

zzz

Building and Executing Trusted Execution Environment (TEE) based applications on Azure (and on the Edge)

A starter guide for developers

Version 1.1, September 2019

For the latest information on the Open Enclave (OE) SDK, please see

https://openenclave.io/sdk/

For the latest information on Azure Confidential Computing (ACC), please see https://azure.microsoft.com/en-us/solutions/confidential-compute/

For the latest information on the Confidential Consortium Framework (CCF), please see

https://aka.ms/ccf/

This page is intentionally left blank.

Table of contents

[Notice 3](#_Toc19197963)

[About this guide 4](#_Toc19197964)

[Guide elements 9](#_Toc19197965)

[Guide prerequisites 10](#_Toc19197966)

[Module 1: DC-series VM setup with Open Enclave 11](#_Toc19197967)

[Overview 11](#_Toc19197968)

[Step-by-step directions 12](#_Toc19197969)

[Deploying a DC-series VM on Azure 12](#_Toc19197970)

[Connecting to your DC-series VM 17](#_Toc19197971)

[Using the Open Enclave SDK 20](#_Toc19197972)

[Module 2: TEE-based application development with the Open Enclave SDK 24](#_Toc19197973)

[Overview 24](#_Toc19197974)

[Important concepts 27](#_Toc19197975)

[Terminology 27](#_Toc19197976)

[Enclave interface definition 27](#_Toc19197977)

[Data marshalling 28](#_Toc19197978)

[Azure IoT Platform for the “Intelligent Cloud, Intelligent Cloud” 29](#_Toc19197979)

[Step-by-step directions 31](#_Toc19197980)

[Building a TEE-based Linux application on Intel SGX 31](#_Toc19197981)

[Building a TEE-based Linux application on a simulated ARM TrustZone environment 39](#_Toc19197982)

[Building a TEE-based Linux module on an Edge ARM TrustZone device 51](#_Toc19197983)

[Module 3: Confidential Consortium Framework (CCF) setup and configuration on DC-series VMs 75](#_Toc19197984)

[Overview 75](#_Toc19197985)

[Important concepts 76](#_Toc19197986)

[Blockchain network 76](#_Toc19197987)

[Confidential Computing Framework (CCF) 77](#_Toc19197988)

[Step-by-step directions 78](#_Toc19197989)

[Deploying a 2-node network 79](#_Toc19197990)

[Installing CCF on your nodes 80](#_Toc19197991)

[Sending requests to CCF 84](#_Toc19197992)

[Appendix A. Prerequisites and additional configuration 89](#_Toc19197993)

[Setting up a core Azure IoT environment 89](#_Toc19197994)

[Creating an Azure Container Registry 89](#_Toc19197995)

[Creating an Azure IoT Hub 91](#_Toc19197996)

[Registering an Azure IoT Edge device to your Azure IoT hub 94](#_Toc19197997)

[Installing and starting the Azure IoT Edge runtime on your device 96](#_Toc19197998)

[Appendix B. Support files for the guide 102](#_Toc19197999)

[network.py file 102](#_Toc19198000)

Notice

This guide for developers is intended to illustrate a new way for companies to build and execute so-called Trusted Execution Environment (TEE) based applications using the Microsoft Open Enclave SDK in C and C++. The Microsoft Open Enclave (OE) SDK is available in open source at <https://openenclave.io/sdk/>.

MICROSOFT DISCLAIMS ALL WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, IN RELATION WITH THE INFORMATION CONTAINED IN THIS WHITE PAPER. The white paper is provided "AS IS" without warranty of any kind and is not to be construed as a commitment on the part of Microsoft.

Microsoft cannot guarantee the veracity of the information presented. The information in this guide, including but not limited to internet website and URL references, is subject to change at any time without notice. Furthermore, the opinions expressed in this guide represent the current vision of Microsoft France on the issues cited at the date of publication of this guide and are subject to change at any time without notice.

All intellectual and industrial property rights (copyrights, patents, trademarks, logos), including exploitation rights, rights of reproduction, and extraction on any medium, of all or part of the data and all of the elements appearing in this paper, as well as the rights of representation, rights of modification, adaptation, or translation, are reserved exclusively to Microsoft France. This includes, in particular, downloadable documents, graphics, iconographics, photographic, digital, or audiovisual representations, subject to the pre-existing rights of third parties authorizing the digital reproduction and/or integration in this paper, by Microsoft France, of their works of any kind.

The partial or complete reproduction of the aforementioned elements and in general the reproduction of all or part of the work on any electronic medium is formally prohibited without the prior written consent of Microsoft France.

Publication: August 2019 (updated September)

Version 1.1

© 2019 Microsoft France. All rights reserved

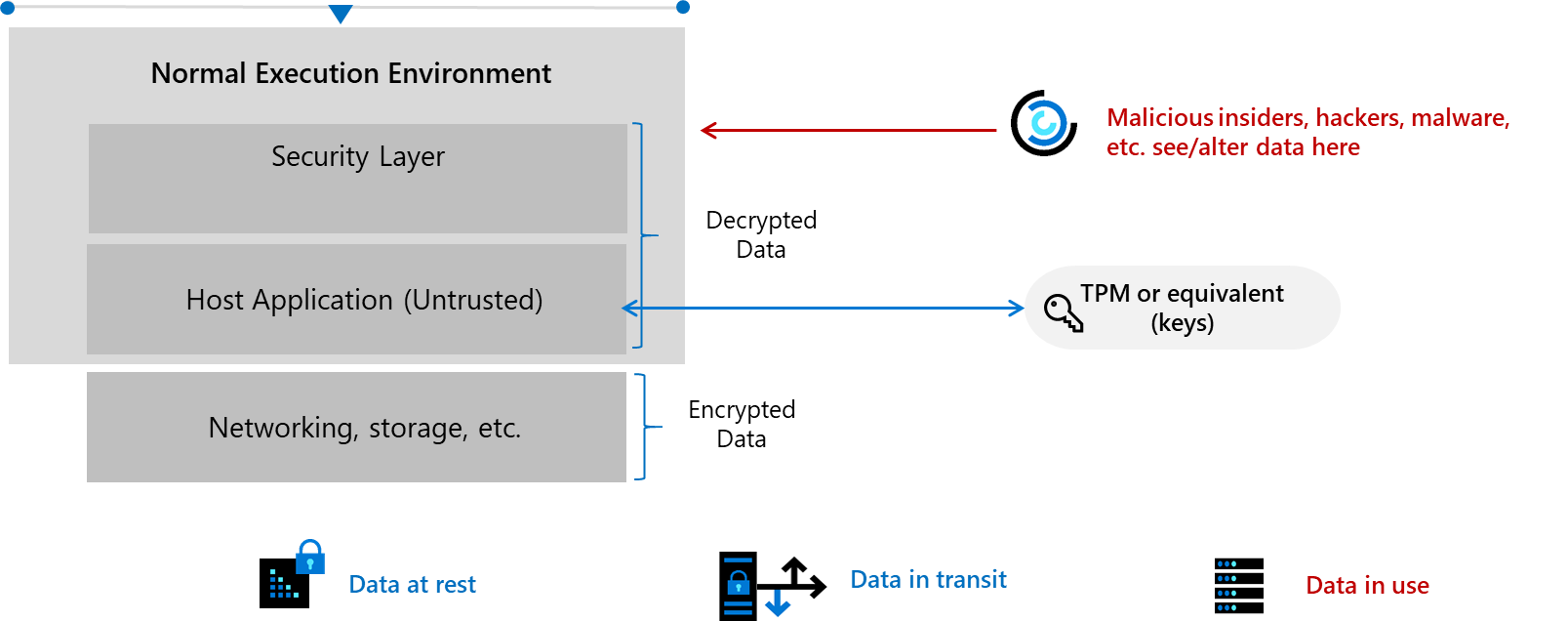
# About this guide

Welcome to the **Building and Executing Trusted Execution Environment (TEE) based applications on Azure** starter guide for developers.

Data can be uploaded encrypted (i.e. **in transit**) to the cloud. Furthermore, in most situations, not to say in all of them, data is stored encrypted (i.e. **at rest**) in the cloud and decrypted on the fly when used or computed by a program. This is both a usual and an adapted way to proceed for the most common data.

But sometimes, protecting data at rest and data in transit is not enough. Data may be indeed too sensible to appear in clear in memory (i.e. **in use**), even if the (virtual) machine and the workload processing data can be considered hardened respectively secured.

Financial data processing constitutes one typical illustration but is far from being the only one.



The specificities of your workload may require protecting data in use confidentiality and integrity from malicious insiders with administrative privilege or direct access, safeguarding against hackers and malware that exploit bugs in the operating system, application, or hypervisor, protecting against third-party access without consent, etc.

By extension, you may also consider the situation where multiple data sources from different organizations that do not necessarily trust each other, or are even competitor, must be combined.

For example, multiples organizations, such as health facilities/institutions and pharmaceutic industries, may have to joint their effort and combine their own respective private patient/health data sources to build, train, evaluate a deep/machine learning model for a better algorithmic outcome without sacrificing data confidentiality: organizations do not see each other’s data sets.

The resulting solution, known as a [privacy-preserving multi-party machine learning](https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/paper.pdf)[[1]](#footnote-2), should allow to fusion sensitive data sources across different organizations while not revealing data to participants or the cloud platform.

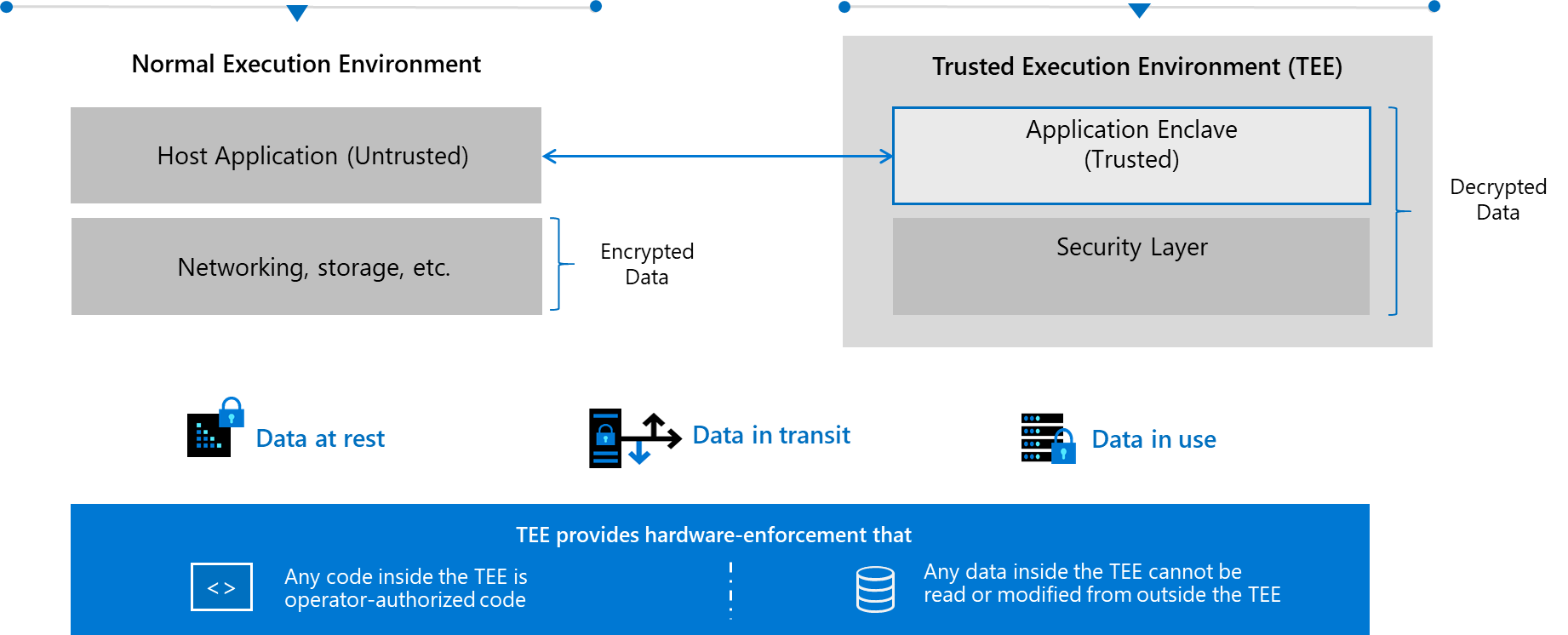
**Note** For more information on privacy-preserving multi-party machine learning, see the various presentations of the one-day workshop [NIPS 2016 Workshop Private Multi‑Party Machine Learning](https://pmpml.github.io/PMPML16/)[[2]](#footnote-3).

In some cases, your sensitive content is the code and not the data. To secure sensitive IP, you may require protect confidentiality and integrity of your code while it’s in use.

Increasing popularity of use cases like the above ones has led to secure compute workloads within the confines of [Trusted Execution Environments](https://en.wikipedia.org/wiki/Trusted_execution_environment)[[3]](#footnote-4) (TEEs).

This concept called Confidential Computing is an ongoing **effort to protect data and/or code throughout its lifecycle at rest, in transit and now in use**.

With the use of TEEs or simply enclaves, you can build applications that protect workloads during computation.



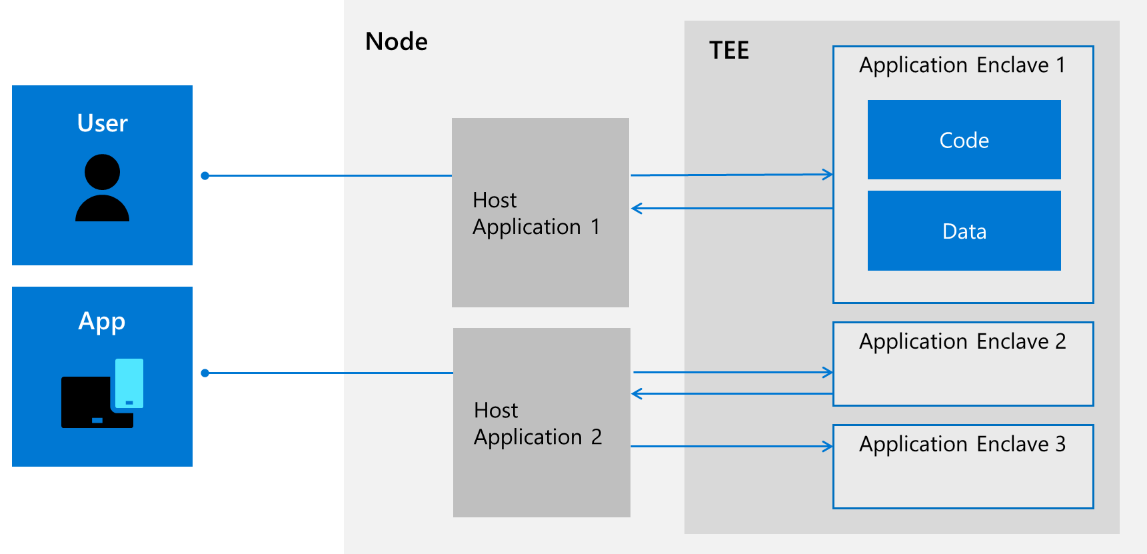
A TEE-based application partitions itself into two components 1) an untrusted component (called the host application) and 2) a trusted component, i.e. a TEE or enclave:

1. The host component runs unmodified on the untrusted operating system (OS), while the trusted component runs within the enclave. It’s a normal user mode application that loads an enclave into its address space before starting to interact with an enclave.
2. The enclave is a secured container provided by a TEE implementation whose memory (text and data) is protected from entities outside the enclave, including the host application, privileged users, and even the hardware: a user (remotely) connected to the machine (even a trusted administrator or the operating system (OS)) can’t see what is running and processing inside this enclave.

These protections allow enclaves to perform secure computations with assurances that secrets will not be compromised. Thus, all functionality that needs to be run in an enclave should be compiled into the enclave binary. The enclave may run in an untrusted environment with the expectation that secrets will not be compromised.

One can obtain a remote attestation of the enclave’s identity, call the enclave’s exposed functions’ interface, but cannot access the code itself within the enclave, the defined variables (and therefore data). In this context, data stays encrypted all the way long from the user point of view: in transit, at rest, and in use. Same considerations may apply for the code itself.

To sump-up, code and date are isolated in encrypted enclaves, preventing snooping or tampering even by the OS or trusted administrators.



TEE-based applications root trust in any secure silicon TEE built on such enclaving technologies like [Intel Software Extension Guard](https://software.intel.com/en-us/sgx)[[4]](#footnote-5) (SGX) - The Intel SGX instruction extension was introduced with 7th Generation Intel Core processor platforms and Intel Xeon processor E3 v5 for data center servers back in 2015 -, [ARM TrustZone](https://www.arm.com/products/silicon-ip-security)[[5]](#footnote-6) (TZ), and embedded Secure Elements using Windows or Linux OSs.

**Note** For more information on the two above TEE technologies, see article [SGX and TrustZone](https://github.com/openenclave/openenclave/blob/feature.new_platforms/new_platforms/docs/sgx_trustzone_arch.md)[[6]](#footnote-7).

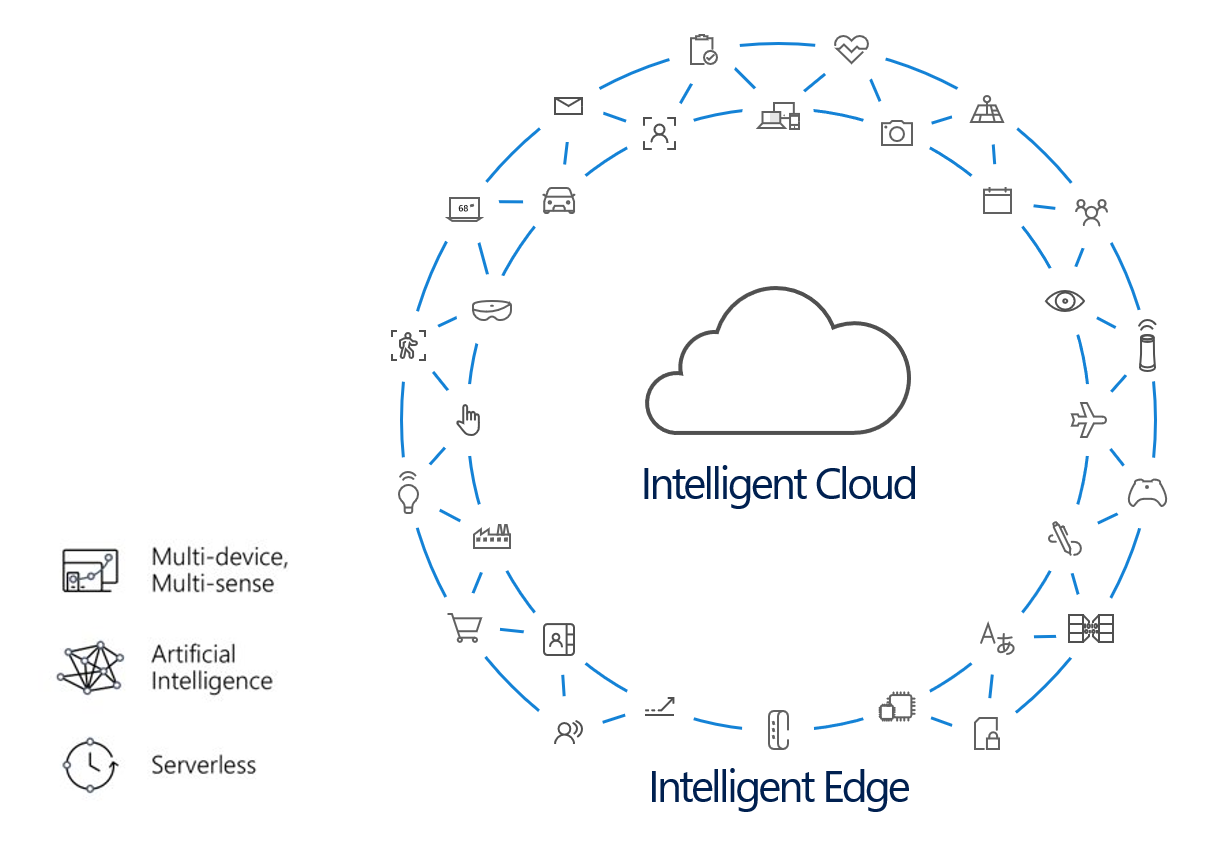
But all of this is still really a low-level work and developing applications above that is really difficult and requires both advanced security expertise and specifics skills.

In this context, Microsoft Research, together with partners, has embarked and invested in a way that simplifies TEE-based application development for all audiences from hardcore hardware security experts to edge and cloud software applications developers, regardless of the underlying enclaving technologies. The effort results in the [Microsoft Open Enclave SDK](https://openenclave.io/sdk/)[[7]](#footnote-8), an open source framework available on GitHub over a year ago under an open source license.

The Open Enclave SDK aims at creating a single unified enclaving abstraction for developer to build applications once that run across multiple TEE architectures, and thus was designed to:

* Make it easy to write and debug code that runs inside TEEs.
* Allow the development of code that’s portable between TEEs.
* Provide a flexible plugin model to support different runtimes and cryptographic libraries.
* Have a high degree of compatibility with existing code.

As such, the Open Enclave SDK provides an open source consistent API surface across enclave technologies and platforms from the cloud to the edge[[8]](#footnote-9).



**Note** Beyond the ongoing shift to the “Intelligent Cloud”, one can observe the inevitable rise of the “Intelligent Edge”. The enormity of data and the evolution of intelligence in (Industrial) Internet of Things ((I)IoT) is inspiring new computational models that distribute compute from cloud to edge and synthesize data to improve business outcomes. As the (I)IoT continues to increasingly accelerate and businesses realize the immense benefits, the next breakthrough capability upon us - or even already here.

Enabling intelligence on edge devices means enabling analytics and insights to happen closer to the source of the (sensitive) data, saving organizations money and simplifying their solutions. The result is rapid growth in patterns such as deploying proprietary algorithms and data including Artificial Intelligence (AI), generating valuable insights including monetization avenues in, and creating direct actions including those controlling downstream critical infrastructure from (I)IoT devices.

Increased functionalities and computing power available on the edge are already changing the way organizations design and build products, from intelligent construction site video surveillance[[9]](#footnote-10), to oil rig maintenance tracking[[10]](#footnote-11). But such patterns are potentially susceptible to exposure of algorithms, data, valuable insights, and tampering of actions unless the computational environment is protected by Confidential Computing.

