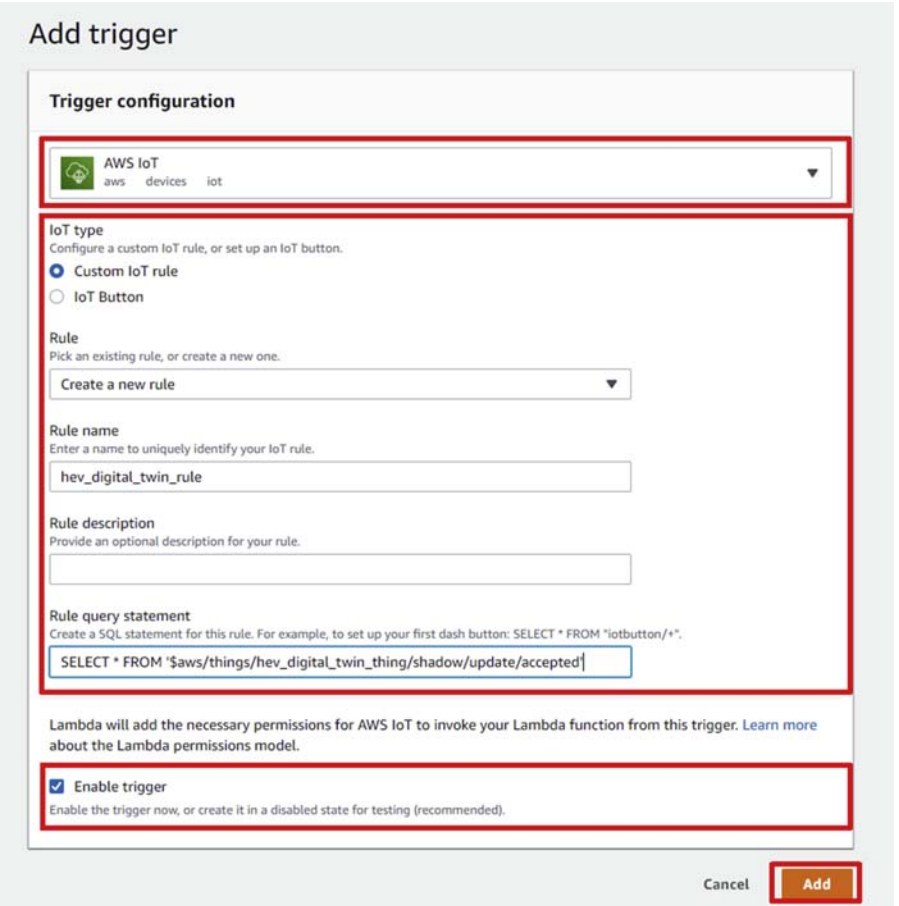


Figure 8.110 Adding a trigger for the Lambda function from the AWS IoT.

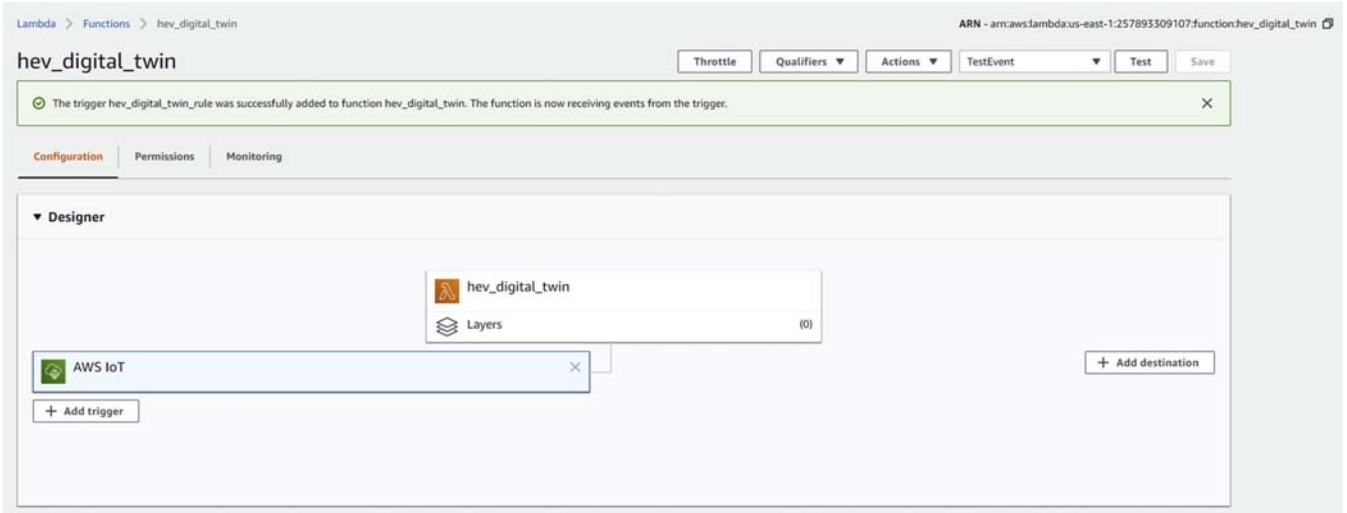


Figure 8.111 Lambda function showing the AWS IoT trigger added.

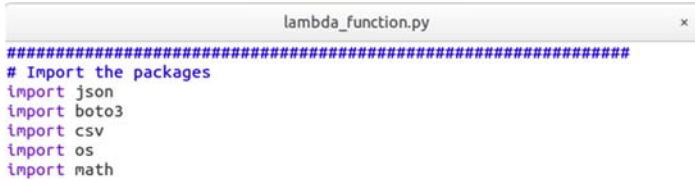


Figure 8.112 Updating Lambda function importing packages.

Simulink model running in Raspberry Pi, so a value of 0 will indicate the data are from the first 5 s of run. So if the counter value 0 is present, we make the flag to get previous data from the S3 bucket to 0, and if the counter value of 0 is not present, we make the flag to get previous data from S3 bucket to 1.

26. Next we will initialize the objects for the AWS S3 and SNS clients and also initialize the arrays to store the HEV signals from the actual hardware and Digital Twin. See Fig. 8.115.

27. Next we use the flag set in Step 25, and if flag is set to copy the previous history information from the S3 bucket, we copy the file **hev_history.csv** from the HEV bucket named **digital-twinbucket**. The process of creating and configuring the S3 bucket is explained in later steps. Combine the inputs from the previous history file along with the new data for the past 5 s received into one file named **hev_history_updated.csv**. See Fig. 8.116.

28. Copy the newly created file **hev_history_updated.csv** back to the AWS S3 bucket as a history information for the next Lambda trigger. Also make another copy of this file and name it as **hen_input.csv** to be given as input to the Digital Twin model. Once the **hev_input.csv** file is created, we will trigger the compiled executable Digital Twin model. Note that we are calling the compiled executable **HEV_Simscape_Digital_Twin**. See Fig. 8.117. When the executable is run, it will create the **hev_output.csv** file in the **/tmp** folder as we have seen and tested before. We will make a copy of that file also to the AWS S3 bucket if in case we need to do offline analysis or debugging of the Digital Twin model.

29. Now we have the actual and Digital Twin model predicted data for the HEV System, and we are ready to perform the Off-BD algorithm. First, read the **hev_input.csv** and **hev_output.csv** files from the **/tmp** folder and store it into arrays as shown in Fig. 8.118.

30. Next as shown in Figs. 8.119 and 8.120, calculates the RMSE between the actual and predicted HEV states and output signals.