Confidential compute resources are different for the “Intelligent Cloud” to the “Intelligent Edge” but the development experience for confidential compute workloads targeting the cloud and edge platforms need not be different. In a single development environment, developers can create trusted applications for deployment in cloud enclaves like [Azure Confidential Computing](https://azure.microsoft.com/solutions/confidential-compute/)[[11]](#footnote-12) (ACC) and in TEE enabled Internet of Things (IoT) Edge devices such as the ones running [Azure IoT Edge](https://azure.microsoft.com/en-us/services/iot-edge/)[[12]](#footnote-13) bits.

**Note** For more information on Azure Confidential Computing, see blog post [Introducing Azure confidential computing](https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/)[[13]](#footnote-14) and the webcast [Azure Confidential Computing updates with Mark Russinovich | Best of Microsoft Ignite 2018](https://www.youtube.com/watch?v=Qu6sP0XDMU8)[[14]](#footnote-15).

**Note** For more information on confidential computing with Azure IoT Edge, see blog post [Simplifying confidential computing: Azure IoT Edge security with enclaves – Public preview](https://azure.microsoft.com/en-us/blog/simplifying-confidential-computing-azure-iot-edge-security-with-enclaves-public-preview/)[[15]](#footnote-16) and the webcast [Deep Dive: Confidential Computing in IoT using Open Enclave SDK](https://channel9.msdn.com/Shows/Internet-of-Things-Show/Deep-Dive-Confidential-Computing-in-IoT-using-Open-Enclave-SDK)[[16]](#footnote-17).

In other words, this means that you as a developer can use the same APIs across multiple enclaves, greatly reducing the complexity of following best practices and encouraging organizations to integrate (host) applications with enclaves.

As TEE technology matures and as different implementations arise, the Open Enclave SDK is committed to supporting an API set that allows developers to build once and deploy on multiple technology platforms, different environments from cloud to hybrid to edge, and for both Linux and Windows.

**Important note** As of this writing, and available with this version is the ability to write enclave applications for cloud workloads targeting TEE technology based on Intel SGX hardware technology with a Linux host application

Preview support is also provided for new TEE platforms, namely ARM TrustZone with a Linux host application, and intel SGX with a Windows host application via the Intel SGX SDK. Support for a Windows host application on ARM TrustZone and native Open Enclave support for a Windows host application on Intel SGX will be added in the future.

This broad applicability across different enclave technologies greatly simplifies the work developers must do to protect sensitive data. Furthermore, with accessibility by all security expertise as topmost goal, this integration is laden with features to truly simplify and shorten the journey from idea to at-scale production deployment of secure (intelligent edge) TEE-based applications.

**Note** Microsoft has recently joined partners and the Linux Foundation to create [Confidential Computing Consortium](https://confidentialcomputing.io/)[[17]](#footnote-18) that will be dedicated to defining and accelerating the adoption of confidential computing.

A picture containing object

Description automatically generated

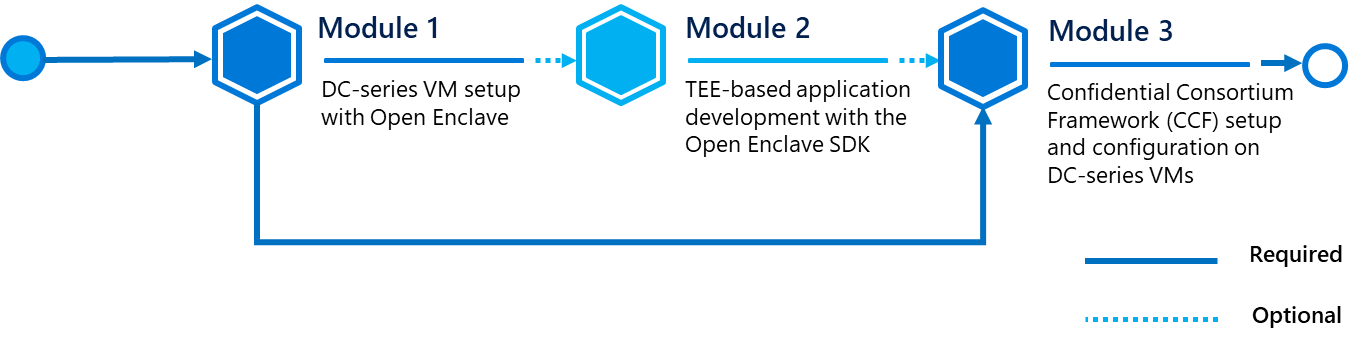
Microsoft will be contributing the Open Enclave SDK to the Confidential Computing Consortium to develop a broader industry collaboration and ensure a truly open development approach. “The Open Enclave SDK is already a popular tool for developers working on Trusted Execution Environments, one of the most promising areas for protecting data in use,” said Mark Russinovich, chief technical officer, Microsoft. “We hope this contribution to the Consortium can put the tools in even more developers’ hands and accelerate the development and adoption of applications that will improve trust and security across cloud and edge computing.”[[18]](#footnote-19)

**In this starter guide, and as its title suggest, we will cover the basics of TEE-based application development.**

You will learn how to create and deploy this new kind of applications on top of Azure Confidential Computing (ACC), using the Open Enclave SDK, or the newly released Confidential Consortium Framework (CCF), which allows to create a trusted distributed blockchain network. (This simplifies consensus and transaction processing for high throughput and confidentiality.)

For that purposes, you’re invited to follow a short series of modules, each of them illustrating a specific aspect of the TEE-based application development.

**Each module within the guide builds on the previous**. You’re free to stop at any module you want, but our advice is to go through all the modules.



**At the end of the starter guide, you will be able to:**

* Understand the Azure Confidential Computing (ACC) offering,
* Instantiate DC-series or other Linux VMs well-suited for trusted applications development,
* Setup a full-pledged development environment with Visual Studio and Visual Studio Code,
* Create new trusted applications or containers using the Open Enclave SDK in C or C++,
* Install the Confidential Consortium Framework (CCF) on DC-series VMs,
* Deploy and manage a multi-node CCF network.

## Guide elements

In the starter guide modules, you will see the following elements:

* **Step-by-step directions**. Click-through instructions - along with relevant snapshots - or links to online documentation for completing each procedure or part.
* **Important concepts**. An explanation of some of the concepts important to the procedures in the module, and what happens behind the scenes.
* **Sample applications, and files**. A downloadable or cloneable version of the project containing the code that you will use in this guide, and other files you will need. **Please go to** **<https://aka.ms/CCDevGuideSamples> on GitHub to download or clone all necessary assets.**

## Guide prerequisites

To successfully leverage the provided code in this starter guide, you will need:

* A [Microsoft account](https://account.microsoft.com/account?lang=en-us)[[19]](#footnote-20).
* An Azure subscription. If you don't have an Azure subscription, create a [free account](https://azure.microsoft.com/free/?WT.mc_id=A261C142F)[[20]](#footnote-21) before you begin.
* A windows 10 local machine.
* A code editor of your choice, such as [Visual Studio](https://visualstudio.microsoft.com/)[[21]](#footnote-22) or [Visual Studio Code](https://code.visualstudio.com/)[[22]](#footnote-23), with C++ for Linux and Open Enclave installed. The related installation and configuration will be further covered later in this guide.
* A terminal console for your Windows 10 local machine, which allows you to remotely connect to a virtual machine (VM) in SSH, such as [PuTTY](https://www.chiark.greenend.org.uk/~sgtatham/putty/)[[23]](#footnote-24), [Git for Windows](https://git-for-windows.github.io/)[[24]](#footnote-25) (2.10 or later).

**Important note** With Git, ensure that long paths are enabled: git config --global core.longpaths true.

**Note** Recent versions of Windows 10 provide OpenSSH client commands to create and manage SSH keys and make SSH connections from a command prompt. For more information, see blogpost [What’s new for the Command Line in Windows 10 version 1803](https://blogs.msdn.microsoft.com/commandline/2018/03/07/windows10v1803/)[[25]](#footnote-26).

# Module 1: DC-series VM setup with Open Enclave

## Overview

This first module of this starter guide will illustrate how to deploy a Confidential Compute (CC) DC-series VM to later leverage the Open Enclave SDK to develop in C and C++ Trusted Execution Environment (TEE) based applications.

In the Azure Platform, the Open Enclave SDK must be indeed installed on top of a Confidential Compute [DC-series](https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-general#dc-series)[[26]](#footnote-27) [[27]](#footnote-28) virtual machine (VM).

For the Azure Confidential Computing (ACC) offering currently in public preview, DC-series VMs are indeed (as of this writing) the (only) type of VMs in Azure that can support Trusted Execution Environment (TEEs). The DC-series is a new family of virtual machines in Azure that are backed by the latest generation of 3.7GHz Intel XEON E-2176G Processor with the intel SGX technology.

With the Intel Turbo Boost Technology, these machines can go up to 4.7GHz.

Two VM sizing options are available for the DC-Series:

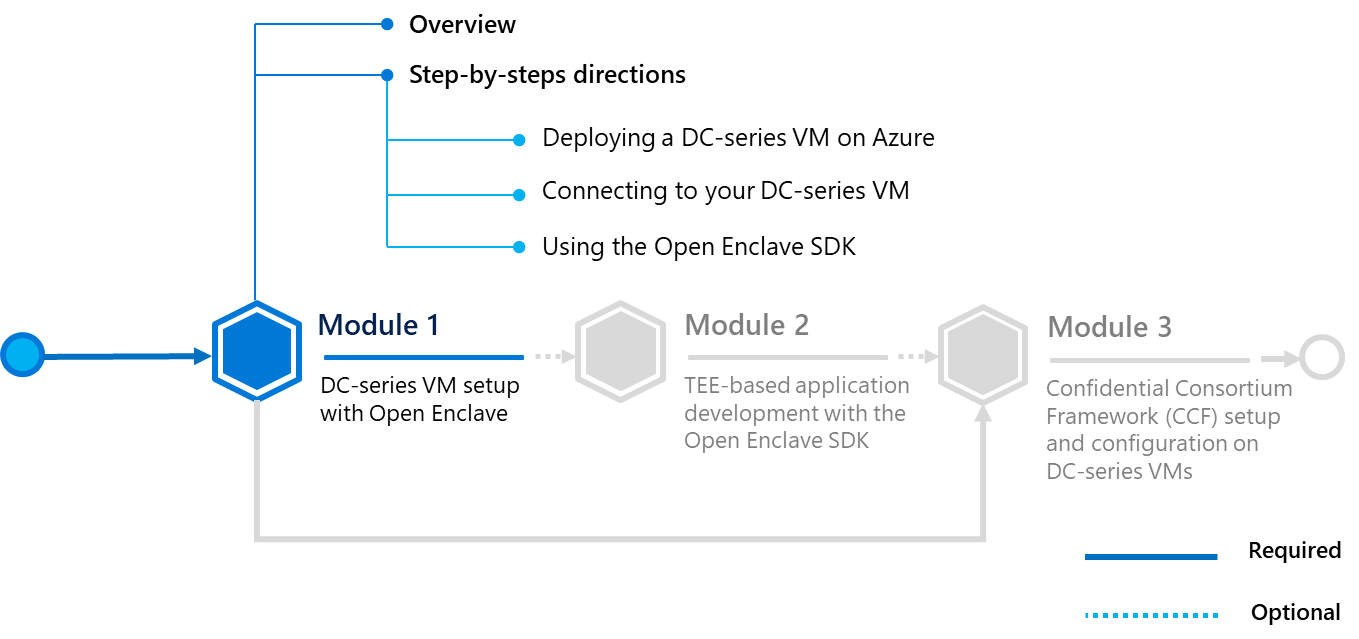
1. Standard\_DC2s with 2 vCPUs and 8 GB of memory,
2. Standard\_DC4s with 4 vCPUs and 16 GB of memory.

DC series instances enable customers to build secure enclave-based applications to protect their code and data while it’s in use.

Currently three operating systems are supported for the DC-series VMs:

1. Windows Server 2016 Datacenter,
2. Ubuntu Server 16.04 LTS,
3. Ubuntu Server 18.04 TLS.

**Note** Additional OS offerings may be supported once the ACC program transitions from public preview to general availability (GA).



## Step-by-step directions

This module covers the following three activities:

1. Deploying a DC-series VM on Azure.
2. Connecting to your DC-series VM.
3. Using the Open Enclave SDK.

Each activity is described in order in the next sections.

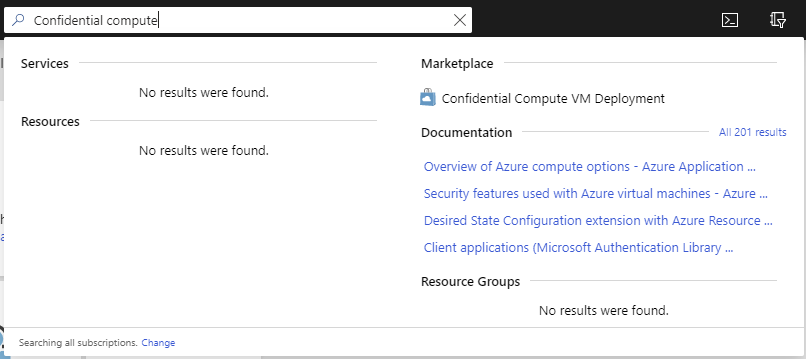
### Deploying a DC-series VM on Azure

DC-series VMs are not listed by default in your **Virtual Machines** tab in the Azure portal. They can instead can be found in the directory in the [Azure Marketplace](https://azuremarketplace.microsoft.com/marketplace/apps/microsoft-azure-compute.confidentialcompute)[[28]](#footnote-29) or by searching “*Confidential Compute*” in the search bar in Azure.

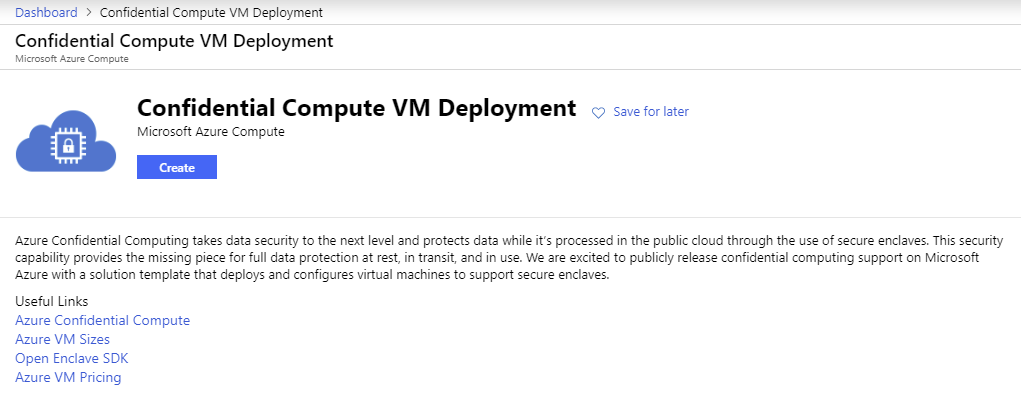
To deploy a DC-series VM in your Azure subscription, perform the following steps:

**Note** For more information, see article [Get Started with Microsoft Azure\* Confidential Computing](https://software.intel.com/en-us/articles/get-started-with-azure-confidential-computing)[[29]](#footnote-30).

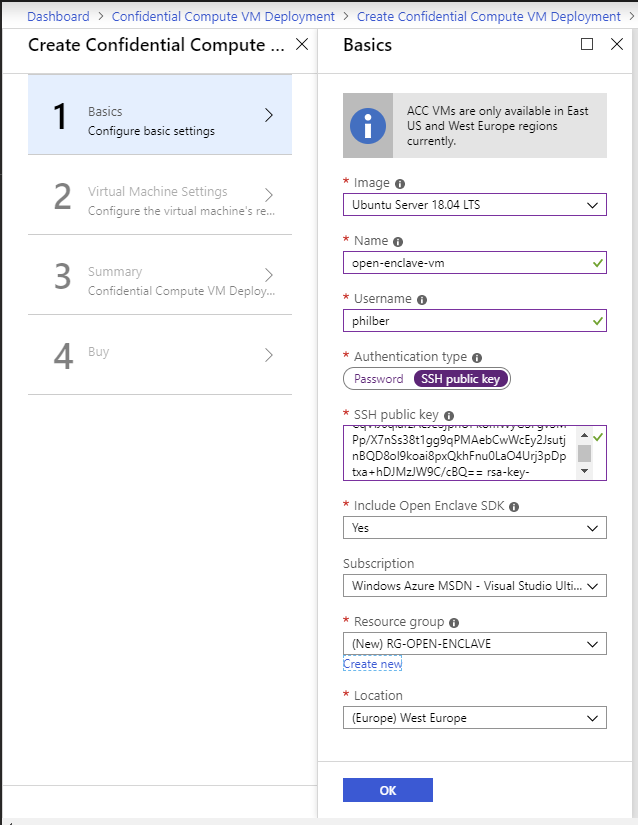
1. Open a browser session and go to the Azure portal at <https://portal.azure.com>.
2. Sign in with your Azure account.
3. First search for “*Confidential Compute*” in the search bar in the Azure portal.



1. Click on **Confidential Compute VM Deployment** under **Marketplace**. You will be then re-directed to the Confidential Compute VM Deployment wizard.



1. Open a browser session and go to the Azure portal at <https://portal.azure.com>.
2. Sign in with your Azure account.

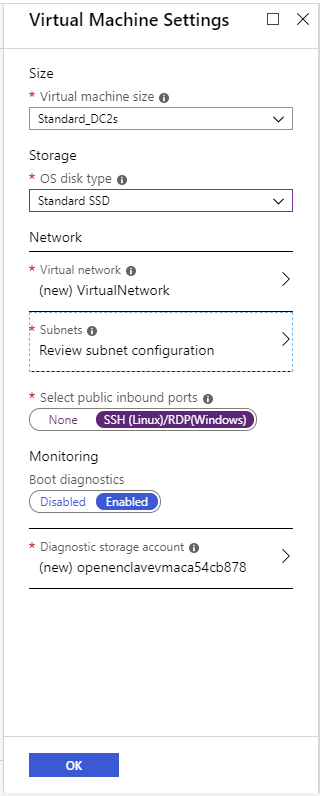


1. Specify the required settings.

|  |  |
| --- | --- |
| Setting | Description |
| Image | Select **Ubuntu Server 18.04 LTS**. |
| Name | Provide the VM a hostname (as a resource, which will be displayed in Azure). |
| Username | Specify a username for the privileged user account of the VM. |
| Authentication type | Select **SSH public key** for stronger authentication to later remotely connect to your VM. |
| SSH public key | Specify a [RSA public key](https://www.ssh.com/ssh/public-key-authentication)[[30]](#footnote-31) in the single-line format beginning with “*ssh-rsa*” - you can use instead the multi-line PEM format -.  **Note** You can generate SSH keys by using [PuTTYGen](https://www.puttygen.com/)[[31]](#footnote-32) on a Windows 10 local machine.    You will need to save the private key to later remotely connect to the VM. For more information on SSH keys, see article [How to use SSH keys with Windows on Azure](https://docs.microsoft.com/en-us/azure/virtual-machines/linux/ssh-from-windows)[[32]](#footnote-33). |
| Include Open Enclave SDK | Ensure that Open Enclave SDK will be included with this VM deployment. **Yes** should be selected. By selecting this option, all the prerequisite build tools will be installed on your behalf.  **Note** For more information on how to install the Open Enclave SDK, see article [Install the Open Enclave SDK (Ubuntu 18.04)](https://github.com/openenclave/openenclave/blob/v0.6.x/docs/GettingStartedDocs/install_oe_sdk-Ubuntu_18.04.md)[[33]](#footnote-34). |
| Subscription | Select your own subscription. If you have more than one, select the most appropriate subscription. |
| Resource group | For public preview, the wizard will only allow deployment to an empty resource group. Create one during VM deployment as follows:   1. Under **Resource group**, click on **Create new**. 2. In the dialog box, name the new resource group and click on **OK**. |
| Location | Select the Microsoft Azure data enter location to which you want to deploy. Choose between **East US** and **West Europe** as this particular type of VM is only available in these locations. Any selection other than these two locations will fail validation checks. |

Click on **OK** to continue to the **Virtual Machine Settings** page.

1. This second page of the form is about more specific settings for the DC-series VM, which include VM size, storage type, and virtual network details. Fill in the **Virtual Machine Settings**.



|  |  |
| --- | --- |
| Setting | Description |
| Size | Azure Confidential Computing (ACC) public preview VMs come in two sizes. Let **Standard\_DC2s** selected by default. |
| Storage | Select your preferred storage type. Select **Standard SSD**. |
| Virtual Network | Configure the (new) virtual network (VNet) where your VM will reside. For simplicity, this starter guide will use the default settings. |
| Subnets | Configure the subnet. Again, this starter guide will use the defaults provided. Go to the configuration sub-menu (click on the red exclamation point) and click on **OK** to accept the default values Click on the option, then click on **OK** below the newly created subnet. |
| Inbound ports | Select **SSH (Linux)/RDP (Windows)** as you’re going to use SSH.  **Important note** The ports will be open for all public inbound traffic from the Internet, posing a serious security issue. For a production environment, it is recommended that you leave selected **none**; after the VM has been created and deployed, configure the VM’s networking inbound port rules to open the required port for a specific IP Address range or enable the Microsoft Azure Security Center Just-in-time VM access. |
| Boot diagnostics | Leave **Enabled** by default. |
| Diagnostic storage account | Leave the default parameter untouched. |

1. Click on **OK** to continue. Validation of your configuration settings will occur on the **Summary** page.
2. If validation has passed, review your configuration and click on **OK** to continue.
3. Before creating and deploying your newly configured VM, carefully read the terms of use and understand any costs associated with the use of Microsoft Azure resources. When you are ready to deploy the VM, click on **Create**. Your DC-series VM will be deployed on Azure.

The completion process will take approximately 10 minutes, at which time you will see a new message in the Microsoft Azure portal notifications tab.

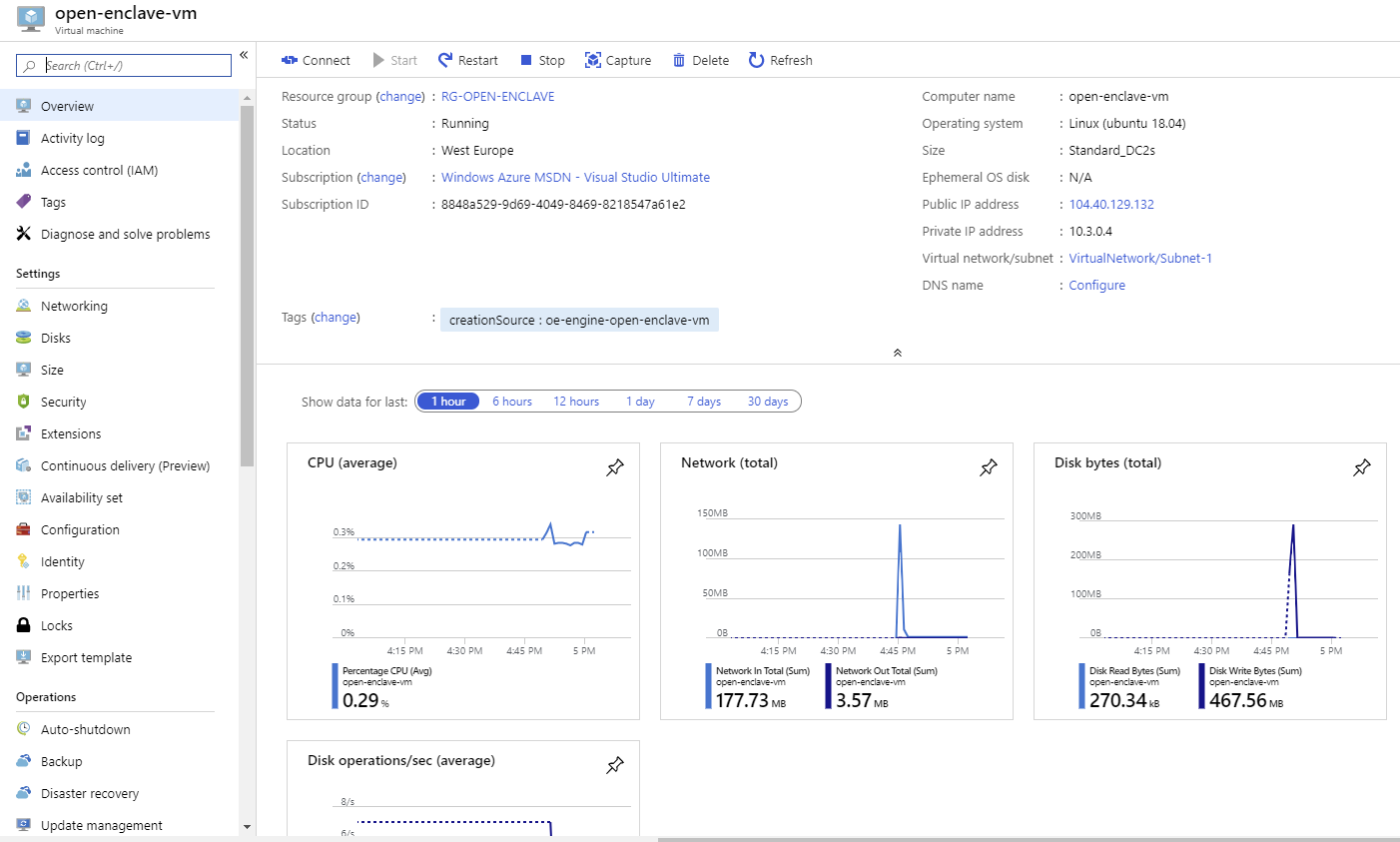
### Connecting to your DC-series VM

Once your VM is online, by using a SSH client of your choice, such as PuTTY, you can test your remote connection to the newly created VM using the administrator credentials provided above. The public IP address of the VM can be found on the VM Networking page.

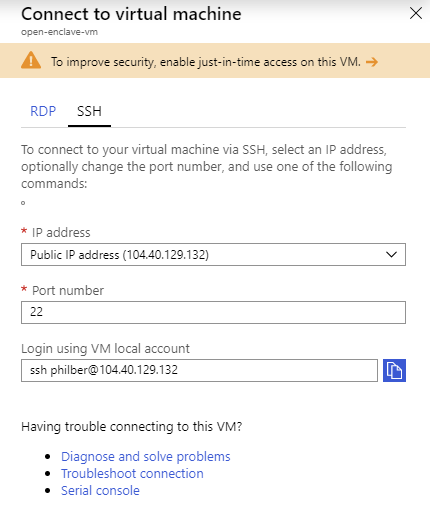
**Note** Depending on your configuration, you may need to configure a proxy in the SSH client to connect to the virtual machine.

To connect to your VM using PuTTY, perform the following steps:

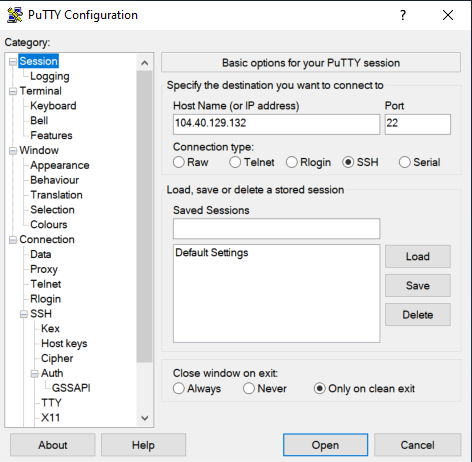
1. From the Azure portal, search for your VM and click on it to display its menu.



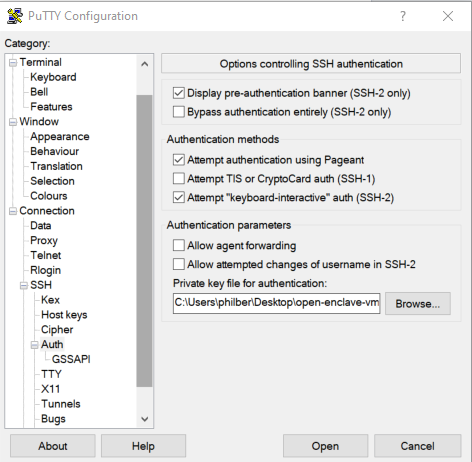
1. Click on **Connect** and make a note of the public IP address under the eponym field.



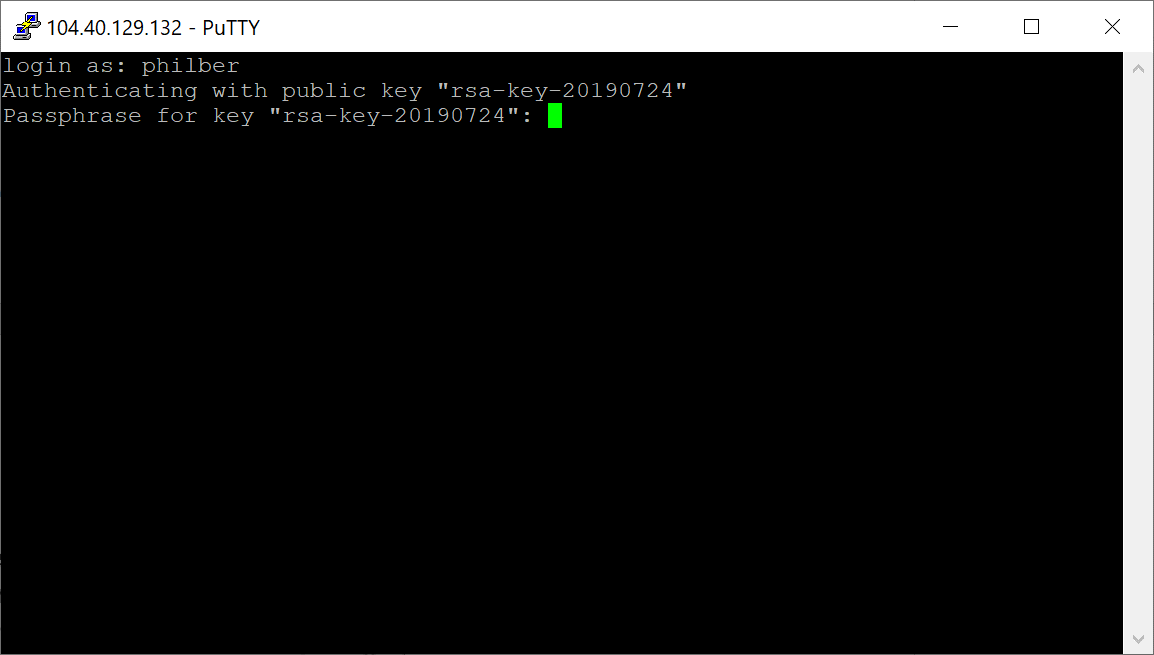
1. Start PuTTY.
2. In the **Category** pane, click on **Session** and complete the following fields:
   1. In the **Host Name** box, paste the public IP address of your VM.
   2. Under **Connection type**, select **SSH**.
   3. Ensure that the **Port** value is 22.



1. In the **Category** pane, expand **Connection**, expand **SSH**, and then click on **Auth**. Complete the followings:
   1. Choose **Browse**.
   2. Select your private key file that corresponds to the public key you specified in the settings for your VM, the .ppk file that you generated for your key pair, and click on **Open**.

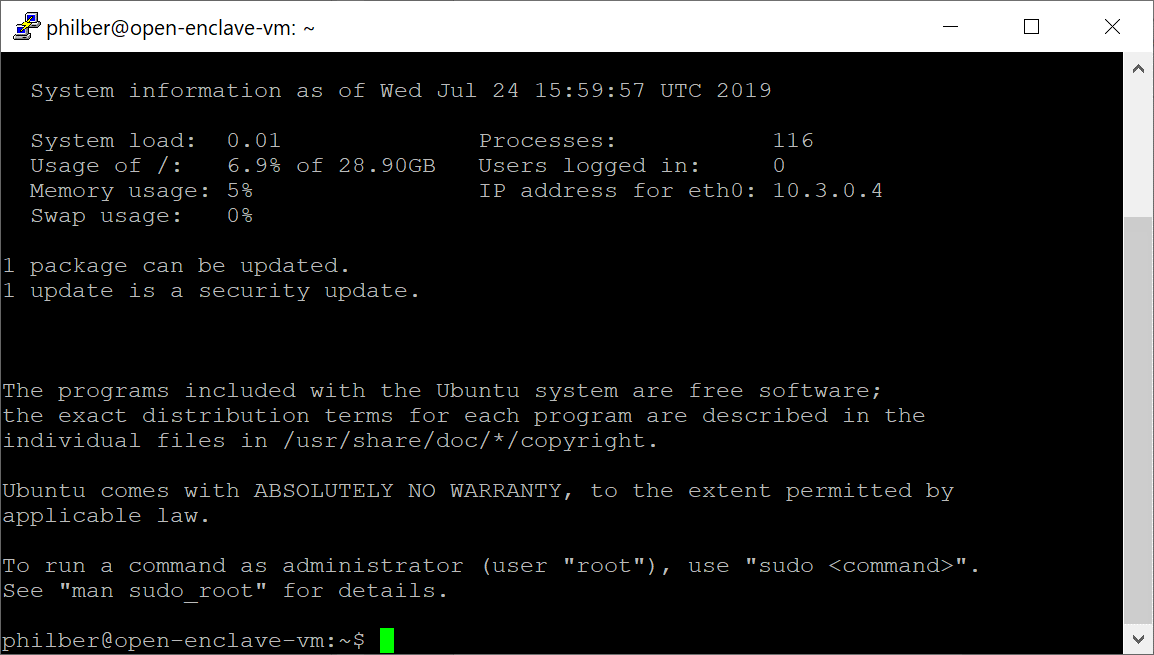


1. Click on **Open**.
2. If this is the first time you have connected to this instance, PuTTY displays a security alert dialog box that asks whether you trust the host to which you are connecting.
3. Choose **Yes**. A window opens.
4. Enter your username and press ENTER.



1. Optionally specify your passphrase if any for your private key.

Et voila! You are now connected to your DC-series VM.



**At this stage, your DC-series VM is ready.**

You can start studying and compiling the Open Enclave SDK sample applications. Let’s see quickly how to begin with.

### Using the Open Enclave SDK

As covered in the introduction, Open Enclave SDK helps you build TEE-based applications and provides you a series of sample applications that demonstrate how to develop enclave applications using Open Enclave APIs.

In your newly created DC-series VM, the Open Enclave SDK is installed to its default directory */opt/openenclave*, which contains the following folders:

|  |  |
| --- | --- |
| Path | Description |
| bin | All the developer tools for developing, debugging and signing TEE-based applications using the Open Enclave SDK. |
| include/openenclave | Open Enclave runtime headers for use in your application enclave (*enclave.h*) and its host application (*host.h*) |
| include/openenclave/3rdparty | Headers for libc, libcxx and mbedlts libraries for use inside the enclave. |
| lib/openenclave/cmake | Open Enclave SDK CMake package for integration with your CMake projects. For example, and as illustrated in the module, Visual Studio 2017/2019 supports CMake projects for cross-platform builds and there are extensions for Visual Studio Code. |
| lib/openenclave/enclave | Libraries for linking into the enclave, including the libc, libcxx and mbedtls libraries for Open Enclave. |
| lib/openenclave/host | Library for linking into the host application process of the enclave. |
| lib/openenclave/debugger | Libraries used by the gdb plug-in for debugging enclaves. |
| share/pkgconfig | Pkg-config files for header and library includes when building TEE-based applications using the Open Enclave SDK. |
| share/openenclave/samples | Sample applications’ code showing how to use the Open Enclave SDK. |

**Note** For more information, see article [Using the Open Enclave SDK](https://github.com/openenclave/openenclave/blob/master/docs/GettingStartedDocs/using_oe_sdk.md)[[34]](#footnote-35).

As far as the sample applications are concerned, it's advised to go through them in the order listed hereafter to progressively familiarize yourself with the Open Enclave SDK.

|  |  |
| --- | --- |
| Sample Application | Description |
| [HelloWorld](https://github.com/openenclave/openenclave/blob/master/samples/helloworld/README.md)[[35]](#footnote-36) | Minimum code needed for an Open Enclave application. Help understand the basic components a TEE-based application with the Open Enclave SDK. |
| [File-Encryptor](https://github.com/openenclave/openenclave/blob/master/samples/file-encryptor/README.md)[[36]](#footnote-37) | Show how to encrypt and decrypt data inside an enclave. |
| [Data-Sealing](https://github.com/openenclave/openenclave/blob/master/samples/data-sealing/README.md)[[37]](#footnote-38) | Introduce the Open Enclave sealing and unsealing features. |
| [Remote Attestation](https://github.com/openenclave/openenclave/blob/master/samples/remote_attestation/README.md)[[38]](#footnote-39) | Explain how the Open Enclave attestation works.  Demonstrate an implementation of such a remote attestation between two enclaves running on different machines. |
| [Local Attestation](https://github.com/openenclave/openenclave/blob/master/samples/local_attestation/README.md)[[39]](#footnote-40) | Explain the concept of Open Enclave local attestation.  Demonstrate an implementation of local attestation between two enclaves on the same machine. |
| [Attested TLS](https://github.com/openenclave/openenclave/blob/master/samples/attested_tls/README.md)[[40]](#footnote-41) | Explain what an Attested TLS channel is. See article [What is an Attested TLS channel](https://github.com/openenclave/openenclave/blob/master/samples/attested_tls/AttestedTLSREADME.md#what-is-an-attested-tls-channel)[[41]](#footnote-42).  Demonstrate an implementation for how to establish an Attested TLS channel between i) two enclaves, and ii) one non-enclave client and an enclave. |

**Note** For a detailed explanation of each sample application, see article [Open Enclave SDK Samples](https://github.com/openenclave/openenclave/blob/master/samples/README.md)[[42]](#footnote-43).

All the above sample applications that come with the Open Enclave SDK installation share a similar directory structure with underneath a *host* folder for the host application and an *enclave* folder for the enclave itself, along with build instructions for two different build systems: one using GNU Make and pkg-config, the other using CMake.

However, writing files under the */opt* folder, where the Open Enclave is installed, is not allowed unless the command is running in the context of the superuser, i.e. sudo.

To build the above sample applications and avoid this sudo requirement, perform the following steps:

1. Connect to your DC-series VM as per previous activity.
2. You may want to first copy the sample applications to a user directory of your choice then build and run on those local copy. Copy them to your home directory, for example in a folder *mysamples*:

sudo cp -r /opt/openenclave/share/openenclave/samples ~/mysamples

1. Change the owner of the sample applications’ code directory from root to your account:

sudo chown -R philber ~/mysamples/

1. Before building any sample application code, you first need to source the file *openenclaverc* to setup environment variables for the Open Enclave SDK for ease of development:
   * Open Enclave SDK *pkgconfig* folder to PKG\_CONFIG\_PATH,
   * Open Enclave SDK *bin* folder to PATH.

# Copyright (c) Microsoft Corporation. All rights reserved.

# Licensed under the MIT License.

# Update PKG\_CONFIG\_PATH.

export PKG\_CONFIG\_PATH=${PKG\_CONFIG\_PATH}:/opt/openenclave/share/pkgconfig

# Set CMake Config-package path

export OpenEnclave\_DIR=/opt/openenclave/lib/openenclave/cmake

# Update PATH.

export PATH=${PATH}:/opt/openenclave/bin

The file *openenclaverc* is located in the folder *share/openenclave* of the Open Enclave SDK installation directory. Initialize the Open Enclave build environment.

. /opt/openenclave/share/openenclave/openenclaverc

**Note** You can use . in Bash to source.

1. Go to the sample applications’ directory:

cd ~/mysamples

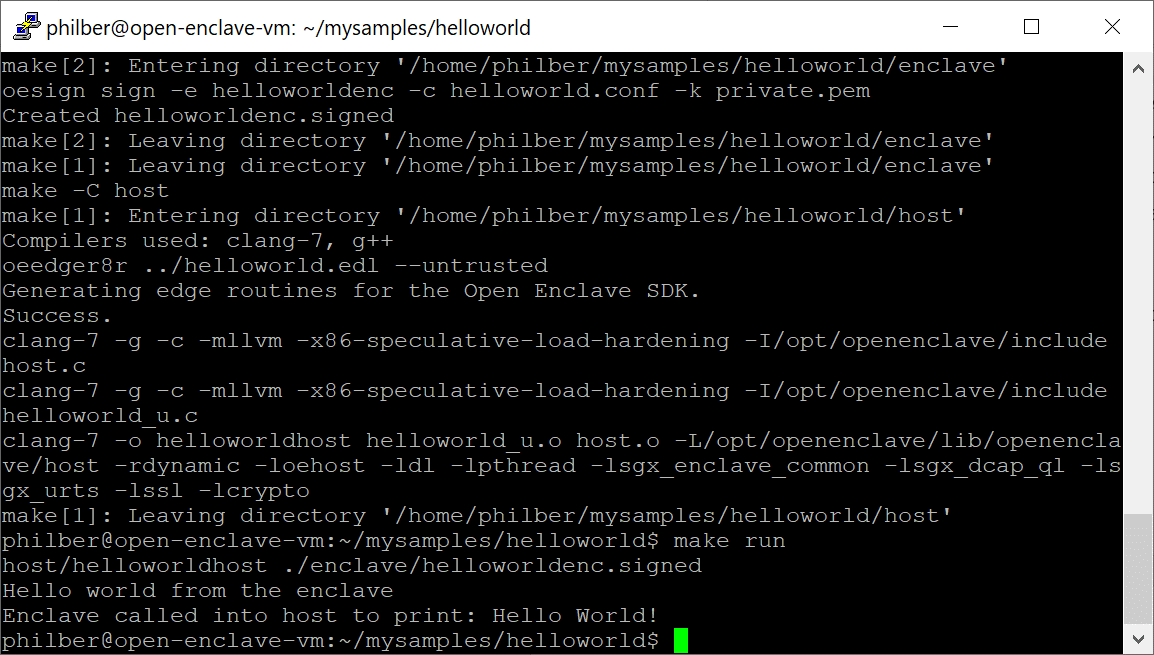
1. To build the sample applications for example using GNU Make, go into each subfolder and build and execute only an individual project, for example:

cd helloworld

make build

make run

Verify that the sample application runs successfully. You should see the following messages in the console: Hello world from the enclave and Enclave called into host to print: Hello World!



Enjoy your first exploration of the Azure Confidential Computing (ACC) and the Open Enclave SDK.

Studying and the sample applications’ code will help you understand how to develop enclaves using the Open Enclave SDK.

**This is also the purpose of the next module.**

# Module 2: TEE-based application development with the Open Enclave SDK

## Overview

This second module of this guide will illustrate the basics on how to develop Trusted Execution Environment (TEE) based application (or containers) for Linux with the Open Enclave SDK in C and C++ and will help you understand the key characteristics of such applications (or containers) in the context of the so-called “Intelligent Cloud, Intelligent Edge”.

For that purpose, this module will more specifically cover the following types of TEE-based applications (or containers):

1. A Linux host app and an enclave on Intel SGX.
2. A Linux host app and an enclave on a (simulated) ARM TrustZone environment.
3. An enclave-based Linux container on an Edge ARM TrustZone device.

**Note** This second module is partially inspired by the webcast [Deep Dive: Confidential Computing in IoT using Open Enclave SDK](https://channel9.msdn.com/Shows/Internet-of-Things-Show/Deep-Dive-Confidential-Computing-in-IoT-using-Open-Enclave-SDK)[[43]](#footnote-44) available on the [IoT Show](https://aka.ms/IoTShow)[[44]](#footnote-45) on Microsoft Channel 9.

As stated in the **Guide prerequisites**, you will use a local Windows 10 machine to develop and cross-build such applications (or containers) for Linux.

In the first above case, you will unsurprisingly use the DC-series VM running Ubuntu 18.04 you have setup in the previous module as a build machine: this machine that is SGX-capable will thus be used as your remote compiler and linker for your application.

**Important note** A non-SGX machine can still be used in simulation mode.

**Note** For a system to be considered to be SGX enabled, it must meet all the following three conditions: i) the CPU in the system must support the Intel SGX extension, ii) the system BIOS must support Intel SGX control, and iii) Intel SGX must be enabled in the BIOS. For more information, see article [Determine the SGX support level](https://github.com/openenclave/openenclave/blob/master/docs/GettingStartedDocs/SGXSupportLevel.md)[[45]](#footnote-46).

Visual Studio will be used on the local Windows 10 machine for the integrated development environment (IDE) and you will need to configure it with the address (or name) of your Linux machine for cross-building the TEE-based application. This machine can also be any of the followings running Ubuntu 18.04 or Ubuntu 16.04 (64-bit) and the Open Enclave SDK:

* A remote (SGX-capable) Linux machine,
* A Linux VM running on your local Windows 10 machine.

**Note** To install the Open Enclave SDK, see articles [Install the Open Enclave SDK (Ubuntu 18.04)](https://github.com/microsoft/openenclave/blob/master/docs/GettingStartedDocs/install_oe_sdk-Ubuntu_18.04.md)[[46]](#footnote-47) or [Install the Open Enclave SDK (Ubuntu 16.04)](https://github.com/microsoft/openenclave/blob/master/docs/GettingStartedDocs/install_oe_sdk-Ubuntu_16.04.md)[[47]](#footnote-48).