31. The Off-BD algorithm has calculated the RMSE value and now we need to compare the RMSE value against certain threshold and take a decision. As shown in Fig. 8.121, we picked different thresholds for different signals. So if the corresponding RMSE values are less than the threshold Off-BD, algorithm decides there is no failure detected, and if the RMSE exceeds the thresholds, there is a failure. Finding out the right threshold value may involve a little bit of a tuning exercise generally. For publishing the Off-BD decision to the SNS, we use the SNS object using the boto3 package. We will use the topic ARN noted from Step 4 to publish message. Call the function **sns.publish** with the topic ARN and custom message based on the Off-BD diagnostic decision. User can edit the Message

```
#####################################################################
# Lambda Handler function which runs when IoT Trigger Happens
def lambda_handler(event, context):
    # TODO implement

#####################################################################
# Create a file "input.csv" under /tmp folder and write the input and output values
# separated by commas in each row with the newly recieved data from the AWS IoT.
# There will be 50 data samples for coming in for the past 5 Seconds
    with open('/tmp/input.csv', 'w') as writeFile:
        writer = csv.writer(writeFile)
        for loop_index in range(1,51):
            temp_str = "Data_" + str(loop_index)
            input_list = event['state']['desired'][temp_str]
            writer.writerows([[(input_list[0]),(input_list[1]),(input_list[2]),(input_list[3]),(input_list[4]),(input_list[5]),
(input_list[6])]])
```

Figure 8.113 Updating Lambda function reading trigger input data.

```
# If the index number in the data entry if it is 0, that means it is the start of the
# HEV simulation in the Raspberry PI, if so we dont have to get the previous stored data
# file from AWS S3 Bucket. If the count is not zero that means this is not the first time
# the Lambda function is triggered for this HEV simulation in the Raspberry PI. So we have
# to set a flag to retrieve the previously stored data from S3 Bucket.
            if loop_index ==1:
                temp_first_val = float(input_list[6])
                if int(temp_first_val) ==0:
                    get_file_from_s3_bucket = 0;
                else :
                    get_file_from_s3_bucket =1;
            #print(input_list)
    writeFile.close()
```

Figure 8.114 Updating Lambda function decision logic to check start of run or not.

```
# Setup the AWS S3 Bucket and SNS Clients and also SNS Topic ARN for sending messages
    s3 = boto3.resource('s3')
    sns = boto3.client(service_name="sns")
    topicArn = '[redacted]:digital_twin_topic'
# Initialize the arrays for storing actual signals from hardware and digital twin predicted signals
    data_count =0;
    Actual_Veh_Speed = [];Actual_Motor_Speed = [];Actual_Generator_Speed = [];Actual_Engine_Speed = [];Actual_Battery_SOC = [];
    Digital_Twin_Veh_Speed = [];Digital_Twin_Motor_Speed = [];Digital_Twin_Generator_Speed = [];Digital_Twin_Engine_Speed =
[];Digital_Twin_Battery_SOC = [];
```

Figure 8.115 Updating Lambda function initializing SNS and S3 clients and arrays to store actual and Digital Twin data.

string however they want, this is the message we will see in the TEXT and E-mail messages when the Lambda function runs. The Lambda function development is now finished.

32. Save the above developed Lambda function into a new folder. In this case, we named the folder to be **Digital_Twin_Lambda_Function** under the folder **Digtal_Twin_Simscape_-book/Chapter_8**. Copy the Digital Twin compiled executable **HEV_Simscape_Digital_Twin** from the previous section also to this folder. Fig. 8.122 shows a set of Linux commands to give the necessary executable permissions and packaging of the Lambda function and Digital Twin executable. The commands are explained below. Open a Linux terminal and run the below commands in steps.

- **a.** The ***ls*** command shows the folder has the compiled executable **HEV_Simscape_Digital_Twin** and the lambda function **lambda_function.py.**
- **b.** We need to give full read/write/executable permissions to all files in this folder for the AWS to be able to run it. Use the command ***sudo chmod -R 777 <folder_name>***.
- **c.** Run the command ***ls -l.*** This will list all the files and folders in the current folder and their permissions. Everything should be showing ***rwxrwxrwx***.
- **d.** Now package the lambda function and compiled executable using the command ***zip bundle.zip lambda_function.py HEV_Simscape_Digital_Twin***. This will zip the files ***lambda_function.py*** and ***HEV_Simscape_Digital_Twin*** into a file named ***bundle.zip***.
- **e.** After the bundle.zip is created, once again repeat the Step c to allow read/write/executable permissions to the ***bundle.zip*** file as well.
- **f.** The ***ls*** command will show the newly created ***bundle.zip*** file as well. Now we are ready to deploy the Lambda function to AWS and do the final testing.

```
# if the Lambda function is not triggered for the first time in this HEV Raspberry PI run cycle that is
# if time is greater than first 5 seconds, copy the previously stored CSV file hev_history.csv from the Amazon S3 Bucket
# named digitaltwinhevbucket. Download the hev_history.csv to the /tmp folder
    # Get File from S3 Bucket
    if get_file_from_s3_bucket:
        s3.meta.client.download_file('digitaltwinhevbucket', 'hev_history.csv', '/tmp/hev_history.csv')
# Open a new file hev_history_updated.csv for writing in the /tmp folder
        with open('/tmp/hev_history_updated.csv', 'w') as writeFile:
            writer = csv.writer(writeFile)
# Open the recently downloaded history file from S3 bucket hev_history.csv for reading
            with open('/tmp/hev_history.csv','r') as csvfile:
                plots = csv.reader(csvfile, delimiter=',')
# Copy all entries from hev_history.csv to hev_history_updated.csv
                for row in plots:
                    writer.writerows([[(row[0]),(row[1]),(row[2]),(row[3]),(row[4]),(row[5]),(row[6])]])
                    Actual_Veh_Speed.append(float(row[1]))
                    Actual_Motor_Speed.append(float(row[2]))
                    Actual_Generator_Speed.append(float(row[3]))
                    Actual_Engine_Speed.append(float(row[4]))
                    Actual_Battery_SOC.append(float(row[5]))
                    data_count = data_count +1;
            csvfile.close()
# Now open the input.csv file which is created with fresh sample of 50 data points from the past
# 5 Seconds and append that dat points to hev_history_updated.csv file
            with open('/tmp/input.csv','r') as csvfile:
                plots = csv.reader(csvfile, delimiter=',')
                for row in plots:
                    writer.writerows([[(row[0]),(row[1]),(row[2]),(row[3]),(row[4]),(row[5]),(row[6])]])
                    Actual_Veh_Speed.append(float(row[1]))
                    Actual_Motor_Speed.append(float(row[2]))
                    Actual_Generator_Speed.append(float(row[3]))
                    Actual_Engine_Speed.append(float(row[4]))
                    Actual_Battery_SOC.append(float(row[5]))
                    data_count = data_count +1;
            csvfile.close()
```

Figure 8.116 Updating Lambda function combining previous history input data with newly received data.

```
# Copy the hev_history_updated.csv back to the AWS S3 Bucket
        s3.meta.client.upload_file('/tmp/hev_history_updated.csv', 'digitaltwinhevbucket', 'hev_history.csv')
# Copy the file hev_history_updated.csv from /tmp to /tmp/hev_input.csv. The hev_input.csv will be the input file
# for running the Digital Twin Model. And it contains all the previous data along with the past 5 seconds data
        cmd = 'cp /tmp/hev_history_updated.csv /tmp/hev_input.csv'
        so = os.popen(cmd).read()
# Run the Digital Twin Model HEV_Simscape_Digital_Twin
        cmd = './HEV_Simscape_Digital_Twin'
        so = os.popen(cmd).read()
# Collect the /tmp/hev_output.csv file and copy to the AWS S3 Bucket
        s3.meta.client.upload_file('/tmp/hev_output.csv', 'digitaltwinhevbucket', 'hev_output.csv')
```