In the second case, for simplicity, you will also leverage your DC-series VM running Ubuntu 18.04 as a Linux machine.

Similarly, this machine can also be any of the followings running Ubuntu 18.04 and the Open Enclave SDK:

* A remote Linux machine,
* A Linux VM running on your local Windows 10 machine,
* Or the Windows Subsystem for Linux (WSL) environment on your local Windows 10 machine.

**Note** For information about Windows Subsystem for Linux (WSL), see the [WSL documentation](https://aka.ms/wsldocs)[[48]](#footnote-49), or the [WSL learning resources page](https://aka.ms/learnwsl)[[49]](#footnote-50).

Moreover, Visual Studio Code will be used instead for the IDE. In addition, the [Visual Studio Code Remote Development Extension Pack](https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensionpack)[[50]](#footnote-51) will be used to connect in SSH to your Linux machine, or to integrate with WSL 2, for cross-building the TEE-based application.

Eventually, in the third and last case, a real edge device would typically be one like the [Scalys TrustBox Edge](https://scalys.com/trustbox-industrial/)[[51]](#footnote-52) device, which is an industrial grade, tamper-resistant secured [Azure IoT Edge](https://azure.microsoft.com/en-us/services/iot-edge/)[[52]](#footnote-53) device optimized for confidential computing using TEEs. Azure IoT Edge is an implementation of a secure Intelligent Edge platform that is operating system, processor architecture, and hardware agnostic (see section § **Azure IoT Platform for the “Intelligent Cloud, Intelligent Cloud”** below).



This device comes pre-loaded with Open Enclave SDK and is Azure IoT Edge ready so you can immediately focus on your TEEs-based workloads. It is powered by a [NXP QorIQ LS1012A](https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/layerscape-communication-process/qoriq-layerscape-1012a-low-power-communication-processor:LS1012A)[[53]](#footnote-54) single core 64-bit ARM, and as such, supports the aforementioned ARM TrustZone v8 architecture.

However, to lessens the barrier for the walkthrough, and not requiring you to buy some hardware, you will use instead an [Edge Linux VM](https://azuremarketplace.microsoft.com/en-us/marketplace/apps/microsoft_iot_edge.iot_edge_vm_ubuntu)[[54]](#footnote-55) running Ubuntu 16.04 as a (virtual) Edge device.

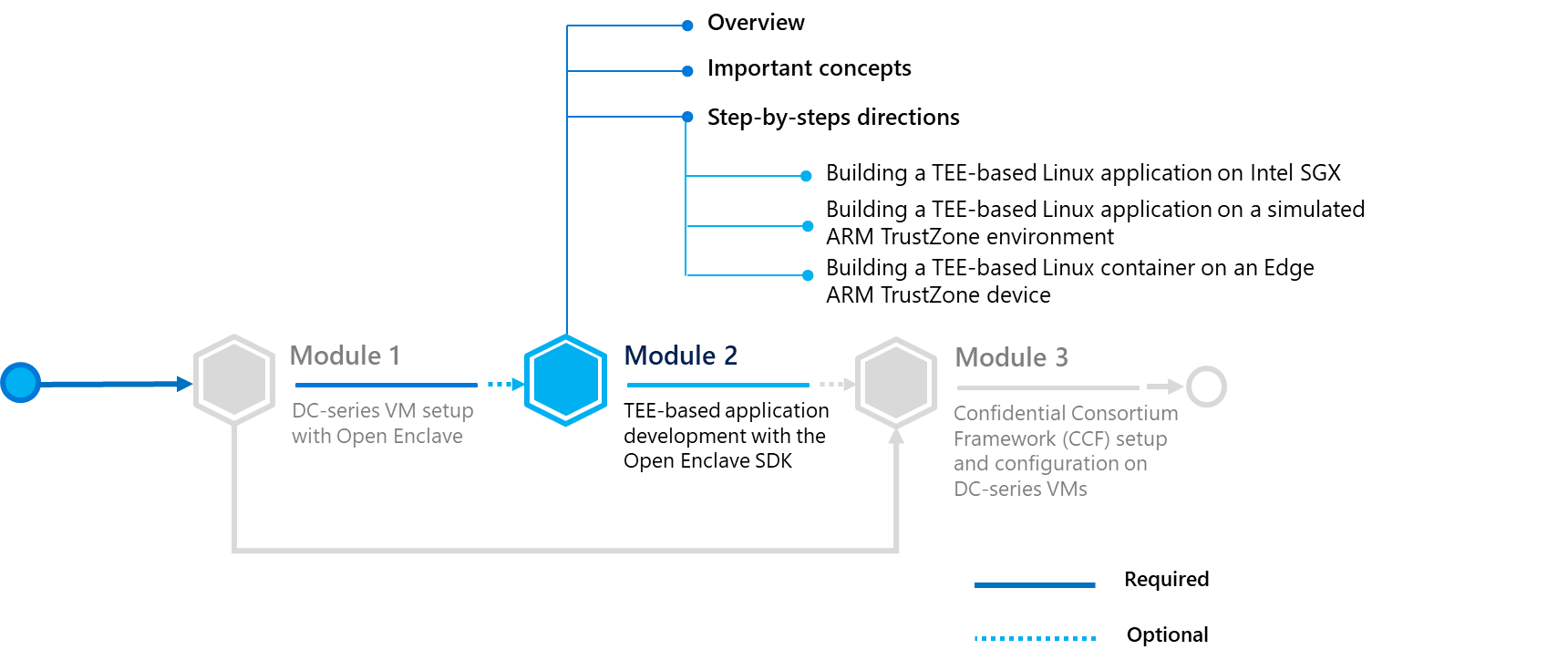
As far as the development machine is concerned, you can use:

* Your local Windows 10 machine,
* A remote Linux machine,
* A Linux VM running on your local Windows 10 machine.

As long as your development machine can run Linux containers: a container engine on the development machine is indeed required for Linux devices.

To limit the required resources and for simplicity, you will also use the same above Edge Linux VM but one must points out both the difference and the distinction between the edge device and the development machine that are normally speaking separate ones.

Likewise, Visual Studio Code will also be used for the IDE as well as the Visual Studio Code Remote Development Extension Pack to connect in SSH to your edge Linux VM for cross-building the TEE-based container.



Before dive in, let’s consider some important concepts regarding:

* Some of the data marshalling and unmarshalling principles and how to transfer control between the host application and the secure enclave,
* As well as the [Azure IoT](https://azure.microsoft.com/en-us/overview/iot/)[[55]](#footnote-56) platform for the so-called “Intelligent Cloud, Intelligent Edge”.

## Important concepts

### Terminology

Let’s clarify some of the commonly used terminology in the rest of this module:

* **Untrusted**. refers to code or construct that runs in the host application environment outside the enclave.
* **Trusted**. refers to code or construct that runs in the Trusted Execution Environment (TEE) inside the enclave.
* **ECALL**. A call from the host application into an interface function within the enclave.
* **OCALL**. A call made from within the enclave to the host application.
* **Generated code**.refers to code automatically generated by the Open Enclave edger8r tool through the definition/use of enclave interface definition files; i.e. EDL files (see below):
  + Boilerplate code in the normal execution environment that executes outside the enclave environment and performs functions such as loading and manipulating an enclave (e.g. destroying an enclave), and making calls (ECALLs) to an enclave and receiving calls (OCALLs) from an enclave.
  + Boilerplate code in the trusted execution environment that executes within the enclave environment and performs functions such as receiving calls (ECALLs) from the host application and making calls outside (OCALLs) the enclave, and managing the enclave itself.

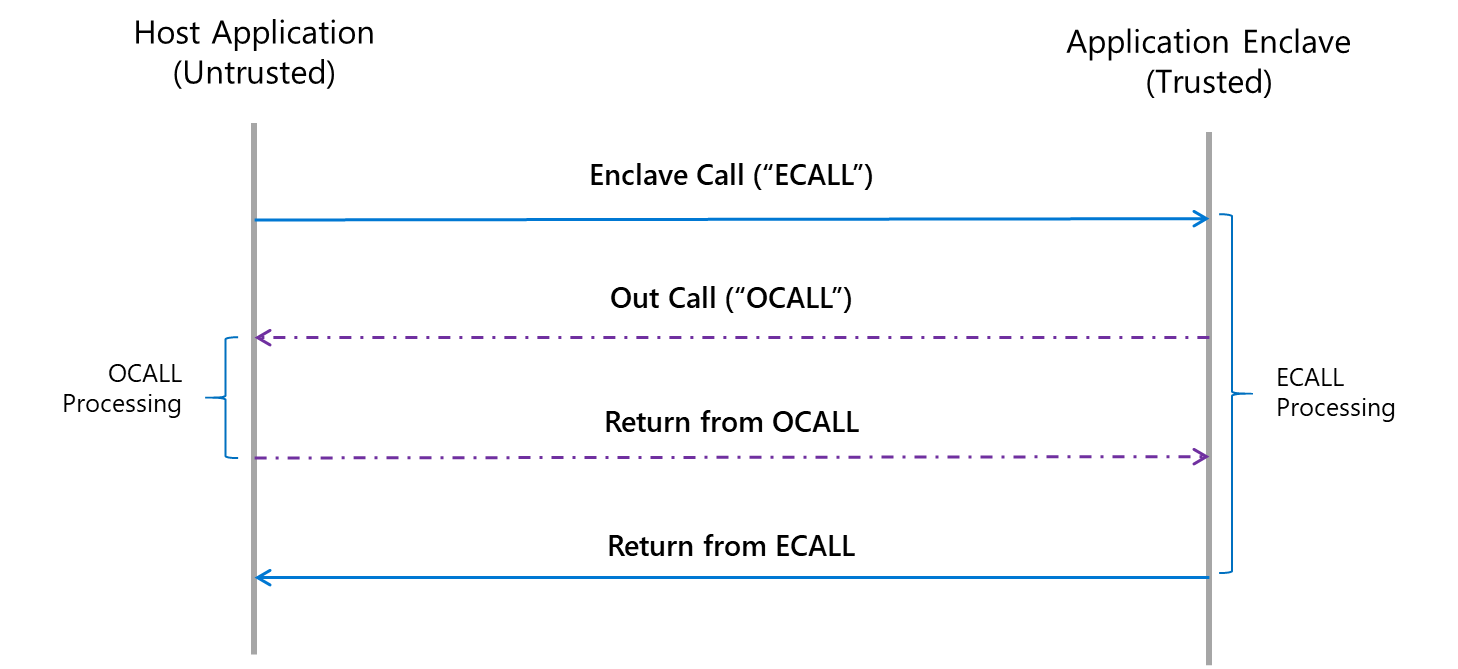
**Important note** For information about the Open Enclave edger8r tool, see article [Getting started with the Open Enclave edger8r](https://github.com/openenclave/openenclave/blob/feature.new_platforms/docs/GettingStartedDocs/Edger8rGettingStarted.md)[[56]](#footnote-57).

### Enclave interface definition

As introduced above, the interface between the host application (untrusted) and the application enclave (trusted) is defined using the Enclave Definition Language or EDL. The EDL file defines the interfaces and data types the enclave will support.

The EDL file is used to define:

1. How a host application calls in to an enclave to request a secure service, i.e. an Enclave CALL or ECALL
2. And how an enclave calls into its host application to request an unsecured service, i.e. an Out CALL or OCALL.



There are thus two parts to an EDL file: the trusted section that defines the ECALLS whereas the untrusted section defines the OCALLS. While an ECALL defines entry point into the enclave, the OCALL defines the transfer of control from inside the enclave to the host application to perform system calls and other I/O operations. OCALLS could also be used in cases where the enclave needs to transfer data back to the host application.

**Important note** An SGX enabled application should always have at least one public ECALL to enter the enclave. OCALLs are optional.

Definitions must be described using the [EDL file syntax](https://software.intel.com/en-us/sgx-sdk-dev-reference-enclave-definition-language-file-syntax)[[57]](#footnote-58).

Furthermore, the same EDL file is used to define the interface between the host application and the enclave, and regardless of whether the enclave is:

1. An Intel SGX enclave,

-or-

1. An Open Portable Trusted Execution Environment (OP-TEE) Trusted Application (TA) based on ARM TrustZone to provide isolation of the TEE from the rich OS in hardware.

An EDL file may include other EDL files and is processed using the Open Enclave edger8r tool, i.e. ooedger8r, which generates boilerplate code for you.

### Data marshalling

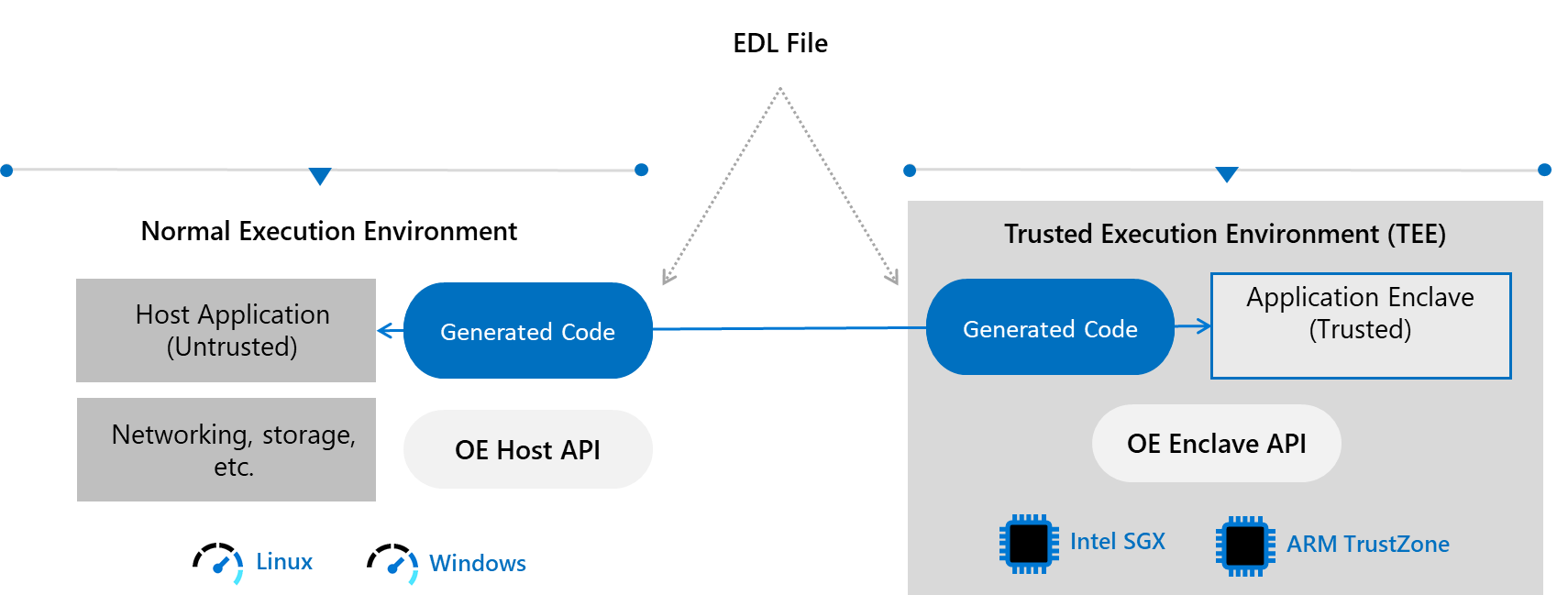
Calling into and out of enclaves is done through special methods that switch into and out of the enclave, along with the marshaling of parameters that are passed into these functions. A lot of the code necessary to handle these calls and parameter marshaling are common to all function calls. Marshaling parameters from the host application to the enclave for security purposes, and in doing so, also helps to mitigate certain processor vulnerabilities such as [Meltdown and Spectre](https://meltdownattack.com/)[[58]](#footnote-59).

The aforementioned Open Enclave edger8r helps to define these special functions through the use of EDL file(s) and then generates boilerplate code for you. It more specifically generates five files as follows:

* A source (*<host>\_u.c*) and a header file (*<host>\_u.h*) to be included by the host application (when ooedger8r is executed with the --untrusted flag),
* Conversely, a source (*<enclave>\_t.c*) and a header file (*<enclave>\_t.c*) to be included by the enclave (when ooedger8r is executed with the --trusted flag),
* And a header file (*<host|enclave>\_args.h*) that defines the parameters that are passed to all functions defined in the EDL file.

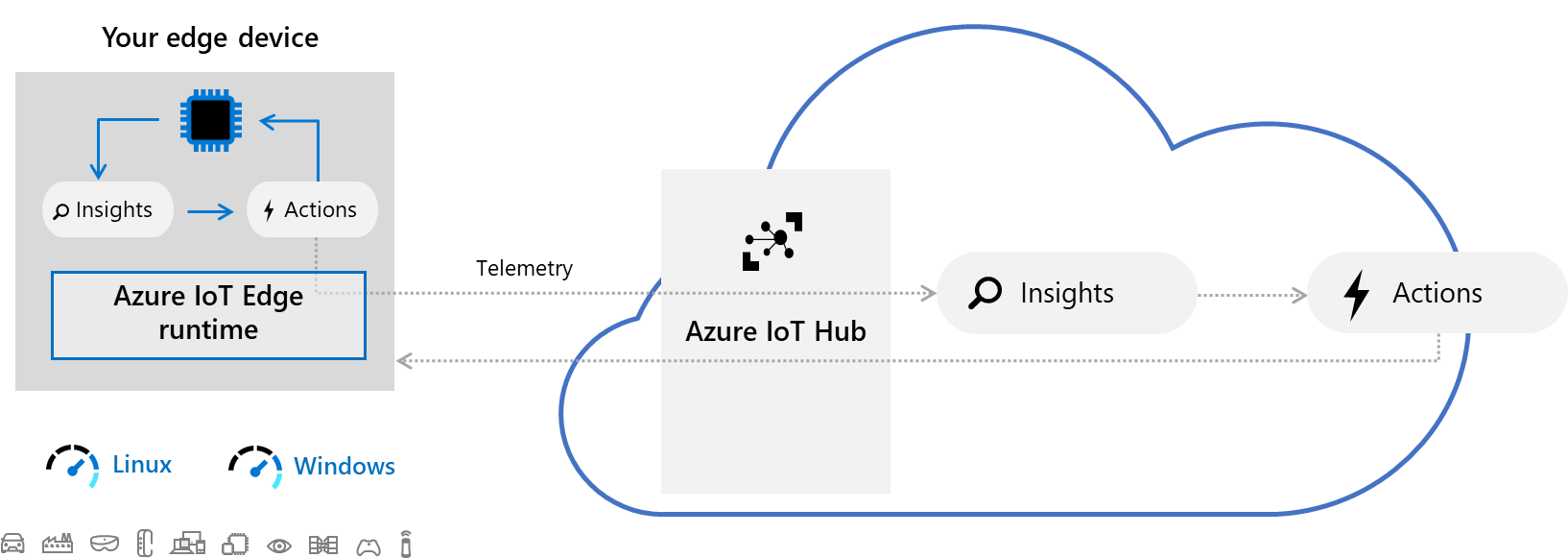
**Note** For more information on using the oeedger8r tool, see article [Getting started with the Open Enclave edger8r](https://github.com/openenclave/openenclave/tree/master/docs/GettingStartedDocs/Edger8rGettingStarted.md)[[59]](#footnote-60).

The above generated files contain code to aid in the marshalling of function calls and data across the host application/enclave boundary such that the ECALLs and OCALLs appears as normal function calls to you as a developer. The underlying platform and TEE specifics behaviors are abstracted away.



### Azure IoT Platform for the “Intelligent Cloud, Intelligent Cloud”

Generally speaking, a cloud gateway represents a key component for the so-called “Intelligent Cloud, Intelligent Edge”. It indeed provides a cloud hub for (edge) devices to connect securely to the cloud and send data. It also provides device management, capabilities, including command and control of devices.



In the Azure IoT platform, [Azure IoT Hub](https://azure.microsoft.com/en-us/services/iot-hub/)[[60]](#footnote-61) is a hosted cloud service that ingests events from (edge) devices, acting as a message broker between these devices and backend services. IoT Hub provides secure connectivity, event ingestion, bidirectional communication, and device management.

Mobile and (Industrial) Internet of Things ((I)IoT) devices can thus securely register with the cloud, here the Azure cloud, and can connect to the cloud to send and receive data. Some devices may be edge devices that perform some data processing on the device itself or in a field gateway.

The Intelligent Edge indeed brings the power of the cloud to edge devices and demands security for trust. “Cloud-enabled computing at the edge means concentrating data, and therefore inherent value even if only momentarily. It also means moving tremendous value from the cloud to the edge in the form of intellectual property, algorithms, curated parameters, and value operations like policy enforcements, metering, and monetization. The Intelligent Edge is without a doubt a high-value bullseye to nefarious hacking and demands a high bar for security.”[[61]](#footnote-62)

For edge processing, we then recommend Azure IoT Edge. Azure IoT Edge is an implementation of a secure Intelligent Edge platform that is operating system, processor architecture, and hardware agnostic.

As such, Azure IoT Edge is a fully managed service built on Azure IoT Hub that allows you to remotely manage code on your devices so that you can send more of your (cloud) workloads to the edge devices - Artificial Intelligence (AI), Azure and third-party services, or your own business logic - to run on these devices via standard containers. By moving certain workloads to the edge of the network, your devices spend less time communicating with the cloud, react more quickly to local changes, and operate reliably even in extended offline periods.

**Note** For more information, see the [Microsoft Azure IoT Architecture Reference](http://download.microsoft.com/download/A/4/D/A4DAD253-BC21-41D3-B9D9-87D2AE6F0719/Microsoft_Azure_IoT_Reference_Architecture.pdf)[[62]](#footnote-63) guide. This guide aims to accelerate customers building IoT solutions on the Azure IoT platform, and more generally speaking on Azure, by providing a proven production ready architecture, with proven technology implementation choices, and with links to Solution Accelerator reference architecture implementations such as [Remote Monitoring](https://github.com/Azure/azure-iot-pcs-remote-monitoring-dotnet/)[[63]](#footnote-64) and [Connected Factory](https://github.com/Azure/azure-iot-connected-factory)[[64]](#footnote-65) on GitHub.

This document offers an overview of the IoT space, recommended subsystem factoring for scalable IoT solutions, prescriptive technology recommendations per subsystems, and detailed sections per subsystem that explore use cases and technology alternatives.

## Step-by-step directions

This module covers the following three activities:

1. Building a TEE-based Linux application on Intel SGX.
2. Building a TEE-based Linux application on a simulated ARM TrustZone environment.
3. Building a TEE-based Linux module on an Edge ARM TrustZone device.

Each activity is described in order in the next sections.

### Building a TEE-based Linux application on Intel SGX

This section covers the following activities:

1. Installing and configuring Visual Studio on your Windows 10 development machine.
2. Creating a C/C++ TEE-based Linux application.
3. Modifying the TEE-based Linux application.

Each activity is described in order in the next sections.

**Note** For more information, see articles [An introduction to creating a sample enclave using Intel Software Guard Extensions](https://software.intel.com/en-us/articles/intel-software-guard-extensions-developing-a-sample-enclave-application)[[65]](#footnote-66) and [Using Visual Studio to Develop Enclave Applications for Linux](https://github.com/openenclave/openenclave/blob/feature.new_platforms/docs/GettingStartedDocs/VisualStudioLinux.md)[[66]](#footnote-67).

You will first need to setup the development environment on your local machine. Let’s see how to proceed.

#### Installing and configuring Visual Studio on your Windows 10 development machine

This section describes how to configure on your local Windows 10 development machine a Visual Studio IDE, as it supports an Open Enclave extension as well as a remote compiler: the [Open Enclave Wizard – Preview extension](https://marketplace.visualstudio.com/items?itemName=MS-TCPS.OpenEnclaveSDK-VSIX)[[67]](#footnote-68).

This extension includes preview support for TEE platforms, including Intel SGX and ARM TrustZone with a Windows or Linux host application. In addition, this preview includes support for testing your enclave under simulation when developing for Intel SGX or ARM TrustZone.

As the title of this activity indicates, you will walk through the development for Intel SGX.

Perform the following steps:

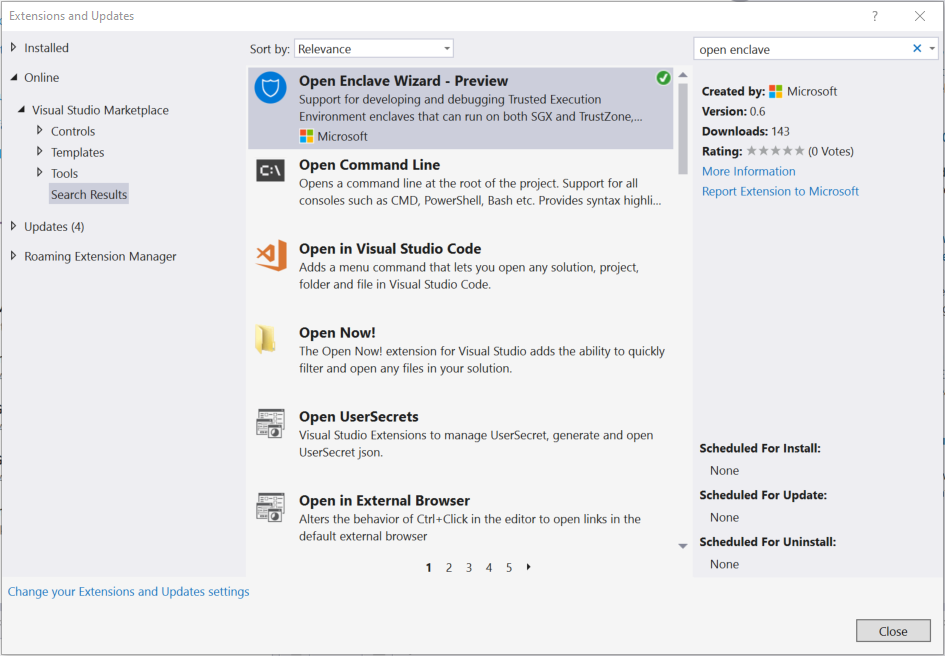
1. Install Visual Studio 2017 or Visual Studio 2019 ([Community Edition](https://visualstudio.microsoft.com/vs/community)[[68]](#footnote-69), or any other edition). (Visual Studio 2017 is featured in the steps below. Any difference with Visual Studio 2019 if any will be highlighted.)
2. Launch Visual Studio.
3. On the menu bar of Visual Studio, select **Tools** and then **Get Tools and Features**. This will launch **Visual Studio Installer**. (If a user control dialog pops up, click on **Yes**.)

**Note** In Visual Studio 2019, simply select **Tools**.

1. In **Visual Studio Installer**, select **Workloads > Other Toolsets > Linux Development with C++**. Click on **Modify**.
2. Back in Visual Studio, on the menu bar, select **Tools > Extensions and Updates > Online**.

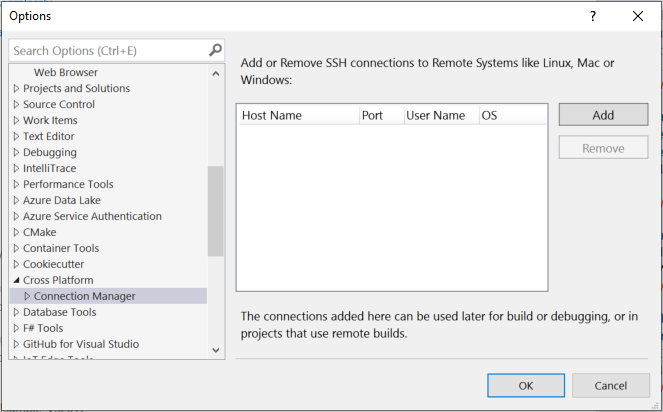
**Note** In Visual Studio 2019, select **Extensions -> Manage Extensions -> Online**.

1. Search for “*Open Enclave Wizard – Preview*“, install the extension.

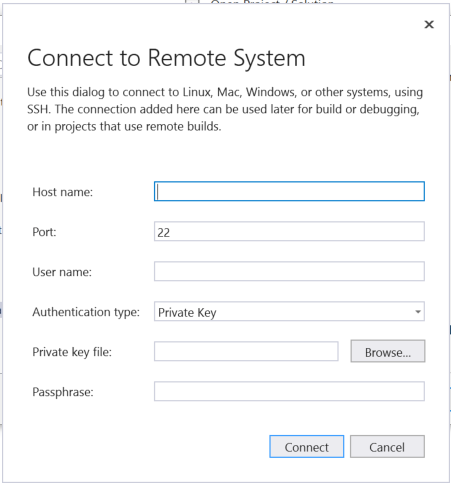


**You MUST restart Visual Studio after installing the extension** **to complete the installation.**

1. Finally, configure Visual Studio with the address (or name) of your DC-series VM (or any other Linux build machine), via **Tools > Options > Cross Platform > Connection Manager**.



Click on **Add**. A Connect to Remote System dialog opens up.



1. Fill in the fields required to connect to your DC-series VM and click on **Connect**.

**Important note** Your PuTTY private key .ppk file that you generated for your key pair must be first converted to the OpenSSH format. You can use the **Conversions** > **Export OpenSSH key** from the application menu.

This step may take a minute or two, as Visual Studio will copy some files locally for use by IntelliSense.

****

**Note** For more information, see article [Connect to your target Linux system in Visual Studio](https://docs.microsoft.com/en-us/cpp/linux/connect-to-your-remote-linux-computer?view=vs-2017)[[69]](#footnote-70).

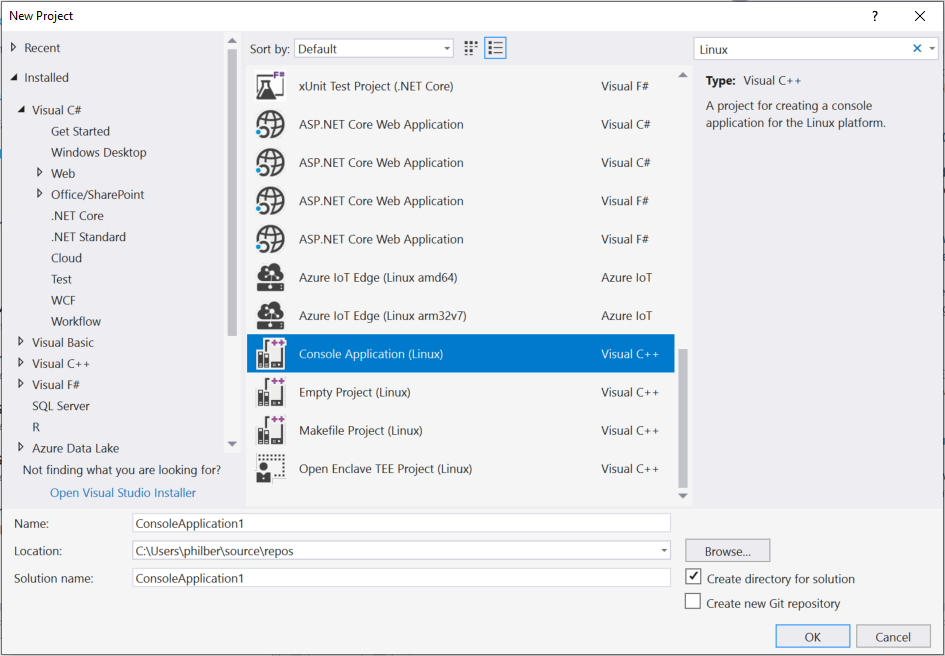
**Congrats! Your Windows 10 development machine is now fully configured.**

You will now walk through the process of creating a C/C++ TEE-based application that uses an enclave.

#### Creating a C/C++ TEE-based Linux application

Perform the following steps:

1. Create a new Linux application by choosing **File > New > Project** on the menu bar of Visual Studio. The **New Project** dialog box opens up.
2. Search the Linux console app template called "Console Application (Linux)" by typing “*Linux*”.



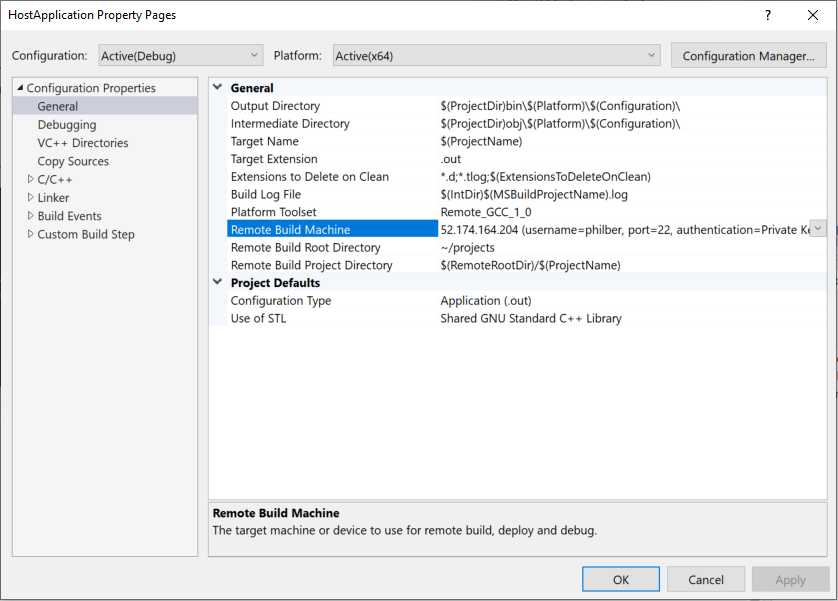
**Important note** This is NOT the "Console App (.NET Core)".

**Note** In Visual Studio 2019, this template is called “Console App”.

(If it is not immediately visible, the template can be found under **Installed > Visual C++ > Cross Platform > Linux**.)

* 1. Enter a name, for example “HostApp” in our illustration, a location, and solution name in the appropriate fields like any other Visual Studio project.
  2. Click on **OK**. This will create a "Hello World" console application.

1. Configure the application project to use your Linux build environment, by right clicking on the project in the **Solution Explorer** and selecting **Properties**. The application property pages opens up.



1. Under **Configuration Properties > General -> Remote Build Machine**, explicitly set the build machine to the build machine you configured in the **Connection Manager**.

**Note** Due to a current Visual Studio bug, this step is required even if the correct value is shown by default. In other words, make sure the connection is shown in **bold**.

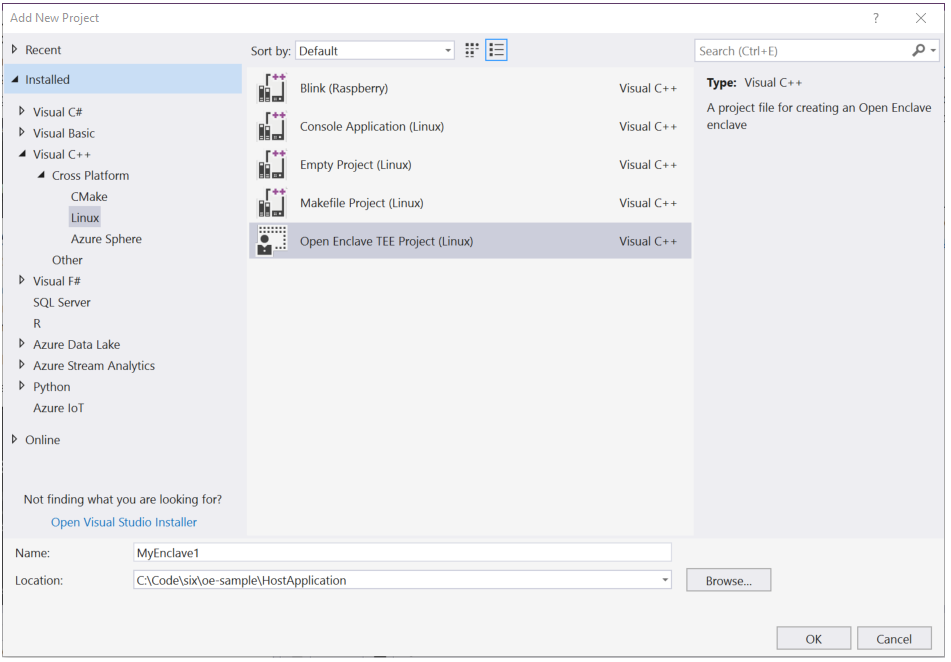
**Note** If you’re using Visual Studio 2019 instead, you also need to update **Configuration Properties > Debugging > Remote Debug Machine** to your build machine, again due to a current Visual Studio 2019 bug.

At this point, you should be able to build and debug your newly created Hello World application.

**Note** For further discussion, see articles [Deploy, run, and debug your Linux project](https://docs.microsoft.com/en-us/cpp/linux/deploy-run-and-debug-your-linux-project?view=vs-2017)[[70]](#footnote-71) and [Linux debugging walkthrough](https://docs.microsoft.com/en-us/cpp/linux/deploy-run-and-debug-your-linux-project?view=vs-2017)[[71]](#footnote-72).

You now have your first application project, but this is only the host application. You now need to create the enclave component.

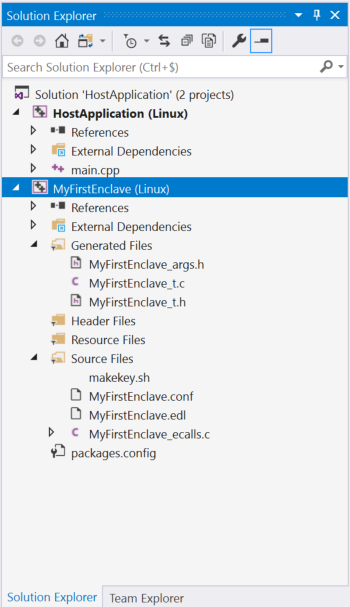
1. Right-click on your newly solution in the **Solution Explorer**, click on **Add > New project** **> Open Enclave TEE Project (Linux)** to add an enclave library project.



(If it is not immediately visible, look under **Installed > Visual C++ > Cross Platform > Linux**.)

1. Give it a name, for example “*MyFirstEnclave*” in our illustration and click on **OK**.

The Wizard creates an enclave project with several files.



The sample enclave has an ecall\_DoWorkInEnclave() method exposed to host applications, that will simply call an ocall\_DoWorkInHost() method that will be implemented in the host application, as reflected in the EDL file (*<*YourEnclaveProjectName>*.edl*).

enclave {

trusted {

/\* define ECALLs here. \*/

public int ecall\_DoWorkInEnclave();

};

untrusted {

/\* define any OCALLs here. \*/

void ocall\_DoWorkInHost();

};

};

In accordance, the wizard will create a sample enclave with an ecall\_DoWorkInEnclave() method exposed to host applications, that will simply call an ocall\_DoWorkInHost() method that will be implemented in the host application. In this walkthrough, you'll leave this project as is for now, but afterwards you can modify it as you like.

1. Configure the enclave project to use your Linux build environment, as you did in the above step 3. At this point, the enclave would build, but cannot be run as the host application doesn't invoke it yet. At this stage, you have indeed two projects in place in solution, but they aren’t linked together for now…
2. You now need to import the enclave into your host application project. To do so, right-click on your host application project in the S**olution Explorer** and select **Open Enclave Configuration > Import enclave**, then navigate to and select the EDL file (*<*YourEnclaveProjectName>*.edl*) in your enclave project. This file is containing all the enclave details.

Visual Studio will then link your projects by adding references between them and importing relevant libraries. As such, this step will modify your host application project settings and add some additional files to it, including a C file named <YourEnclaveProjectName>\_*host.c*. This C file contains a sample\_enclave\_call() method that will load and call ecall\_DoWorkInEnclave(), and also contains a sample implementation of a ocall\_DoWorkInHost() method that just prints a message when called.

Although the app could be compiled and run at this point, sample\_enclave\_call() is still not called from anywhere.

1. Open the host application's file *main.cpp* and add a call to sample\_enclave\_call(). For example, update the file *main.cpp* to look like this, where the extern C declaration is needed because *main.cpp* is a C++ file whereas the <YourEnclaveProjectName>\_*host.c* file is a C file:

include <cstdio>

extern "C" {

void sample\_enclave\_call(void);

};

int main()

{

printf("hello from LinuxApp!\n");

sample\_enclave\_call();

return 0;

}

1. You can now set breakpoints in Visual Studio, e.g., inside ecall\_DoWorkInEnclave() and inside ocall\_DoWorkInHost() and run and debug the enclave application just like any other application.

The resulting Visual Studio solution will have three configurations: **Debug**, **SGX-Simulation-Debug**, and **Release**.

The **SGX-Simulation-Debug** will work the same as **Debug**, except that SGX support will be emulated rather than using hardware support. This allows debugging on hardware that does not support SGX. The **Debug** and **Release** configurations can only be run (whether natively or in a VM) successfully on SGX-capable hardware like your DC-series VM.

For the platform, use **x64** since the Open Enclave SDK currently only supports 64-bit enclaves.

#### Modifying the TEE-based Linux application

Once you have the basic application working, you can modify it as desired.

For example, to define new APIs between the enclave and the host application, perform the following steps:

1. Edit the file <YourProjectName>*.edl*.
2. Define any trusted APIs (called "ECALLs" as covered in section § **Important concepts**) you want to call from your application in the trusted{} section, and in the untrusted{} section.
3. Likewise, define any application APIs (called "OCALLs" as covered in section § **Important concepts**) that you want to call from your enclave.
4. Edit the *<*YourProjectName>*\_ecalls.c* file and fill in implementations of the ECALL(s) you added.
5. Edit your application sources and fill in implementations of the OCALL(s) you added.
6. Run and debug the enclave application just like any other application.

### Building a TEE-based Linux application on a simulated ARM TrustZone environment

This section covers the following four activities:

1. Installing and configuring Visual Studio Code on your Windows 10 development machine.
2. Creating a standalone C/C++ application.
3. Building the standalone C/C++ application.
4. Debugging the standalone C/C++ application.

Each activity is described in order in the next sections.

**Note** For more information, see article [Open Enclave extension for Visual Studio Code](https://github.com/openenclave/openenclave/blob/feature.new_platforms/new_platforms/vscode-extension/README.md)[[72]](#footnote-73).

You will first need to setup the development environment on your local machine. Let’s see how to proceed.

#### Installing and configuring Visual Studio Code on your Windows 10 development machine

This section describes how to configure on your local Windows 10 machine a Visual Studio Code IDE.

In addition, you will need to install some Visual Studio Code extensions:

1. Locally, the Visual Studio Code [Remote Development extension pack](https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensionpack)[[73]](#footnote-74) that allows you to open any folder in a container on a remote machine like your DC-series VM in Azure, A Linux machine elsewhere, or in the local Windows Subsystem for Linux (WSL) of your Windows 10 local machine (see section § Error! Reference source not found. in the Appendix A. Prerequisites and additional configuration). This pack includes three extensions:
   1. [Visual Studio Code Remote - SSH](https://aka.ms/vscode-remote/download/ssh)[[74]](#footnote-75). It allows you to work with source code in any location by opening folders on a remote machine/VM using SSH and supports connecting to x86\_64 Linux SSH servers like your above DC-series VM.
   2. [Visual Studio Code Remote - WSL](https://aka.ms/vscode-remote/download/wsl)[[75]](#footnote-76). It allows you to get a Linux-powered development experience from the comfort of Windows 10 by opening any folder in WSL.
   3. [Visual Studio Code Remote - Containers](https://aka.ms/vscode-remote/download/containers" \t "_blank)[[76]](#footnote-77). It allows you to work with a sandboxed toolchain or container-based application by opening any folder inside (or mounted into) a container.
2. Remotely, the [Open Enclave extension for Visual Studio Code](https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave)[[77]](#footnote-78), supporting the Open Enclave SDK, including development, debugging, emulators, and deployment.

You will create a Standalone project on your Linux machine, once connected to it in SSH. For that purpose, in the suggested configuration, you will need to install on your DC-series VM all the [requirements](https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave#Requirements)[[78]](#footnote-79) listed.