Figure 8.117 Updating Lambda function running Digital Twin model.

```
# Open the /tmp/input.csv and /tmp/hev_output.csv files for signals comparison between Actual data
# from Raspberry PI hardware and Digital Twin predicted values
        with open('/tmp/input.csv','r') as csvfile:
            plots = csv.reader(csvfile, delimiter=',')
            for row in plots:
                Actual_Veh_Speed.append(float(row[1]))
                Actual_Motor_Speed.append(float(row[2]))
                Actual_Generator_Speed.append(float(row[3]))
                Actual_Engine_Speed.append(float(row[4]))
                Actual_Battery_SOC.append(float(row[5]))
                data_count = data_count +1;


    with open('/tmp/hev_output.csv','r') as csvfile:
        plots = csv.reader(csvfile)
        for row in plots:
            Digital_Twin_Veh_Speed.append(float(row[1]))
            Digital_Twin_Motor_Speed.append(float(row[2]))
            Digital_Twin_Generator_Speed.append(float(row[3]))
            Digital_Twin_Engine_Speed.append(float(row[4]))
            Digital_Twin_Battery_SOC.append(float(row[5]))
```

Figure 8.118 Updating Lambda function reading input and output files for actual and predicted data.

```
# Calculate the Error Between Actual and Predicted Speeds
    Veh_Speed_Error = [None]*data_count
    Veh_Speed_Squared_Error = [None]*data_count
    Veh_Speed_Mean_Squared_Error = 0
    Veh_Speed_Root_Mean_Squared_Error = 0

    Motor_Speed_Error = [None]*data_count
    Motor_Speed_Squared_Error = [None]*data_count
    Motor_Speed_Mean_Squared_Error = 0
    Motor_Speed_Root_Mean_Squared_Error = 0

    Generator_Speed_Error = [None]*data_count
    Generator_Speed_Squared_Error = [None]*data_count
    Generator_Speed_Mean_Squared_Error = 0
    Generator_Speed_Root_Mean_Squared_Error = 0

    Engine_Speed_Error = [None]*data_count
    Engine_Speed_Squared_Error = [None]*data_count
    Engine_Speed_Mean_Squared_Error = 0
    Engine_Speed_Root_Mean_Squared_Error = 0

    Battery_SOC_Error = [None]*data_count
    Battery_SOC_Squared_Error = [None]*data_count
    Battery_SOC_Mean_Squared_Error = 0
    Battery_SOC_Root_Mean_Squared_Error = 0
```

Figure 8.119 Updating Lambda function initializing arrays for Root Mean Square Error calculation.

33. From the Linux operating system itself, open the web browser, the AWS Management Console, and the Lambda function **hev_digtal_twin** we created. Select the option from the drop down "**Upload a .zip file**" as shown in Fig. 8.123. Browse and select the ***bundle.-zip*** file we packaged earlier as shown in Fig. 8.124.

```
# Calculate the Root Mean Squared Error Between Actual and Predicted Data
    for index in range(data_count-1):
        Veh_Speed_Error[index] = Actual_Veh_Speed[index] - Digital_Twin_Veh_Speed[index];
        Veh_Speed_Squared_Error[index] = Veh_Speed_Error[index]*Veh_Speed_Error[index];
        Veh_Speed_Mean_Squared_Error = Veh_Speed_Mean_Squared_Error + Veh_Speed_Squared_Error[index];
        Motor_Speed_Error[index] = Actual_Motor_Speed[index] - Digital_Twin_Motor_Speed[index];
        Motor_Speed_Squared_Error[index] = Motor_Speed_Error[index]*Motor_Speed_Error[index];
        Motor_Speed_Mean_Squared_Error = Motor_Speed_Mean_Squared_Error + Motor_Speed_Squared_Error[index];
        Generator_Speed_Error[index] = Actual_Generator_Speed[index] - Digital_Twin_Generator_Speed[index];
        Generator_Speed_Squared_Error[index] = Generator_Speed_Error[index]*Generator_Speed_Error[index];
        Generator_Speed_Mean_Squared_Error = Generator_Speed_Mean_Squared_Error + Generator_Speed_Squared_Error[index]
        Engine_Speed_Error[index] = Actual_Engine_Speed[index] - Digital_Twin_Engine_Speed[index];
        Engine_Speed_Squared_Error[index] = Engine_Speed_Error[index]*Engine_Speed_Error[index];
        Engine_Speed_Mean_Squared_Error = Engine_Speed_Mean_Squared_Error + Engine_Speed_Squared_Error[index]
        Battery_SOC_Error[index] = Actual_Battery_SOC[index] - Digital_Twin_Battery_SOC[index];
        Battery_SOC_Squared_Error[index] = Battery_SOC_Error[index]*Battery_SOC_Error[index];
        Battery_SOC_Mean_Squared_Error = Battery_SOC_Mean_Squared_Error + Battery_SOC_Squared_Error[index]

    Veh_Speed_Mean_Squared_Error = Veh_Speed_Mean_Squared_Error/data_count;
    Veh_Speed_Root_Mean_Squared_Error = math.sqrt(Veh_Speed_Mean_Squared_Error)

    Motor_Speed_Mean_Squared_Error = Motor_Speed_Mean_Squared_Error/data_count;
    Motor_Speed_Root_Mean_Squared_Error = math.sqrt(Motor_Speed_Mean_Squared_Error)

    Generator_Speed_Mean_Squared_Error = Generator_Speed_Mean_Squared_Error/data_count;
    Generator_Speed_Root_Mean_Squared_Error = math.sqrt(Generator_Speed_Mean_Squared_Error)

    Engine_Speed_Mean_Squared_Error = Engine_Speed_Mean_Squared_Error/data_count;
    Engine_Speed_Root_Mean_Squared_Error = math.sqrt(Engine_Speed_Mean_Squared_Error)

    Battery_SOC_Mean_Squared_Error = Battery_SOC_Mean_Squared_Error/data_count;
    Battery_SOC_Root_Mean_Squared_Error = math.sqrt(Battery_SOC_Mean_Squared_Error)
```