Perform the following steps:

1. Install Visual Studio Code or [Visual Studio Code – Insiders](https://code.visualstudio.com/insiders)[[79]](#footnote-80).
2. Install [Git for Windows](https://git-for-windows.github.io/)[[80]](#footnote-81) (2.10 or later) and make sure that long paths are enabled:

git config --global core.longpaths true

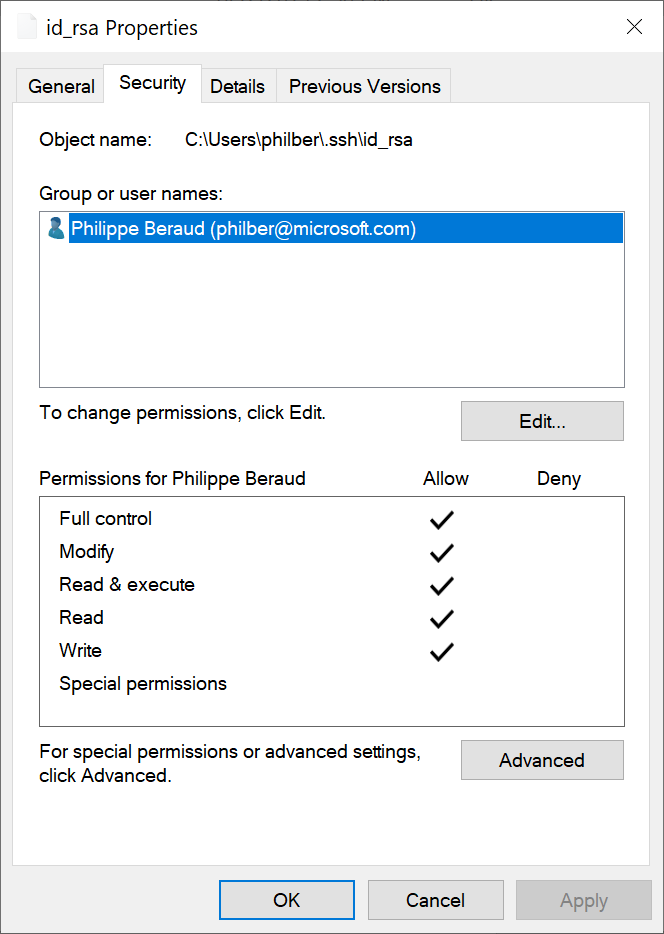
1. Install an [OpenSSH compatible SSH client](https://code.visualstudio.com/docs/remote/troubleshooting#_installing-a-supported-ssh-client)[[81]](#footnote-82) if one is not already present on your local Windows 10 machine.
2. Install the Remote Development extension pack.

**Note** For details on setting up and working with each specific extension of the Remote Development extension pack, see articles [Remote Development using SSH](https://code.visualstudio.com/docs/remote/ssh)[[82]](#footnote-83), [Developing in WSL](https://code.visualstudio.com/docs/remote/wsl)[[83]](#footnote-84), and [Developing inside a Container](https://code.visualstudio.com/docs/remote/containers)[[84]](#footnote-85). For troubleshooting tips and tricks for each of these extensions, see article [Remote Development Tips and Tricks](https://code.visualstudio.com/docs/remote/troubleshooting#_installing-a-supported-ssh-client)[[85]](#footnote-86).

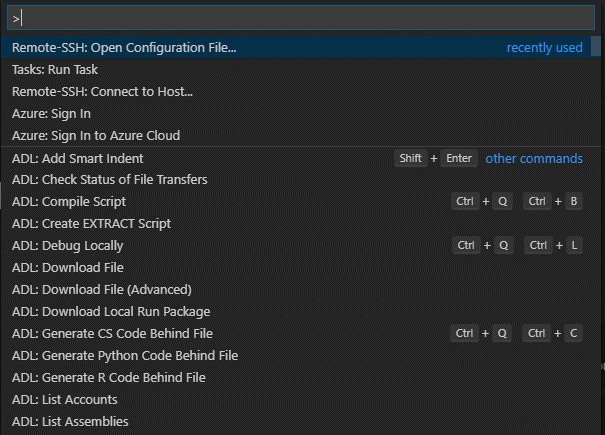
1. **Visual Studio Code uses** [**SSH configuration files**](https://linux.die.net/man/5/ssh_config)**[[86]](#footnote-87) and requires SSH key based authentication to connect to your host.**

Since you used PuTTYGen to set up SSH public key authentication for the DC-series VM you are connecting to (see section § **Deploying a DC-series VM on Azure**), you need first to convert your private key to the OpenSSH format:

* 1. Open PuTTYGen locally and load your PuTTY private key .ppk file that you generated for your key pair.
  2. Click on **Conversions** > **Export OpenSSH key** from the application menu. Save the converted key file, for example, *id\_rsa*, to a local location such as *%USERPROFILE%\.ssh*.
  3. Validate that the local permissions on the exported key file only grant **Full Control** to your user.



1. Back in Visual Studio Code, press F1 or CTRL-Shift-P to open the Command Palette, and run **Remote-SSH: Open Configuration File...**



1. Select the SSH config file you wish to change, for example *%USERPROFILE%\.ssh\config*, and add (or modify) a host entry in the config file as follows:

# Read more about SSH config files: https://linux.die.net/man/5/ssh\_config

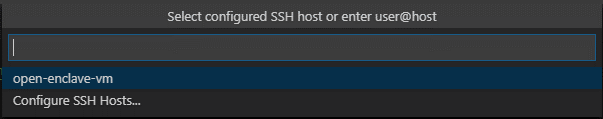
Host <*name-of-your-dc-series-vm-here*>

User <*your-user-name-on-your-dcseries-vm*>

HostName <*dc-series-vm-ip-goes-here*>

IdentityFile c:\users\<*your-user-name-on-Windows10*>\.ssh\id\_rsa

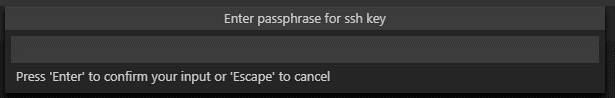
1. Press CTRL+S to save the config file.
2. Open a remote SSH session to your DC-series VM:
3. Press F1 or CTRL-Shift-P to specify a command, run Remote-SSH: Connect to SSH Host..., and select your DC-series VM name in the list.



A SSH session to your DC-series VM is opening. Visual Studio Code will now continue to configure it.



1. When prompted, type your passphrase for your SSH key and press ENTER.



Once finished, you now see a SSH indicator in the bottom left corner, and you'll be able to use Visual Studio Code as you would normally!



ET voila! Any Visual Studio Code operations you perform in this window will be executed in the SSH environment, everything from editing and file operations, to debugging, using terminals, and more.

1. Ensure that the [requirements](https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave#Requirements)[[87]](#footnote-88) are met for the Open Enclave extension for Visual Studio Code:
   1. In Visual Studio Code, make sure that the [Visual Studio Code Native Debug extension](https://marketplace.visualstudio.com/items?itemName=webfreak.debug)[[88]](#footnote-89) is installed. Press CRTL-p and run ext install webfreak.debug in Visual Studio Code and install GDB/LLDB.
   2. Install [CMake 3.12 or above](https://cmake.org/download/)[[89]](#footnote-90), currently 3.15:
      1. From Visual Studio Code, click on **Terminal** > **New Terminal**.
      2. From the terminal console, download the archive file *cmake-3.15.2.tar.gz* and compile it:

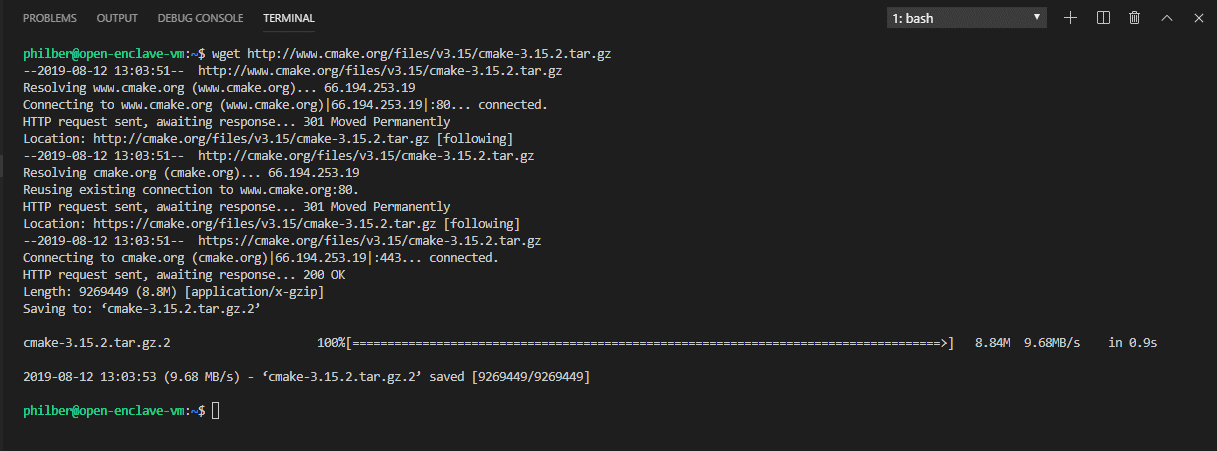
wget http://www.cmake.org/files/v3.15/cmake-3.15.2.tar.gz

tar -xvzf cmake-3.15.2.tar.gz

cd cmake-3.15.2/

./configure

make



* + 1. Make's install command installs cmake by default in the folder */usr/local/bin/cmake*. Run the following command to install (copy) the binary and libraries to the new destination:

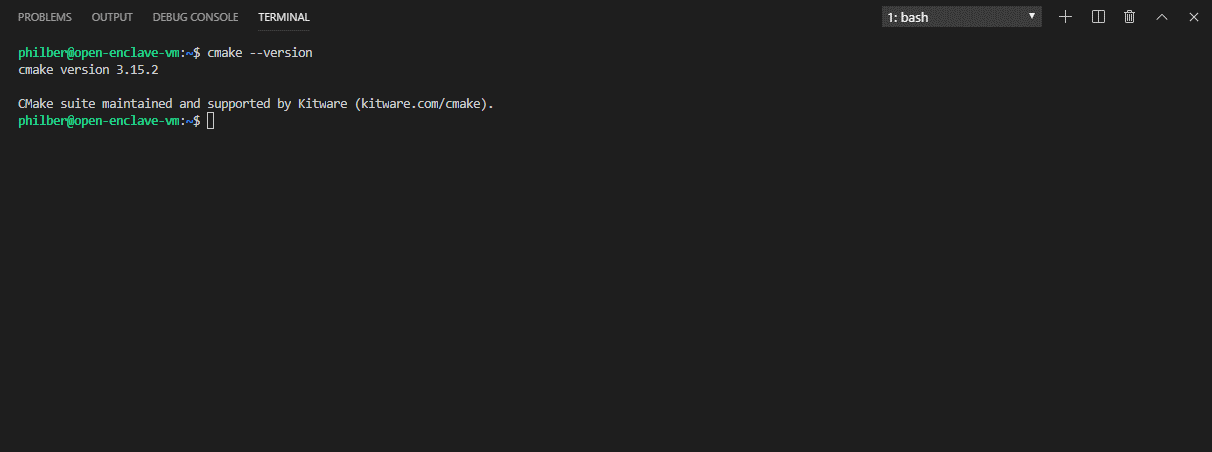
sudo make install

* + 1. If you haven't already installed a newer cmake installation, run the following command to tell your distro, e.g. Ubuntu in our illustration, that the cmake command is now being replaced by an alternative installation:

sudo update-alternatives --install /usr/bin/cmake cmake /usr/local/bin/cmake 1 --force

Verify the cmake version using the following command:

cmake --version



* 1. Install the required build components from the bash shell:

sudo apt update && sudo apt install -y build-essential cmake gcc-arm-linux-gnueabihf gcc-aarch64-linux-gnu g++-arm-linux-gnueabihf g++-aarch64-linux-gnu gdb-multiarch python

When prompted, type your password.

Furthermore, you may get the following warning and error:

W: GPG error: https://packages.microsoft.com/repos/azure-cli bionic InRelease: The following signatures couldn’t be verified because the public ckey is not available: NO\_PUBKEY EB3E94ADBE1229CF

E: the repository ‘https://packages.microsoft.com/repos/azure-cli bionic InRelease’ is not signed.

N: Updating from such a repository can’t be done securely, and its therefore disabled by default.

N: See apt-secure(8) manpage for repository creation and user configuration details.

It happens when you don't have a suitable public key for this repository.

To solve this problem, perform the following steps:

1. Use this command, which retrieves the key from ubuntu key server:

gpg --keyserver hkp://keyserver.ubuntu.com:80 --recv EB3E94ADBE1229CF

1. And then this one, which adds the key to apt trusted keys:

gpg --export --armor EB3E94ADBE1229CF | sudo apt-key add -

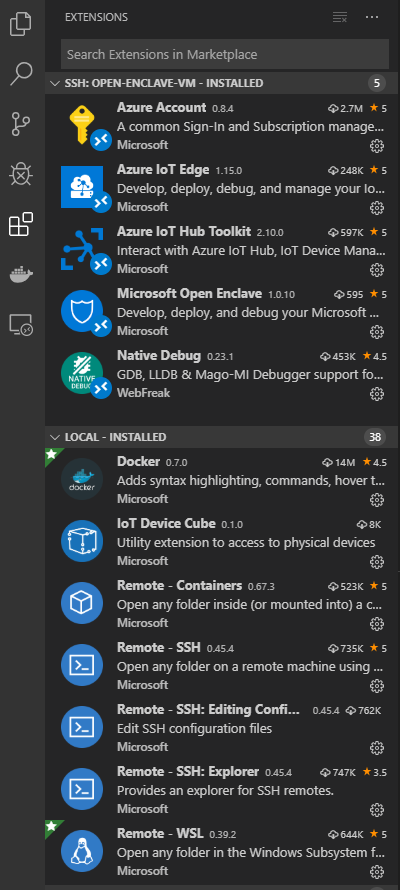
1. Rerun the above command to install the required build components.
   1. Ensure that all QEMU dependencies are installed. [QUEMU](https://www.qemu.org/)[[90]](#footnote-91) is a system emulator that can run ARM TrustZone Trusted Applications (TAs) on a x64 machine as though they were running on a TrustZone-capable hardware.

In the terminal console, run:

sudo apt update && sudo apt install -y libpixman-1-0 zlib1g libc6 libfdt1 libglib2.0-0 libpcre3 libstdc++6

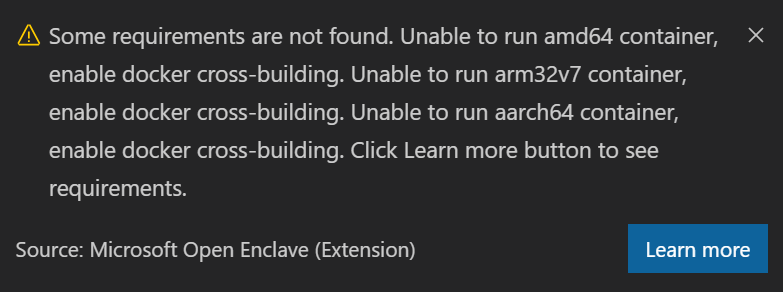
1. Back in Visual Studio code, on the menu bar, install the Open Enclave extension for Visual Studio Code. When prompted, click on **Install**.

Visual Studio Code runs extensions in one of two places: i) locally on the UI / client side, or ii) remotely in the SSH session. While extensions that affect the Visual Studio Code UI, like themes and snippets, are installed locally, most extensions needed here will reside inside your DC-series VM. This will be the case for the Open Enclave extension for Visual Studio Code.



1. Use the Microsoft Open Enclave: Check System Requirements command - commands can be found using F1 or CTRL-Shift-P to validate your system.

The command will query whether the required tools and the required versions are present on your system. Any unmet requirements will be presented in a Visual Studio Code warning window.



Otherwise, you should see instead the message “Your system meets the requirements”.

#### Creating a standalone C/C++ application

Perform the following steps:

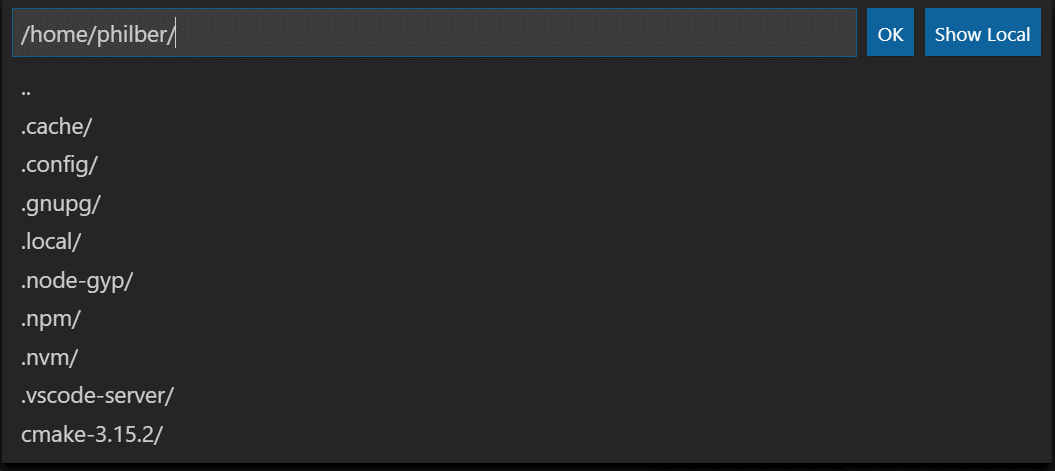
1. Launch Visual Studio Code.
2. As already stated, and illustrated above, the Visual Studio Code Remote - SSH extension lets you connect to your DC-series VM as your full-time development environment right from Visual Studio Code. You can develop in a Linux-based environment, use Linux specific toolchains and utilities, and run and debug your Linux-based applications all from the comfort of Windows.

Open a remote SSH session to your DC-series VM:

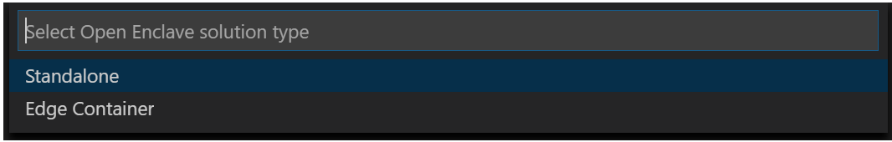
* 1. Press F1 or CTRL-Shift-P to specify a command, run Remote-SSH: Connect to SSH Host..., and select your DC-series VM name in the list.
  2. When prompted, type your passphrase for your SSH key and press ENTER.

**Note** For more information, see article [Remote Development using SSH](https://code.visualstudio.com/docs/remote/ssh)[[91]](#footnote-92).

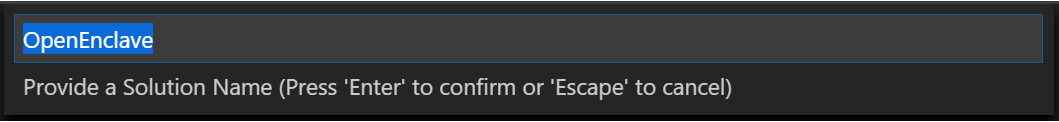
1. Use the Microsoft Open Enclave: New Open Enclave Solution command - commands can be found using F1 or CTRL-Shift-P - to create your first new standalone application.



1. Specify the folder in which you want to create the application.

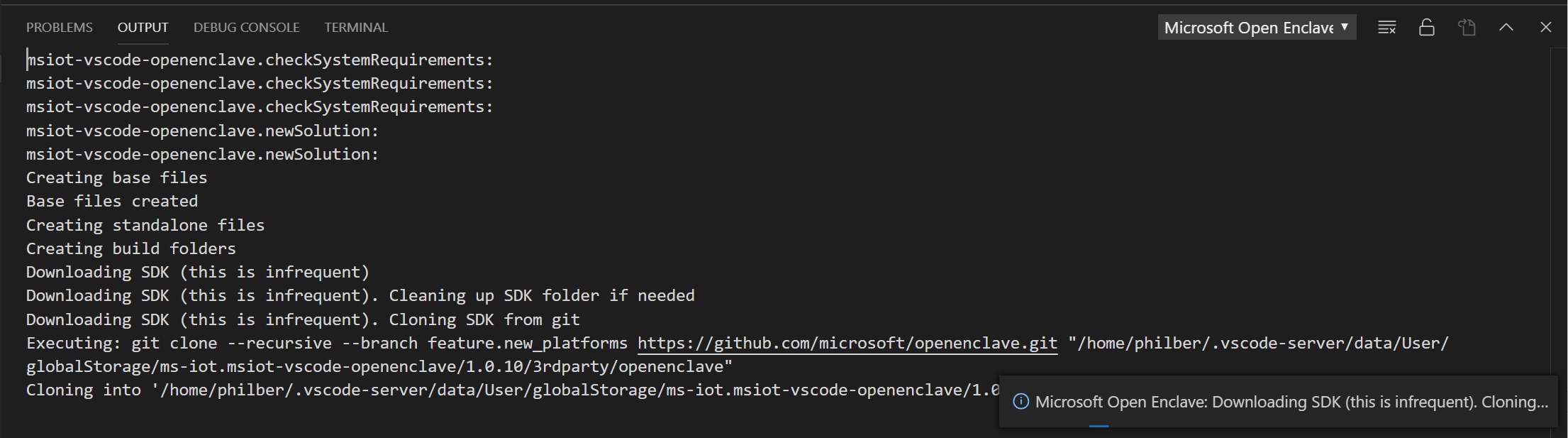


1. When invited to choose the option, create a **Standalone** project.

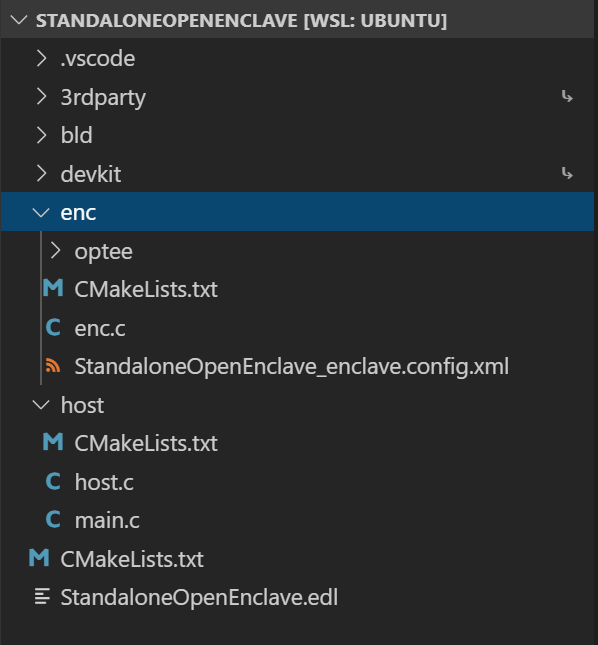


1. Provide a name for the host application/enclave, for example “*Standalone*”, and press ENTER to confirm.

A new solution will be created in the folder you've selected.



That solution will contain both the host and enclave as well as the required EDL file. The solution appears to somehow like the one in the previous activity.



The EDL file is as follows:

// Copyright (c) Microsoft Corporation. All rights reserved.