Figure 8.120 Updating Lambda function calculating Root Mean Squared Error for actual and predicted HEV states and output.

```
# Create a Message String for Email and Text
    msg_str1 = 'HEV Digital Twin Off-BD Detected a Problem... Root Mean Square Error for Vehicle Speed = ' +
str(Veh_Speed_Root_Mean_Squared_Error) + ' for Motor Speed = ' +  str(Motor_Speed_Root_Mean_Squared_Error) + ' for Generator Speed = ' +
str(Generator_Speed_Root_Mean_Squared_Error) + ' for Engine Speed = ' +  str(Engine_Speed_Root_Mean_Squared_Error) + ' for Battery SOC = ' +
str(Battery_SOC_Root_Mean_Squared_Error)

    msg_str2 = 'No Problem Detected by HEV Digital Twin Off-BD... Root Mean Square Error for Vehicle Speed = ' +
str(Veh_Speed_Root_Mean_Squared_Error) + ' for Motor Speed = ' +  str(Motor_Speed_Root_Mean_Squared_Error) + ' for Generator Speed = ' +
str(Generator_Speed_Root_Mean_Squared_Error) + ' for Engine Speed = ' +  str(Engine_Speed_Root_Mean_Squared_Error) + ' for Battery SOC = ' +
str(Battery_SOC_Root_Mean_Squared_Error)

# Compare the RMSE Values against some threshold to determine if there is Diagnostics Failure conditions
# detected . If Failure Detected send SNS notification to send Email and Text Alerts
    if (Veh_Speed_Root_Mean_Squared_Error > 1 or Motor_Speed_Root_Mean_Squared_Error > 10 or Generator_Speed_Root_Mean_Squared_Error > 150 or
Engine_Speed_Root_Mean_Squared_Error > 150 or Battery_SOC_Root_Mean_Squared_Error >1):
        sns.publish(
        TopicArn = topicArn,
        Message = msg_str1
        )
        print(msg_str1)
    else:
        sns.publish(
        TopicArn = topicArn,