// Licensed under the MIT License.

enclave {

from "openenclave/stdio.edl" import \*;

trusted {

/\* define ECALLs here. \*/

public int ecall\_handle\_message([in, string] char \*input\_msg, [out, count=enclave\_msg\_size] char \*enclave\_ms, unsigned int enclave\_msg\_size);

};

untrusted {

/\* define OCALLs here. \*/

int ocall\_log([in, string] char \*msg);

};

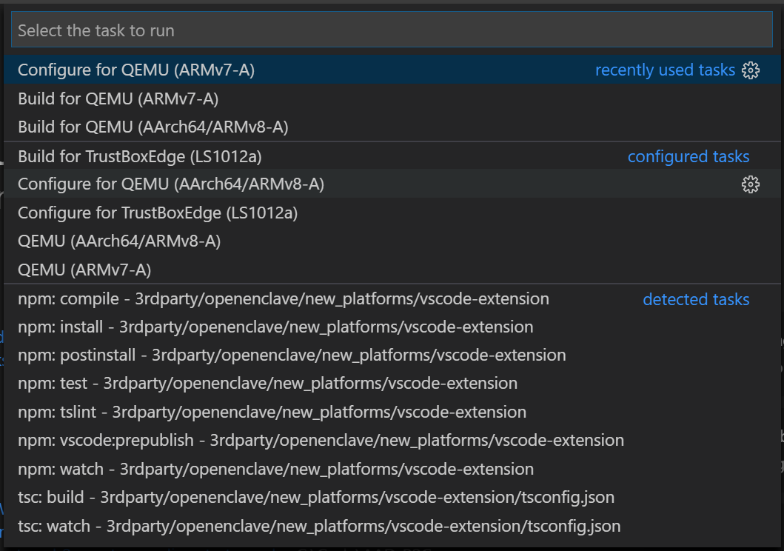
};

#### Building the standalone C/C++ application

As already noticed, and considering previous requirements, the underlying system used in Visual Studio Code to build is CMake.

Perform the following steps

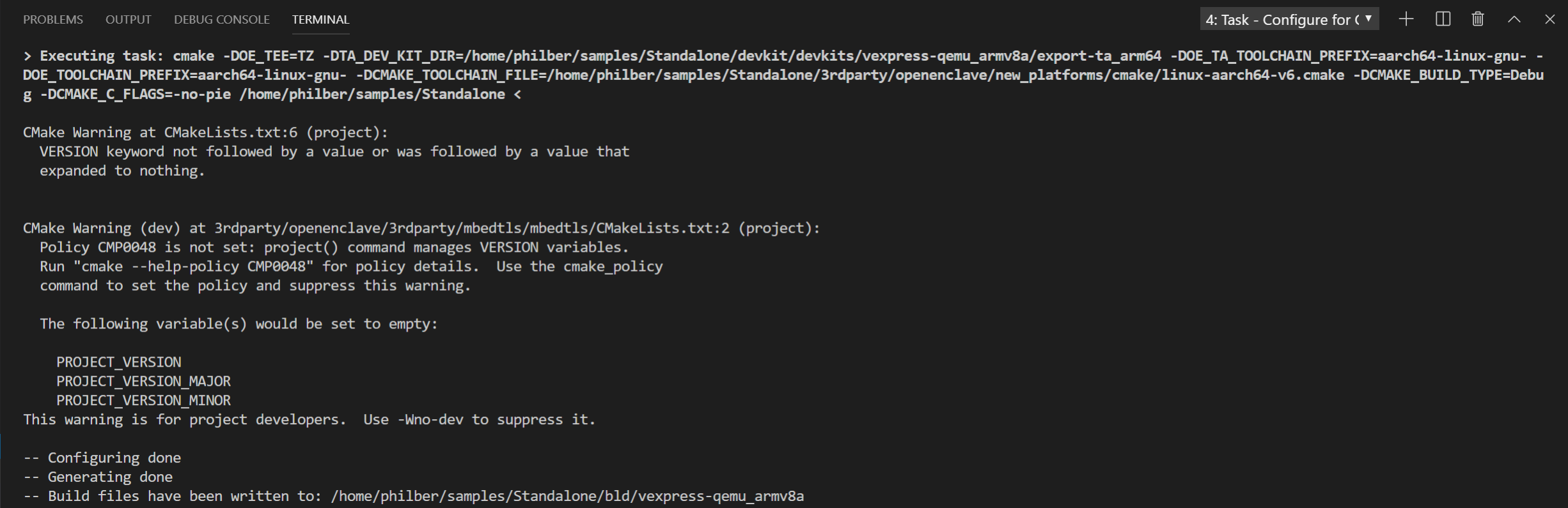
1. Once still connect to your DC-series VM via a SSH session, press F1 or CTRL-Shift-P to specify a command.
2. Select Tasks: Run Task. You should see tasks configured and build for each target: **ARMv7-A** (32-bit), **AArch64/ARMv8-A** (64-bit), and **TrustBoxEdge (LS1012a)**.



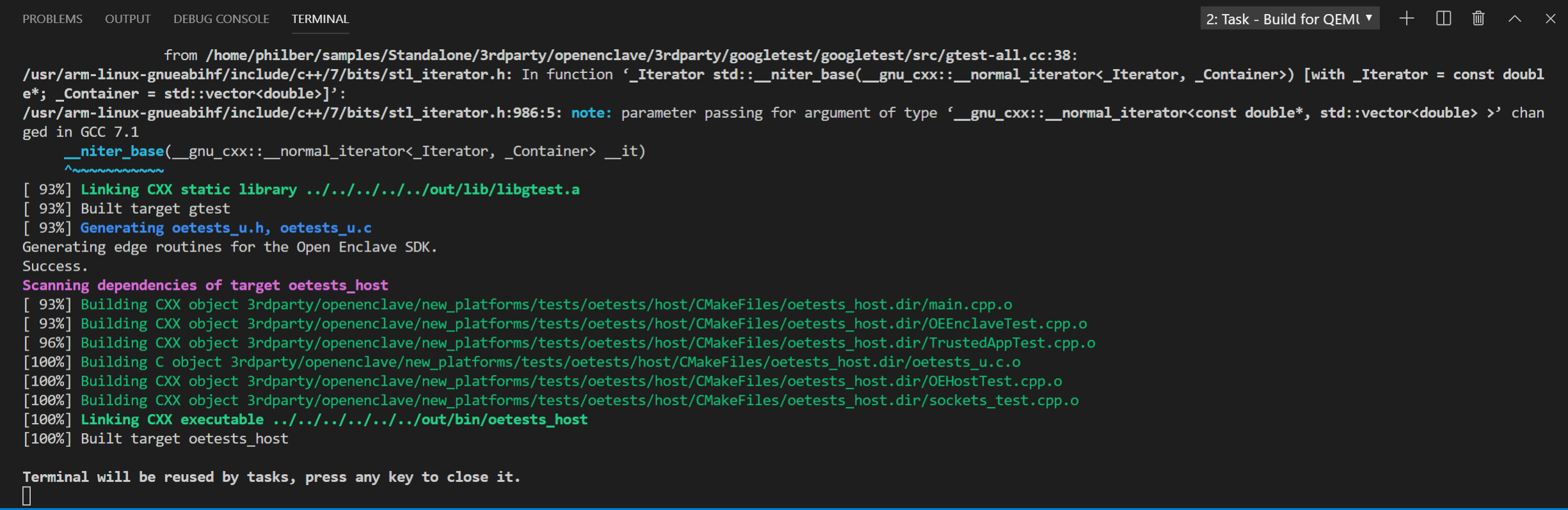
1. Select Configure for AArch64/ARMv8-A.

**Note** For more information, see document [TrustZone technology for ARM v8-M Architecture](https://static.docs.arm.com/100690/0100/armv8_m_architecture_trustzone_technology_100690_0100_00_en.pdf)[[92]](#footnote-93).

The configure task will invoke cmake to create the required build files. This is only required to be run once.



1. Select Build for AArch64/ARMv8-A. The build task will do the actual compiling and linking.



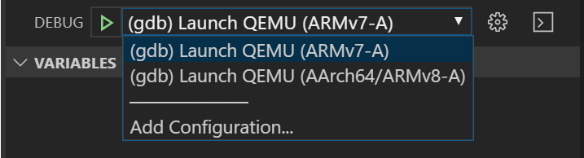
#### Debugging the standalone C/C++ application

Debugging your standalone project's enclave is easy as you will see now.

**Important note** Ensure that all of the QEMU dependencies are installed in your development environment. See step 4.d in above section § **Installing and configuring Visual Studio Code on your Windows 10 development machine**.

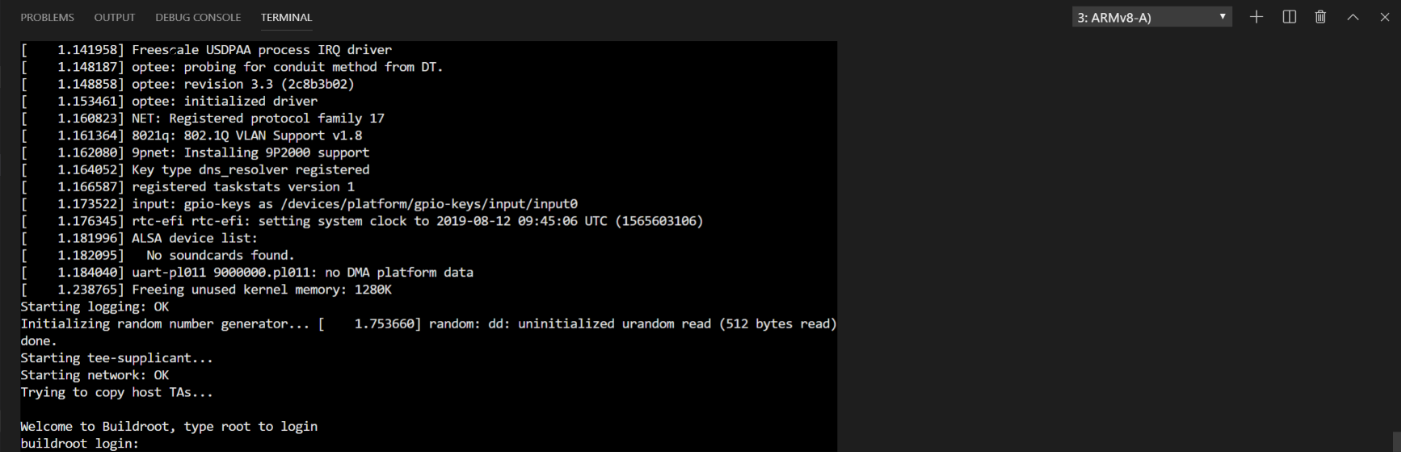
Perform the following steps:

1. Set breakpoints in the files you wish to debug. Breakpoints in the enclave may only be added before the emulator (QEMU) starts or when the debugger is already broken inside the enclave.
2. Choose the architecture you are interested in debugging:
   1. Navigate to the Visual Studio Debug view – Use CTRL-Shift-D -.



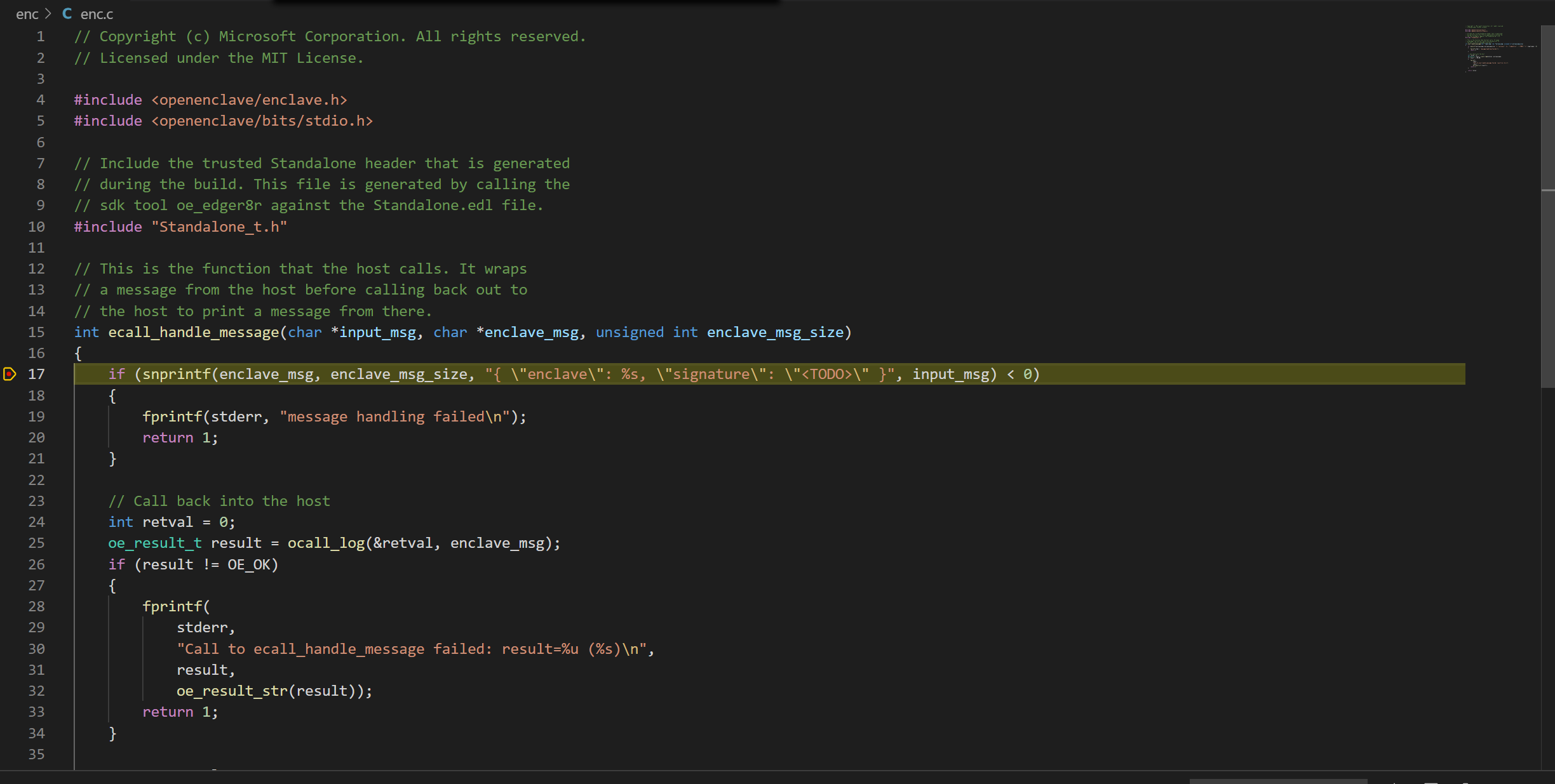
* 1. Select (gdb) Launch QEMU (AArch64/ARMv8-A) from the debug configuration dropdown.

1. You can simply hit F5. This will run cmake configuration, run the build, start QEMU, and load the host and enclave symbols into an instance of the debugger.
2. Open the terminal view.



1. Log into QEMU using root (no password is required).
2. Start the host process by entering “/mnt/host/bin/<*YourOpenEnclaveSolutionName*>”, for example in our illustration:

/mnt/host/bin/Standalone



Once you have the standalone application working, you can modify it as desired.

### Building a TEE-based Linux module on an Edge ARM TrustZone device

As already introduced, the Intelligent Edge brings the power of the cloud to mobile and (Industrial) Internet of Things ((I)IoT) devices and demands security for trust.

So, let’s now considered more specifically TEE-based application. As already stressed, we recommend Azure IoT Edge for edge processing (see section § **Azure IoT Platform for the “Intelligent Cloud, Intelligent Cloud”** earlier in this document), and consequently the Azure IoT Edge will be featured in this section.

With an IoT Edge device as a target, the TEE-based code will take the form of an Azure IoT Edge module. An Azure IoT Edge module, or sometimes just module for short, is a container that contains executable code. You can deploy one or more modules to an Azure IoT Edge device. Modules perform specific tasks like ingesting data from sensors, performing data analytics or data cleaning operations, or sending messages to an IoT hub.

**Note** For more information, see article [Understand Azure IoT Edge modules](https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-modules)[[93]](#footnote-94)..

As such, this section and illustration walks through what it takes to develop and deploy your own TEE-based module to an IoT Edge device.

More specifically, this section covers the following ten activities:

1. Setting up an Azure IoT environment with an edge device.
2. Setting up your development machine.
3. Configuring your development environment for the Open Enclave SDK.
4. Creating an Edge container C/C++ project.
5. Providing your registry credentials to the IoT Edge agent.
6. Reviewing the generated host program code for the Azure IoT Edge module.
7. Selecting your target architecture for the Azure IoT Edge module.
8. Building the Azure IoT Edge module.
9. Pushing the Azure IoT Edge module to your container registry.
10. Deploying the Azure IoT Edge module to your actual Azure IoT Edge device.

Each activity is described in order in the next sections.

**Note** For more information, see articles [Tutorial: Develop IoT Edge modules for Linux devices](https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-develop-for-linux)[[94]](#footnote-95) and [Tutorial: Develop a C IoT Edge module for Linux devices](https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-c-module)[[95]](#footnote-96).

#### Setting up an Azure IoT environment with an edge device

As already introduced, you will create an Azure IoT Edge device using a Linux VM.

For that purpose, perform all the instructions outlined in section $ **Setting up a core Azure IoT environment** in Appendix A. Prerequisites and additional configuration.

Once completed, you will now need to setup your development machine to connect to your Azure IoT environment and your newly created Azure IoT Edge device. This is the purpose of the next section.

#### Setting up your development machine

This section describes how to configure your development environment. As such, and as already outlined, when developing IoT Edge modules, it's important to understand the different concepts between the development machine and the target IoT Edge device where the module will eventually be deployed.

Microsoft recommend that you don't run IoT Edge on your development machine, but instead use a separate device. This distinction between development machine and IoT Edge device more accurately mirrors a true deployment scenario and helps to keep the different concepts straight.

However, for the sake of simplicity, we will here use the same Ubuntu VM in Azure: IoT Edge modules are packaged as containers, so you need a container engine on your development machine to build and manage the Linux containers of the edge device. A container engine, i.e. Docker CE, is already up and running on this Ubuntu VM.

You will use on your local Windows 10 machine your Visual Studio Code IDE to connect to this Ubuntu VM and do TEE-based applications cross-development with it.

In the previous illustration of this module (see section § **Building a TEE-based Linux application on a simulated ARM TrustZone environment**), you already installed Visual Studio Code along with the Visual Studio Code Remote Development Extension Pack that allows you to open any folder in a container on a remote machine like your Ubuntu VM.

##### Installing the Azure IoT Tools extension pack

In addition, you will now need to install the [Azure IoT Tools extension pack](https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-tools)[[96]](#footnote-97) to help developing IoT Edge modules.

These extensions make it easy to discover and interact with Azure IoT Hub that power your Azure IoT Edge device(s), provide project templates, automate the creation of the deployment manifest, and allow you to monitor and manage your Azure IoT Edge device(s):

* The [Azure IoT Hub Toolkit](https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-toolkit)[[97]](#footnote-98) extension allows you to interact with an Azure IoT Hub, manage connected Azure IoT Edge devices, and enable distributed tracing for your Azure IoT applications.
* The [Azure IoT Edge for Visual Studio Code](https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-edge)[[98]](#footnote-99) allows you to easily code, build, and debug your custom logic and deploy it to your Azure loT Edge devices.

You will install The Azure IoT Tools extension pack, then set up your Azure account to manage IoT Hub resources from within Visual Studio Code.

**Note** For more information, see article [Use Visual Studio Code to develop and debug modules for Azure IoT Edge](https://docs.microsoft.com/en-us/azure/iot-edge/how-to-vs-code-develop-module)[[99]](#footnote-100)..

Perform the following steps:

1. Launch Visual Studio Code.
2. select **View** > **Extensions**.
3. Search for **Azure IoT Tools**, which is actually a collection of extensions that help you interact with your IoT Hub and your devices, as well as developing IoT Edge modules.
4. Select **Install**. Each included extension installs individually.

As previously stated, you will use the Ubuntu VM also as your Linux development machine. Since it’s should be already up and running if you followed the steps in order, Let’s now connect it to your Visual Studio Code IDE.

##### Connecting to your remote Linux development machine

As you already did, you will use the Visual Studio Code Remote Extension Pack to connect your Visual Studio Code IDE to your development machine, i.e. to your Ubuntu VM.

To do so, perform the following steps:

1. From Visual Studio Code, press F1 or CTRL-Shift-P to open the Command Palette, and run **Remote-SSH: Open Configuration File...**
2. Select the SSH config file you wish to update with the Ubuntu VM information, for example *%USERPROFILE%\.ssh\config* used before and add a host entry in the config file for this machine as follows:

# Read more about SSH config files: https://linux.die.net/man/5/ssh\_config

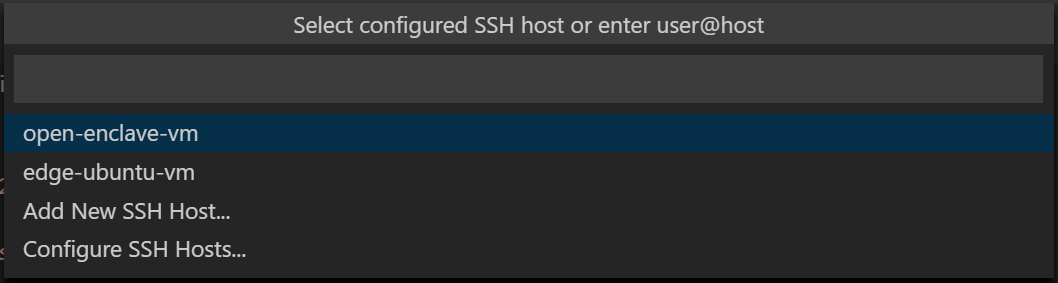
Host <*name-of-your-edge-ubuntu-vm-here*>

User <*your-user-name-on-your-edge-ubuntu-vm*>

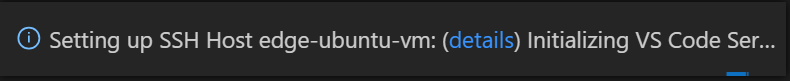
HostName <*edge-ubuntu-vm-ip-goes-here*>

IdentityFile c:\users\<*your-user-name-on-Windows10*>\.ssh\id\_rsa

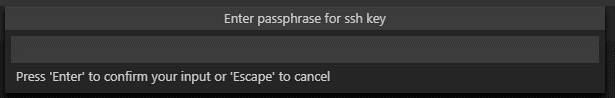
1. Press CTRL+S to save the config file.
2. Open a remote SSH session to your Ubuntu VM as previously done:
3. Press F1 or CTRL-Shift-P to specify a command, run Remote-SSH: Connect to SSH Host..., and select your Ubuntu VM name in the list.