        Message = msg_str2
        )
        print(msg_str2)

    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

Figure 8.121 Updating Lambda function Off-BD detection algorithm with SMS/text SNS alert.

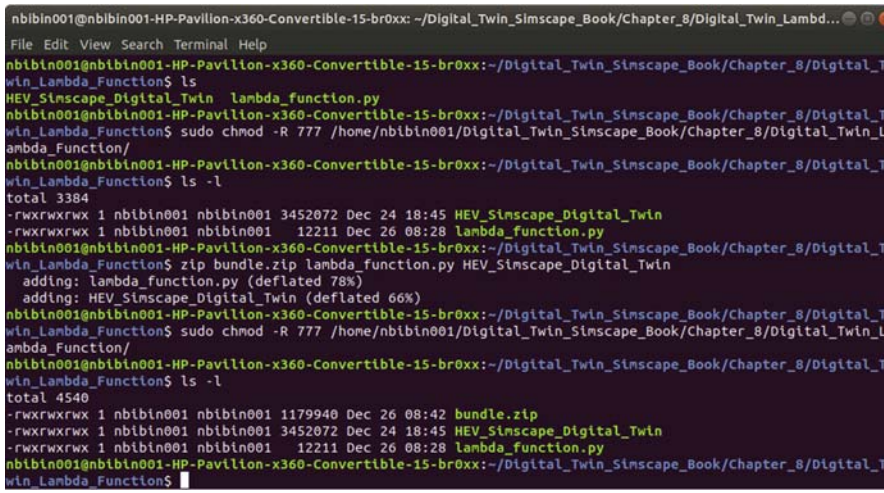


Figure 8.122 Linux command line showing packaging the Lambda function and Digital Twin executable.

34. After the bundle.zip file is successfully uploaded, click on the ***Upload*** and ***Save*** button as shown in Fig. 8.125.
35. We can see, the AWS Lambda function editor now shows our updated Lambda function, and also the Digital Twin compiled executable. See Fig. 8.126.
36. As a last step, we just need to create and configure the AWS S3 bucket which will be used to store and reuse the HEV data in the Lambda function. From the AWS service console, select the **Storage >> S3** as shown in Fig. 8.127.
37. Click on the **Create Bucket** button as shown in Fig. 8.128. On the new window as shown in Fig. 8.129, give a name to the S3 bucket. In this case, we give the name to be **digitaltwin-hevbucket**, note that this should be the same name we used earlier in the Lambda function. Click on Next.
38. On the **Configure Options** screen, we can leave all the settings to be default as shown in Fig. 8.130. Click **Next.**
39. Next on the **Set Permissions** screen, we can check the box "**Block all public access.**" Only we will be using the S3 bucket from Lambda function, so we don't have to grand any public access. See Fig. 8.131, and click **Next**, review, and finalize the S3 bucket creation as shown in Fig. 8.132.
40. Congratulations!!!We are ready to test the whole Off-BD algorithm for the HEV system now.

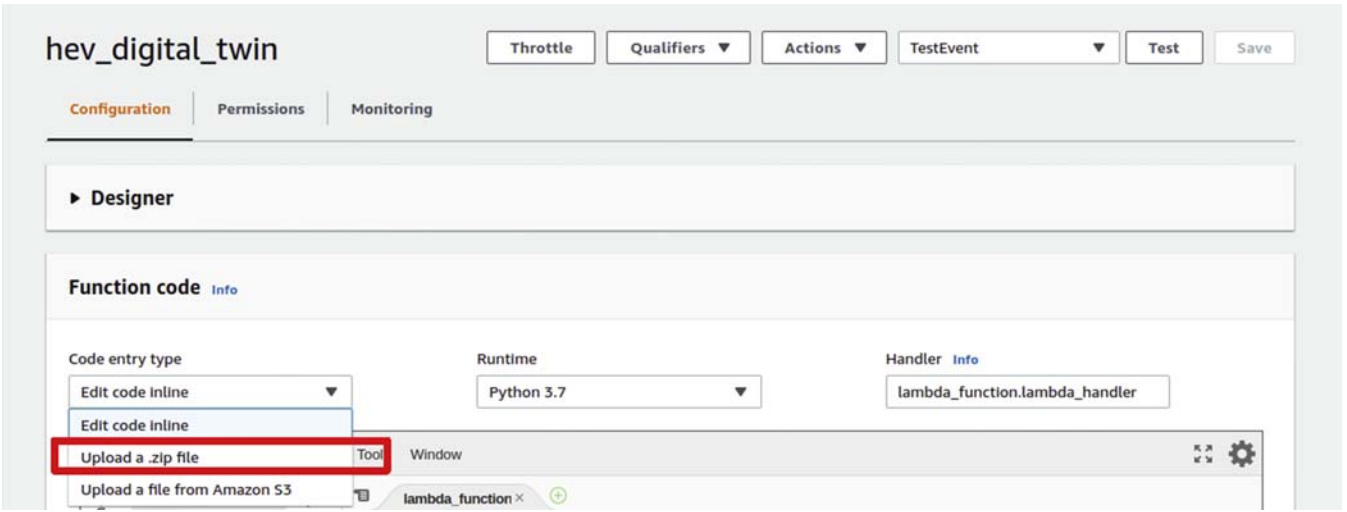


Figure 8.123 Uploading the packaged Lambda function to AWS.

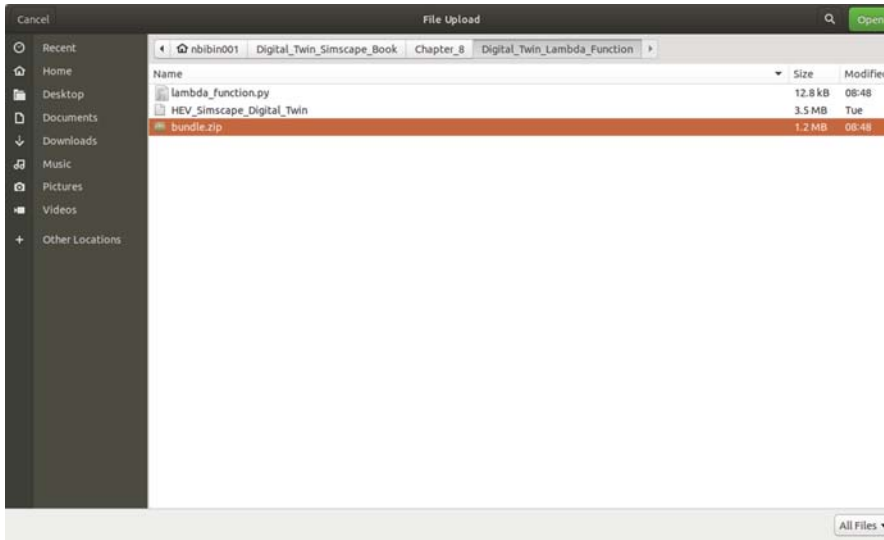


Figure 8.124 Browse and Select the Lambda function zip file package.

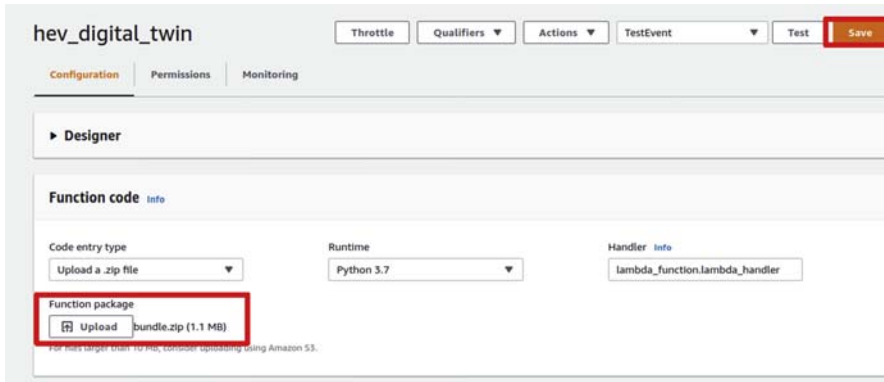


Figure 8.125 Saving the uploaded Lambda function package.

41. First, we will test the No-Fault condition. From the host computer, run the model **HEV_-Simscape_Model_Rasp_Pi_with_MQTT.slx**, which we made earlier, that will run on the Raspberry Pi hardware. First, we will run the no-fault case and test the Off-BD with Digital Twin. Also run the Python code **Raspberry_Pi_AWS_IOT_Cloud_Connection.py** on the Raspberry Pi, while MATLAB compiles and deploys the model to PI. As the model

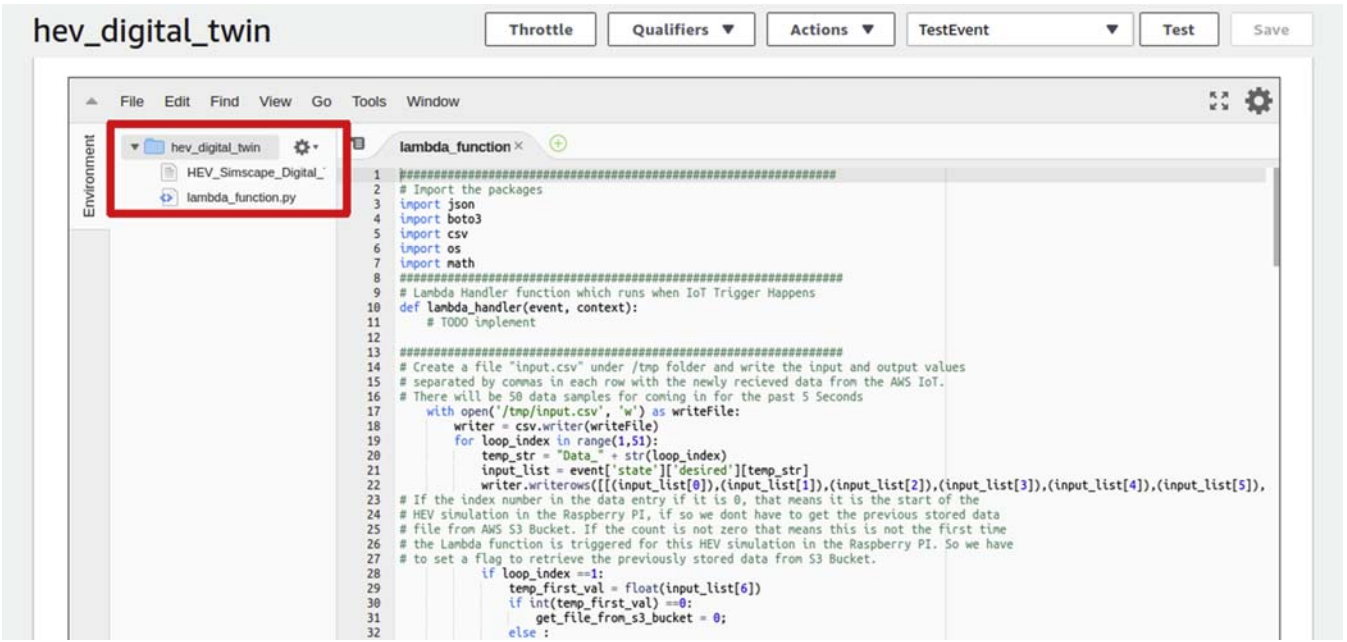


Figure 8.126 Lambda function inline editor showing the uploaded function and executable.

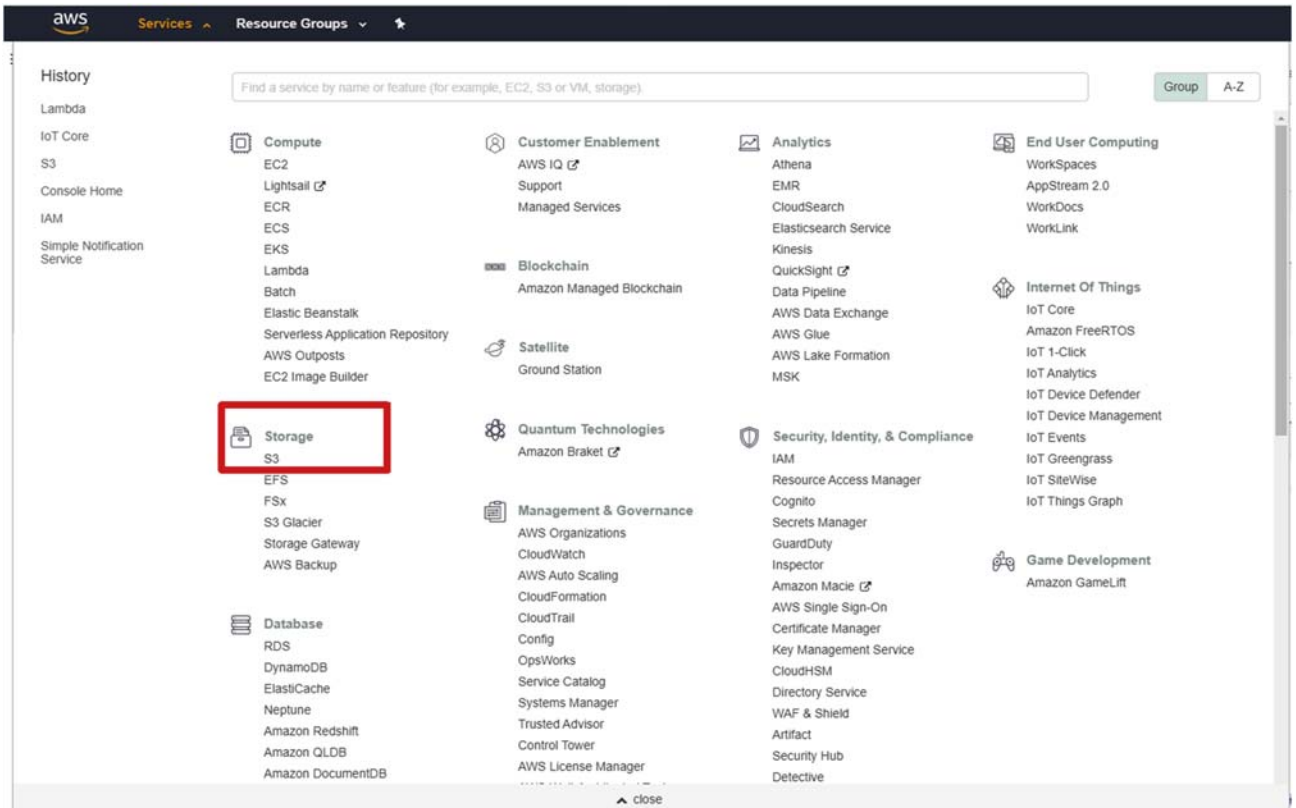


Figure 8.127 Selecting the S3 bucket service from AWS console.

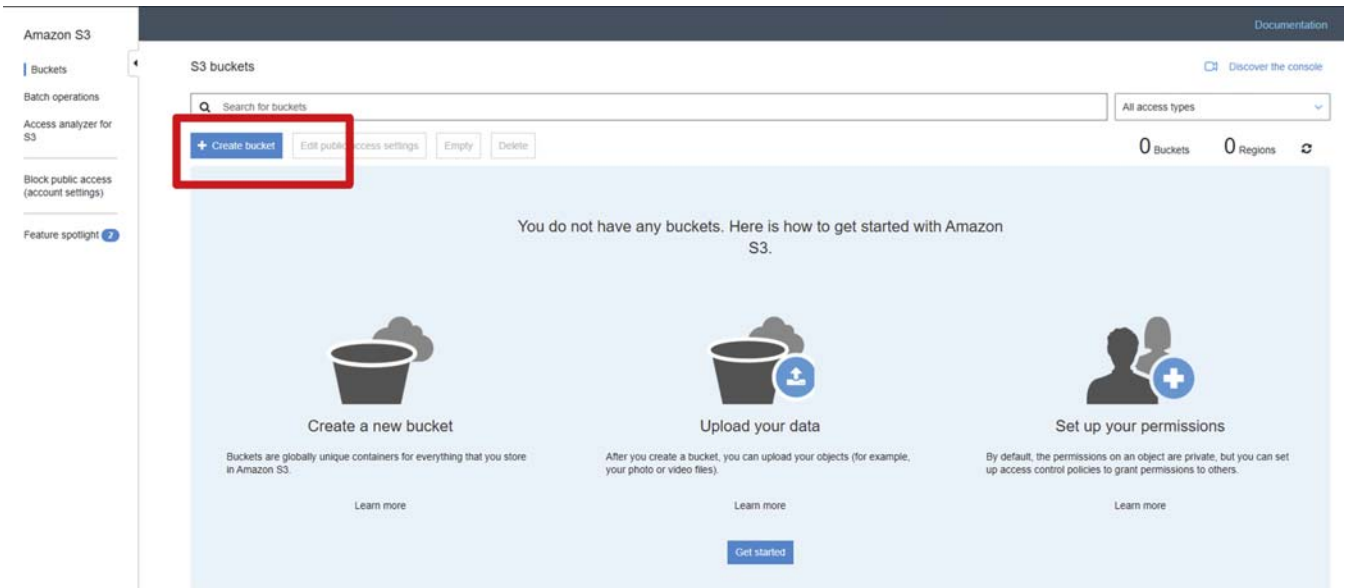


Figure 8.128 Creating a new S3 bucket.

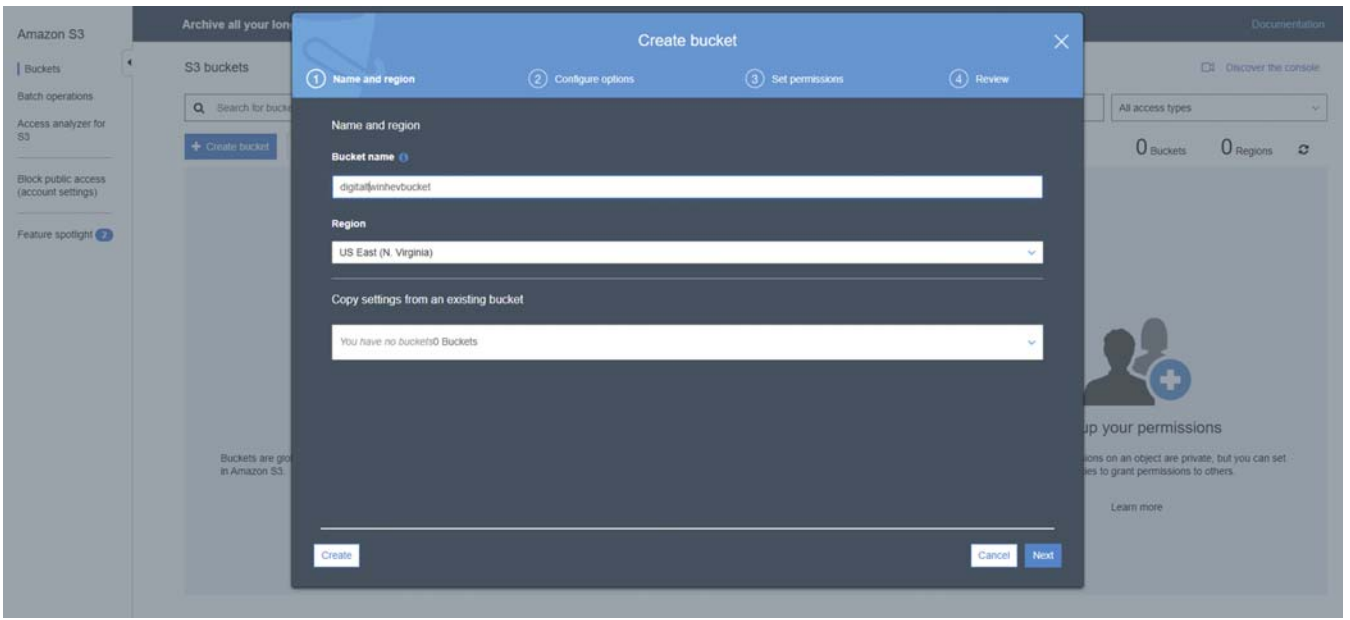


Figure 8.129 Naming the AWS S3 bucket.

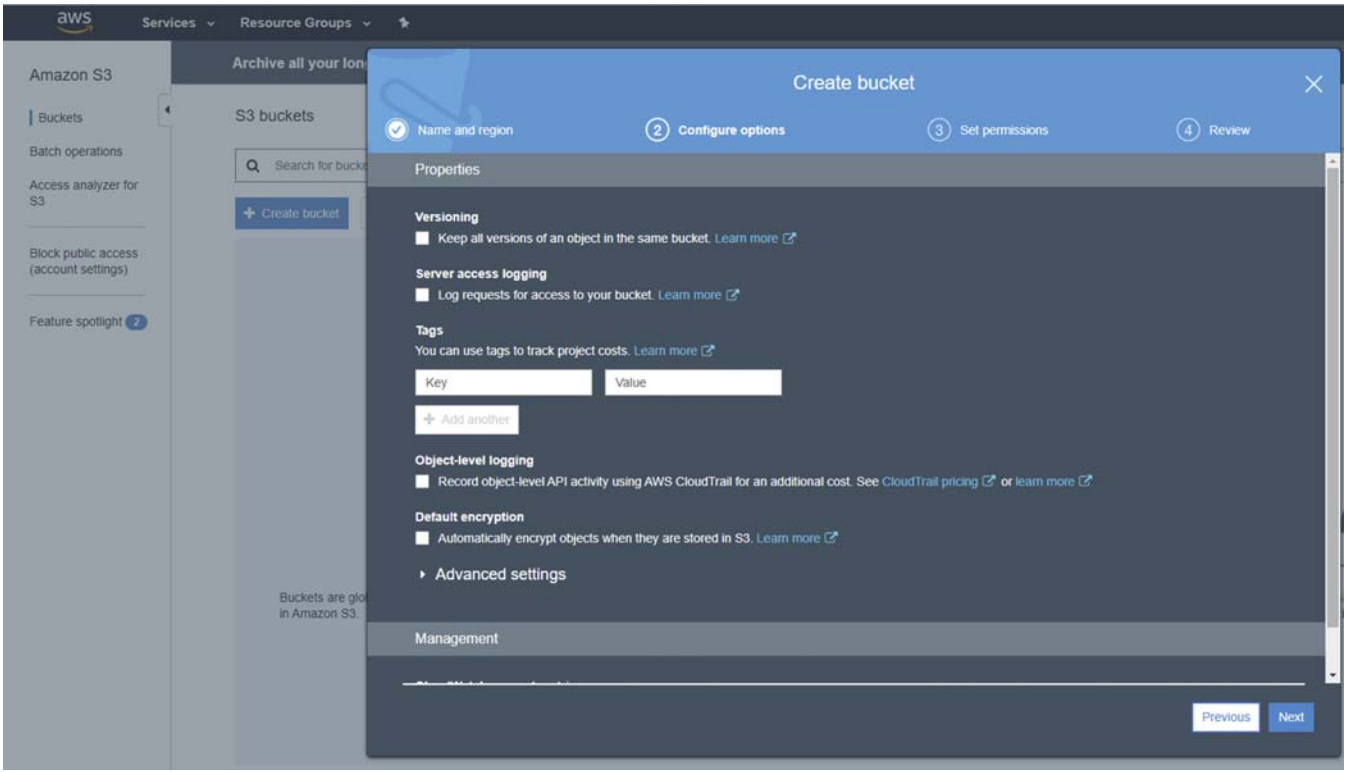


Figure 8.130 AWS S3 Configure options.

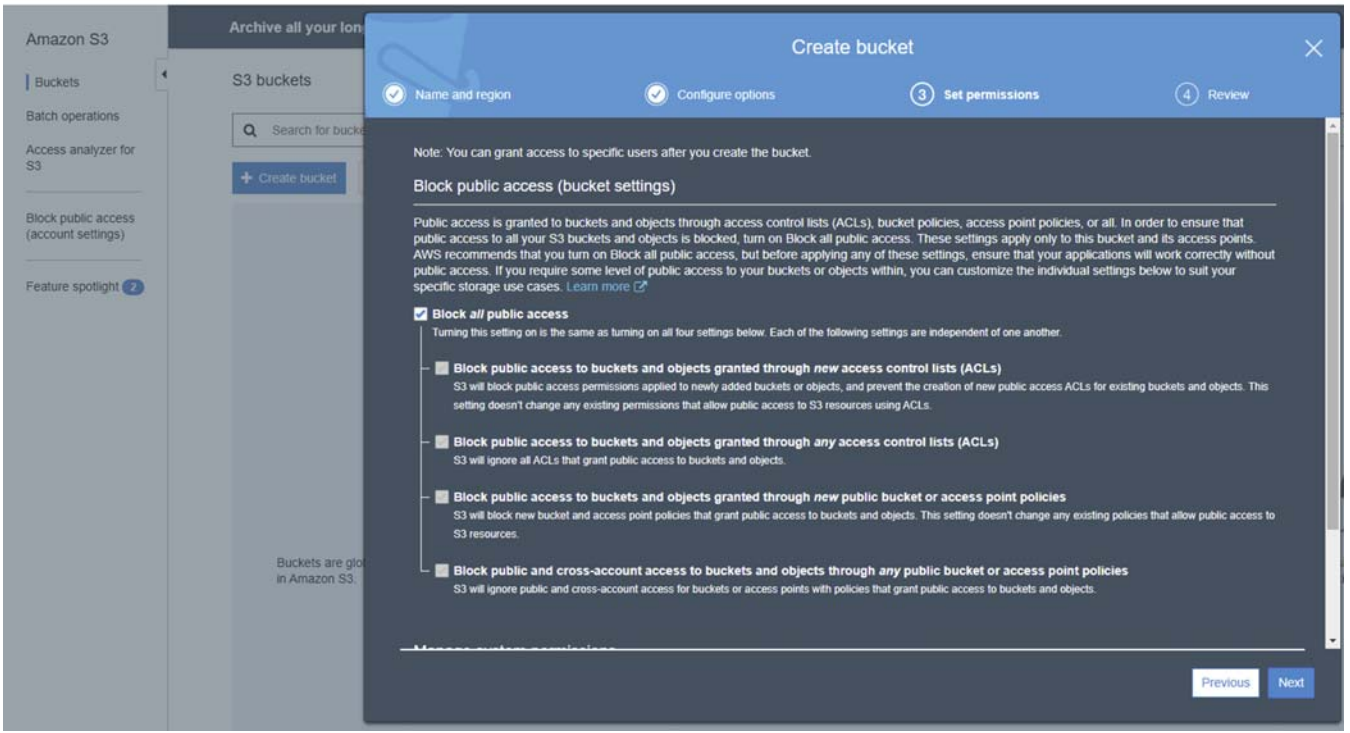


Figure 8.131 Permissions for the AWS bucket.

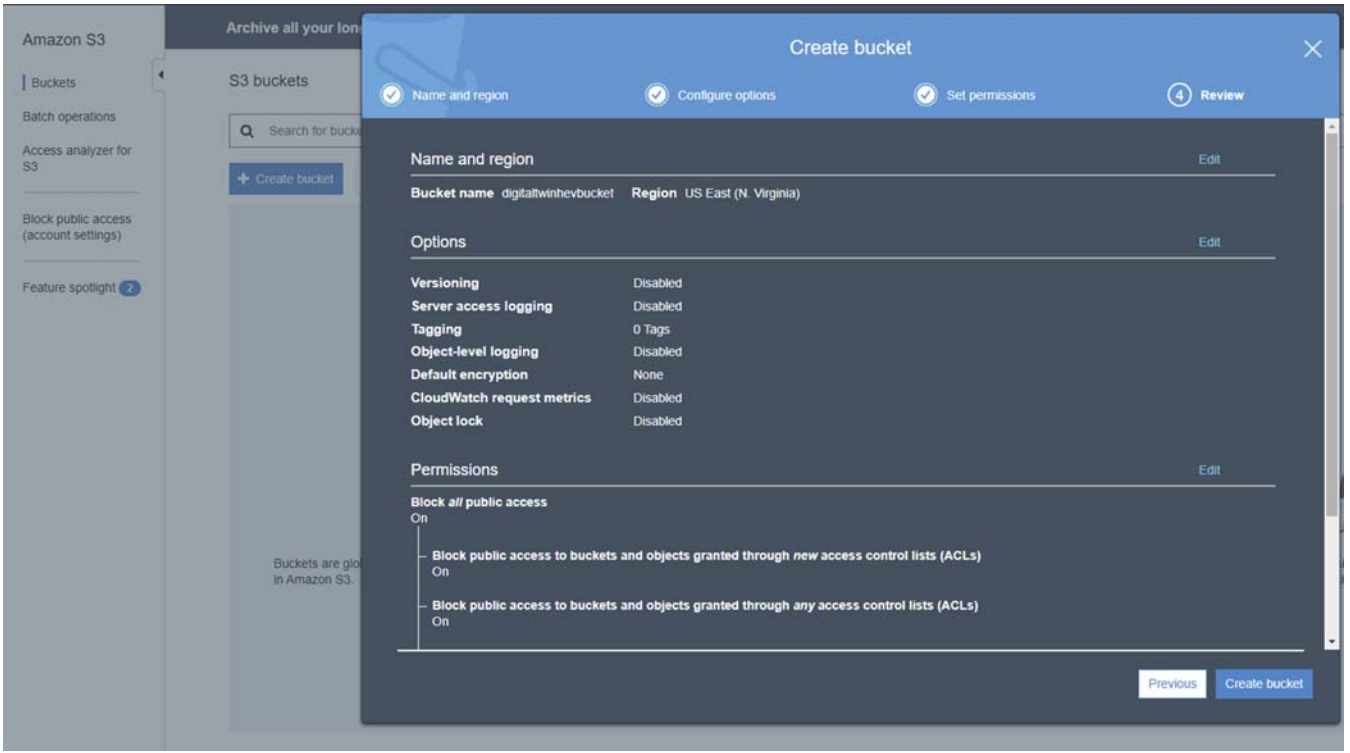


Figure 8.132 Review and finalize S3 bucket creation.

Figure 8.133 No Fault Condition E-mail notifications from the Digital Twin about the Off-BD status.

starts running, we can see from the SNS service we configured in the Lambda function will send Text and E-mail Notifications on the Off BD status based on the Digital Twin prediction, decision logic in the Lambda function. Figs. 8.133 and 8.134 show the E-mail messages received from the Digital Twin notification service, note in this case no failure condition is detected.

42. Next we will test the Engine Throttle Fault condition. From the host computer, run the model **HEV_Simscape_Model_Rasp_Pi_with_MQTT_Throttle_Fault.slx**, which we made earlier, that will run on the Raspberry Pi hardware. As we have seen earlier, this model has the introduced fault condition for the Engine Throttle signal. Also run the Python code **Raspberry_Pi_AWS_IOT_Cloud_Connection.py** on the Raspberry Pi while MATLAB compiles and deploys the model to PI. As the model starts running, we can see from the SNS service we configured in the Lambda function will send Text and E-mail Notifications on the Off BD status based on the Digital Twin prediction, decision logic in the Lambda function. Figs. 8.135 and 8.136 show the E-mail messages received from the Digital Twin notification service, note in this case, since only the HEV system running on Raspberry Pi sees the faulty throttle but not the digital twin, the signals between two systems vary significantly, which will be detected by the Digital Twin Off-BD algorithm.

8.10 Application problem

1. Develop an Off-BD process to detect a battery capacity failure of the HEV system.

Hint: In order to simulate a failed battery, we can change the HEV model battery variable rated capacity variable **HEV_Param.Battery_Det.Rated_Capacity** on the workspace initialization Mat file from 8.1 to a low value and repeat the whole process. For the Digital Twin model that needs to be deployed on to the cloud, keep the Rated Capacity same as original. So the digital twin is not aware of the reduced battery capacity, and it should be able to detect the difference in the signals between actual system and the digital twin predicted system.

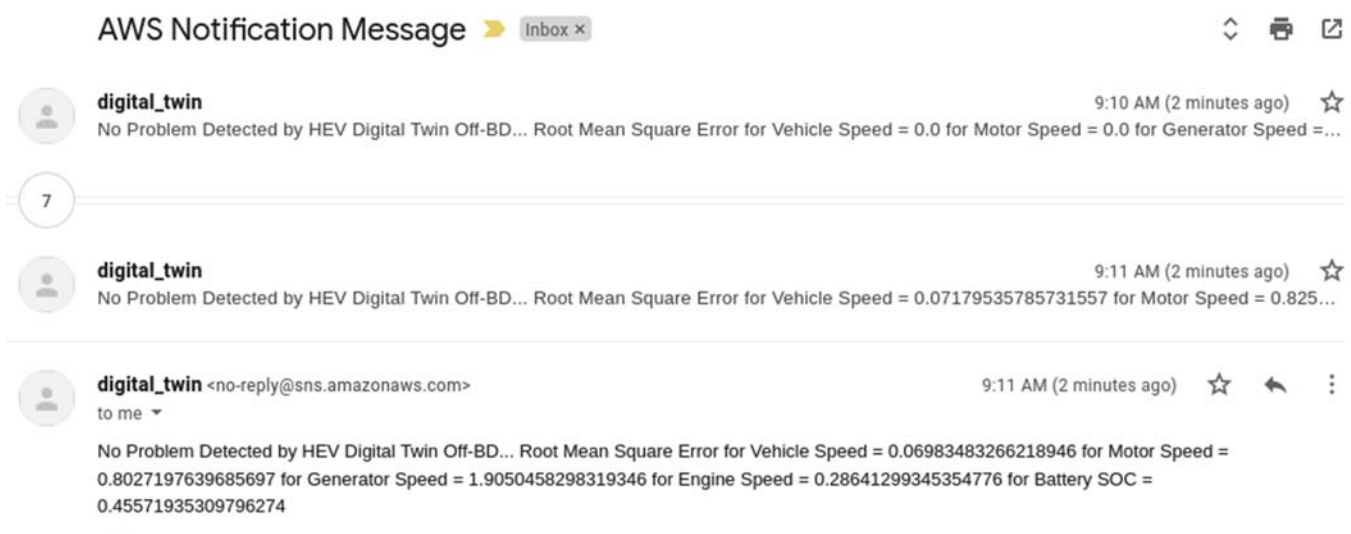


Figure 8.134 No Fault Condition detailed E-mail message information from the Digital Twin about the Off-BD status.

Figure 8.135 Engine Throttle Fault Condition, E-mail Notifications from the Digital Twin about the Off-BD status.

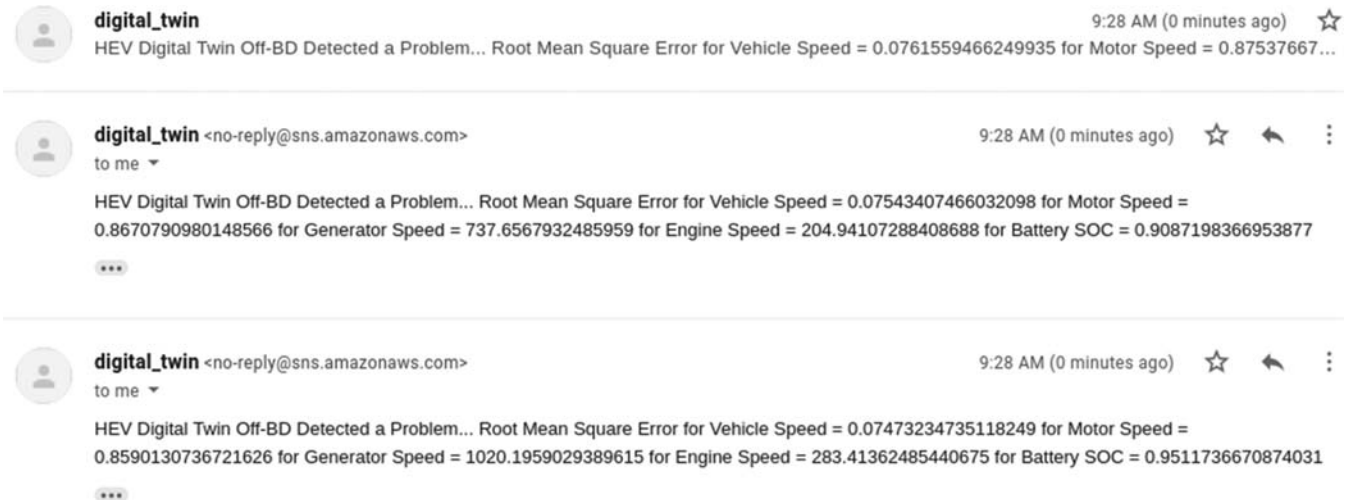


Figure 8.136 Engine Throttle Fault Condition, detailed E-mail message information from the Digital Twin about the Off-BD status.

References

[1] Raspberry Pi 3 B+ Hardware. https://www.amazon.com/CanaKit-Raspberry-Power-Supply-Listed/dp/B07BC6WH7V/ref=sr_1_3?crid=1B02XIHFO3BQK&keywords=raspberry+pi+3b+plus&qid=1576465330&sprefix=raspberry+pi+3%2Caps%2C186&sr=8-3.
[2] Setting up Operating System for Raspberry PI. http://www.mathworks.com/matlabcentral/fileexchange/39354-device-drivers.
[3] Installing Putty on the Host Computer to connect to Raspberry Pi Remotely. https://www.putty.org/.
[4] HEV Matlab® Simulink® Simscape™ Model from Matlab Central File Exchange. https://www.mathworks.com/matlabcentral/fileexchange/28441-hybrid-electric-vehicle-model-in-simulink.
[5] Hybrid Electric Vehicle Types. http://autocaat.org/Technologies/Hybrid_and_Battery_Electric_Vehicles/HEV_Types/.