1. A SSH session to your Ubuntu VM is opening. Visual Studio Code will now continue to configure it.



1. When prompted, type your passphrase for your SSH key and press ENTER.



Once finished, and like before, you now see a SSH indicator in the bottom left corner, and you'll be able to use Visual Studio Code as you would normally!



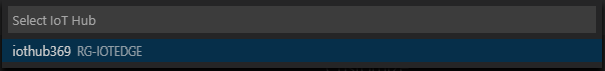
##### Configuring the Azure IoT Hub extension

You will install now set up your Azure account to manage IoT Hub resources from within Visual Studio Code.

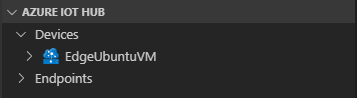
**Note** For more information, see article [Use Visual Studio Code to develop and debug modules for Azure IoT Edge](https://docs.microsoft.com/en-us/azure/iot-edge/how-to-vs-code-develop-module)[[100]](#footnote-101)..

Perform the following steps:

1. Open the command palette by selecting **View** > **Command Palette**.
2. In the command palette, search for and select **Azure: Sign in**. A browsing session opens. Follow the prompts to sign in to your Azure account.
3. In the command palette again, search for and select **Azure IoT Hub: Select IoT Hub**. Follow the prompts to select your Azure subscription, and then your IoT hub.



1. Open the explorer section of Visual Studio Code by either selecting the icon in the activity bar on the left, or by selecting **View** > **Explorer**.
2. At the bottom of the explorer section, expand the collapsed **Azure IoT Hub Devices** menu. You should see your IoT Edge device associated with the IoT hub that you selected through the command palette.



Let’s now configure your (remote) Linux development machine for the Open Enclave SDK.

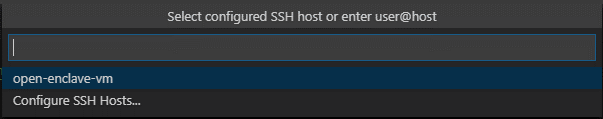
#### Configuring your development environment for the Open Enclave SDK

The Open Enclave SDK requires a series of prerequisites to be fulfilled on your remote Linux development machine before creating an Edge container project from Visual Studio Code.

Let’s see how to complete the configuration of your remote Linux development machine.

Perform the following steps:

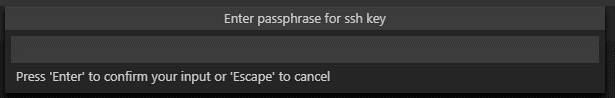
1. From Visual Studio, open a remote SSH session to your Ubuntu VM:
   1. Press F1 or CTRL-Shift-P to specify a command, run Remote-SSH: Connect to SSH Host..., and select your DC-series VM name in the list.



A SSH session to your Ubuntu VM is opening. Visual Studio Code will now continue to configure it.



* 1. When prompted, type your passphrase for your SSH key and press ENTER.



Once finished, you now see a SSH indicator in the bottom left corner, and you'll be able to use Visual Studio Code as you would normally!



ET voila! Any Visual Studio Code operations you perform in this window will be executed in the SSH environment, everything from editing and file operations, to debugging, using terminals, and more.

1. Ensure that the [requirements](https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave#Requirements)[[101]](#footnote-102) are met for the Open Enclave extension for Visual Studio Code:
   1. From Visual Studio Code, click on **Terminal** > **New Terminal** to open a Bash prompt on the Ubuntu VM.
   2. Make sure that the [Visual Studio Code Native Debug extension](https://marketplace.visualstudio.com/items?itemName=webfreak.debug)[[102]](#footnote-103) is installed. This should be the case.
   3. Install the required build components:

sudo apt update && sudo apt install -y build-essential cmake gcc-arm-linux-gnueabihf gcc-aarch64-linux-gnu g++-arm-linux-gnueabihf g++-aarch64-linux-gnu gdb-multiarch python

* 1. Install [CMake 3.12 or above](https://cmake.org/download/)[[103]](#footnote-104), currently 3.15:
     1. From the terminal console, remove the installed version if any:

sudo apt purge cmake

* + 1. Download the binary distribution archive file *cmake-3.15.3.tar.gz*, unpack it, and go to the folder of cmake:

wget http://www.cmake.org/files/v3.15/cmake-3.15.3-Linux-x86\_64.tar.gz

tar -xvzf cmake-3.15.3-Linux-x86\_64.tar.gz

cd cmake-3.15.3-Linux-x86\_64/

* + 1. Go to the folder of cmake and from there run the following commands:

sudo cp -r bin /usr/

sudo cp -r share /usr/

sudo cp -r doc /usr/share/

sudo cp -r man /usr/share/

* + 1. Go outside the folder of cmake and run the following commands:

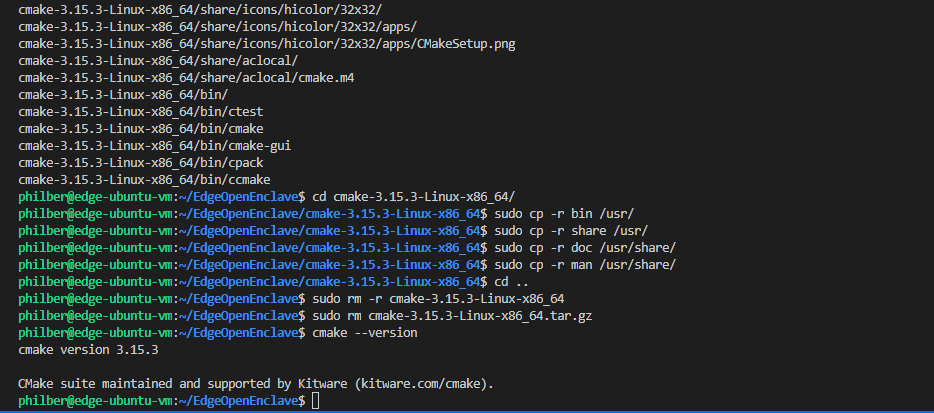
cd ..

sudo rm -r cmake-3.15.3-Linux-x86\_64

sudo rm cmake-3.15.3-Linux-x86\_64.tar.gz

Verify the cmake version using the following command:

cmake --version



1. Ensure that the [requirements](https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave#Requirements)[[104]](#footnote-105) are met for the Azure IoT Edge extension for Visual Studio Code:
   1. On your Linux development machine, for Edge Container projects, you will need to enable cross-building. The container that you will build to hold your TEE-based module code must match the operating system (OS) of the edge device (and target the underlying platform). (You should keep in mind the difference between the development machine OS and the container OS.)

To do so, on Ubuntu 16.04, run the followings:

sudo apt-get install -y qemu qemu qemu-system-misc qemu-user-static qemu-user binfmt-support

sudo mkdir -p /lib/binfmt.d

sudo sh -c 'echo :qemu-arm:M::\\x7fELF\\x01\\x01\\x01\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x02\\x00\\x28\\x00:\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\xfe\\xff\\xff\\xff:/usr/bin/qemu-arm-static:F > /lib/binfmt.d/qemu-arm-static.conf'

sudo sh -c 'echo :qemu-aarch64:M::\\x7fELF\\x02\\x01\\x01\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x02\\x00\\xb7\\x00:\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\xfe\\xff\\xff\\xff:/usr/bin/qemu-aarch64-static:F > /lib/binfmt.d/qemu-aarch64-static.conf'

sudo systemctl restart systemd-binfmt.service

* 1. Add your Linux user to the docker group. This will allow your user to connect and issue commands to the Docker daemon:

sudo usermod -aG docker $USER

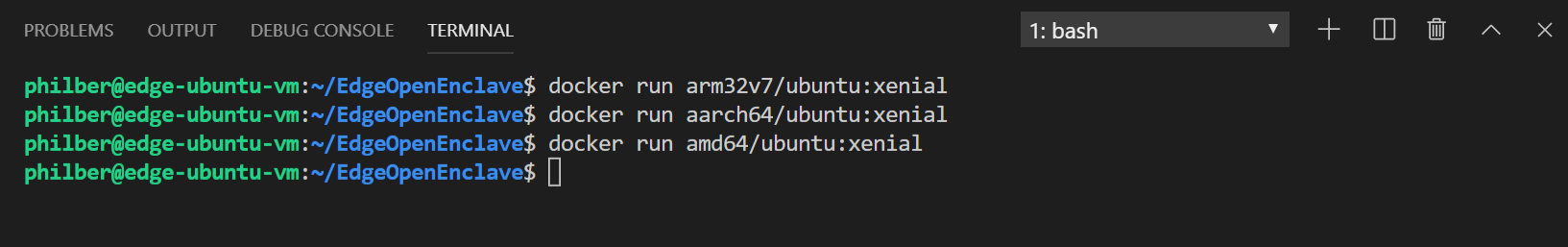
This command requires to log out and log back in so that your group membership is re-evaluated. With your Ubuntu VM, you will need to restart the VM for changes to take effect. Restart the Ubuntu VM and reopen a Bash terminal from Visual Studio Code.

* 1. To validate that your system is configured for cross-building, try testing the docker containers that will be later needed in the build:

docker run arm32v7/ubuntu:xenial

docker run aarch64/ubuntu:xenial

docker run amd64/ubuntu:xenial



* 1. Install the [Azure IoT EdgeHub Dev Tool](https://pypi.org/project/iotedgehubdev/)[[105]](#footnote-106) (iotedgehubdev).

This tool provides a local development experience with a simulator for creating, developing, testing, running, and debugging Azure IoT Edge modules and solutions.

sudo apt install pip-python

pip install --upgrade iotedgehubdev

1. Back in Visual Studio code, on the menu bar, install the Open Enclave extension for Visual Studio Code. When prompted, click on **Install**.



1. Use the Microsoft Open Enclave: Check System Requirements command - commands can be found using F1 or CTRL-Shift-P - to validate your system.

The command will query whether the required tools and the required versions are present on your system. Any unmet requirements will be presented in a Visual Studio Code warning window.

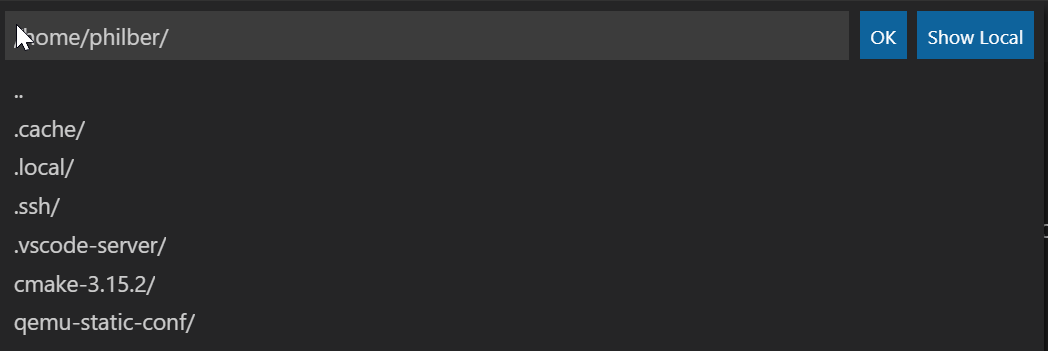
Otherwise, you should see instead the message “System meets the requirements”.



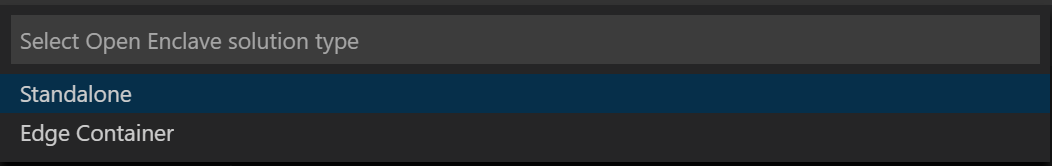
#### Creating an Edge container C/C++ project

Perform the following steps:

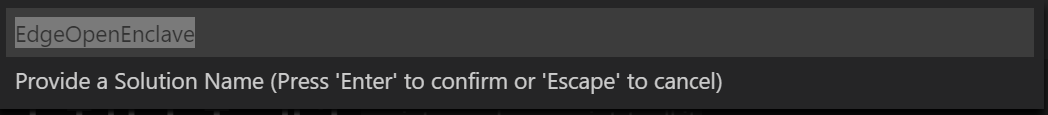
1. Launch Visual Studio Code.
2. As already stated, the Visual Studio Code Remote - SSH extension let you use a remote Linux-based Azure IoT Edge environment as your full-time development environment right from Visual Studio Code.
3. Use the Microsoft Open Enclave: New Open Enclave Solution command - commands can be found using F1 or CTRL-Shift-P - to create your first new Azure IoT Edge Module project.



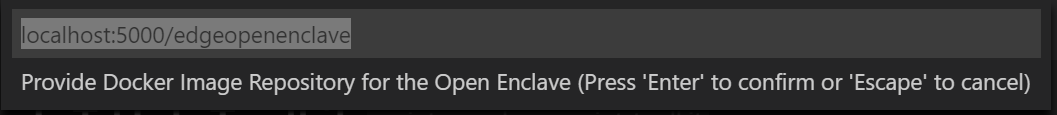
1. Specify the folder in which you want to create the application.



1. When invited to choose the option, create a **Edge container** project.



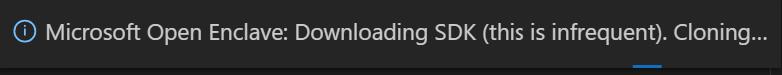
1. Provide a name for the host application/enclave, for example the default “EdgeOpenEnclave” name in our illustration, and press ENTER to confirm.

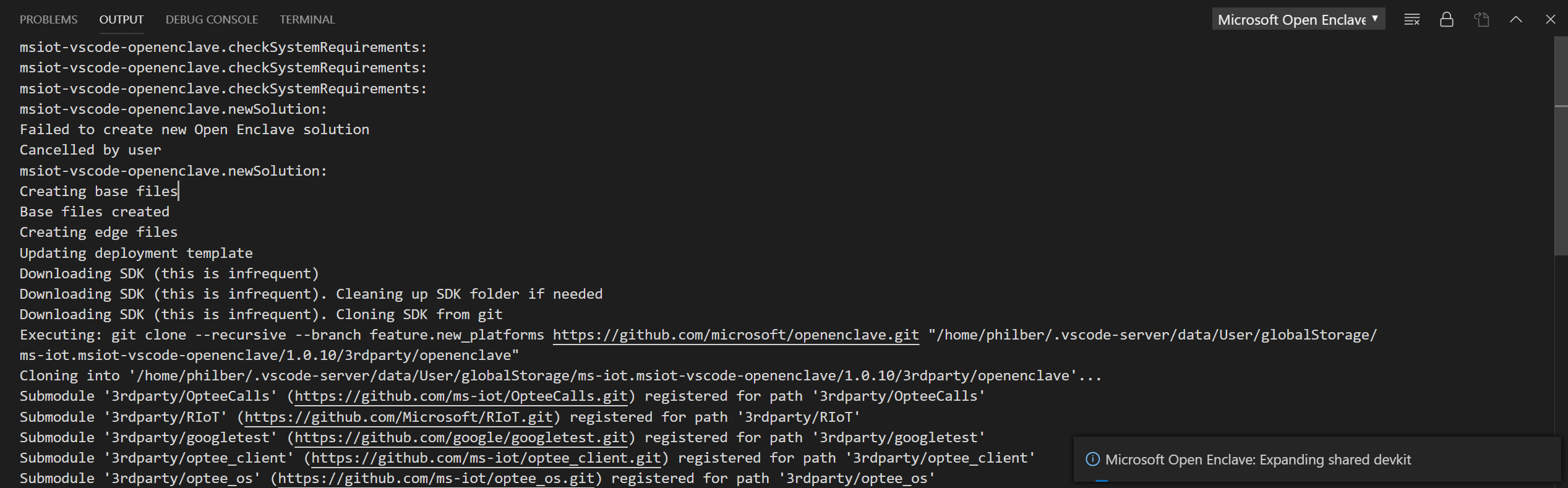


1. Provide your image repository. An image repository includes the name of your container registry and the name of your container image.

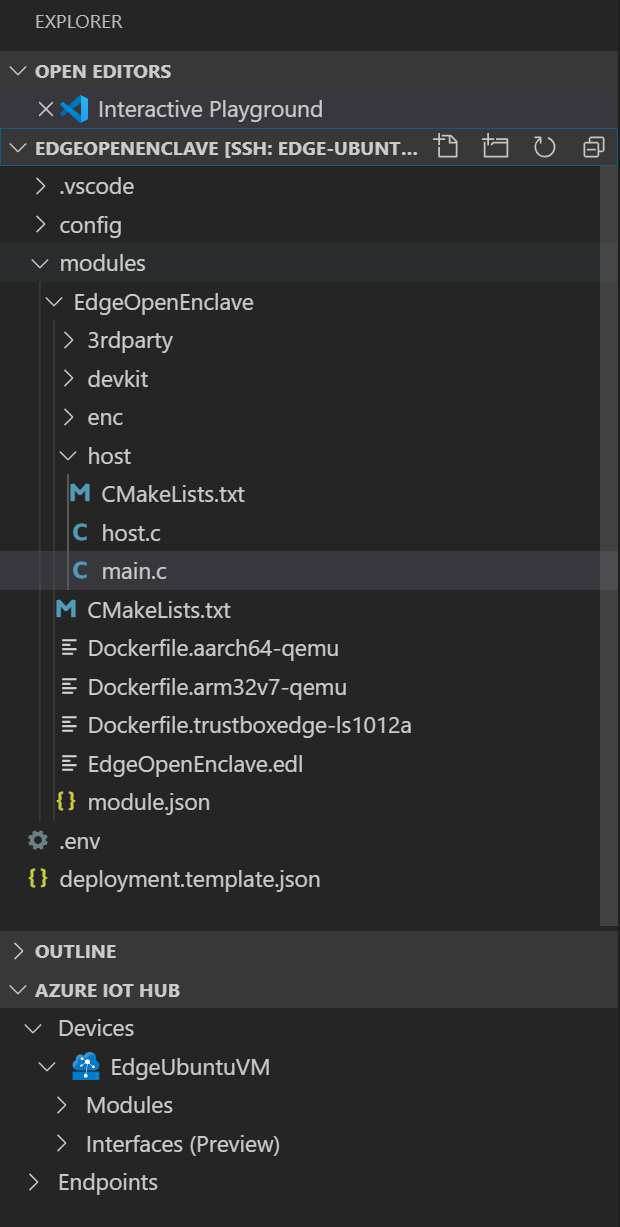
Your container image is prepopulated from the name you provided in the last step. Replace **localhost:5000** with the login server value from your Azure container registry, for example, iotedge369.azure.io in our illustration, see section § **Creating an Azure Container Registry** in Appendix A. Prerequisites and additional configuration. You can retrieve the login server from the **Overview** page of your container registry in the Azure portal.

The final image repository looks like <*registry name*>.azurecr.io/edgeopenenclave. For example, in our illustration: iotedge369.azurecr.io/edgeopenenclave.





A new solution will be created in the folder you've selected.



The solution will contain all the files and code that you need to deploy a working module to test on your Azure IoT Edge device, or give you a starting point to customize it with your own business logic.

As such, that solution will contain both the host and enclave as well as the required EDL file. The solution appears to somehow like the one in the previous activity, where the code that pertains to TEEs is mostly the same. However, the host will include some code that implements the required Azure IoT Hub communication.

Once your new solution loads in the Visual Studio Code window, take a moment to familiarize yourself with the files that it created:

* The *.vscode* folder contains a file called *launch.json*, which is used for debugging modules.
* The *modules* folder contains a folder for each module in your solution. Right now, that should only be *EdgeOpenEnclave*, or whatever name you gave to the module in the above step #6. The *EdgeOpenEnclave* folder contains both the host and enclave program code as well as the required EDL file.

The EDL file is as follows:

// Copyright (c) Microsoft Corporation. All rights reserved.

// Licensed under the MIT License.

enclave {

from "openenclave/stdio.edl" import \*;

trusted {

/\* define ECALLs here. \*/

public int ecall\_handle\_message([in, string] char \*input\_msg, [out, count=enclave\_msg\_size] char \*enclave\_ms, unsigned int enclave\_msg\_size);

};

untrusted {

/\* define OCALLs here. \*/

int ocall\_log([in, string] char \*msg);

};

};

The program code appears to somehow like the one in the previous activity. However, the host will include some code that implements the required Azure IoT Hub communication. As notice above, it also contains the module metadata, and several Docker files.

* The *.env* file holds the credentials to your container registry. These credentials are shared with your Azure IoT Edge device, i.e. your Ubuntu VM, so that it has access to pull the container images.
* The *deployment.debug.template.json* file and *deployment.template.json* file are templates that help you create a deployment manifest (see section § **Pushing the Azure IoT Edge module** below).

A deployment manifest is a file that defines exactly which modules you want deployed on an edge device, how they should be configured, and how they can communicate with each other and the cloud. The template files use pointers for some values. When you transform the template into a true deployment manifest, the pointers are replaced with values taken from other solution files. Locate the two common placeholders in your deployment template:

* + In the **registryCredentials** section, the address is auto filled from the information you provided when you created the solution. However, the username and password reference the variables stored in the *.env* file. This is for security, as the *.env* file is git ignored, but the deployment template is not.
  + In the EdgeOpenEnclave section, the container image isn't filled in even though you provided the image repository when you created the solution. This placeholder points to the *module.json* file inside the *EdgeOpenEnclave* folder.

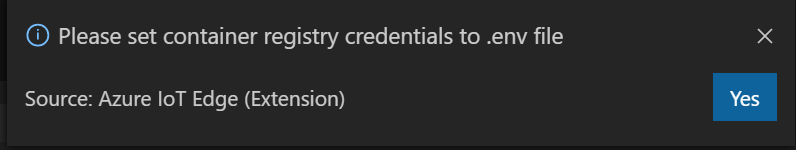
If you go to that file, you'll see that the image field does contain the repository, but also a tag value that is made up of the version and the platform(s) of the container.

You can iterate the version manually as part of your development cycle, and you select the container platform using a switcher that we introduce later in this section.

#### Providing your registry credentials to the IoT Edge agent

The environment file stores the credentials for your container registry and shares them with the IoT Edge runtime. The runtime needs these credentials to pull your container images onto the IoT Edge device.

On Visual Studio Code, the IoT Edge extension tries to pull your container registry credentials from Azure and populate them in the environment file. You may see the following popup during the creation of the above solution.



Check to see if your credentials are already included. If not, add them now.

Perform the following steps:

* Open the ***.env*** file in your module solution.

iotedge369\_USERNAME=

iotedge369\_PASSWORD=

* Add the **username** and **password** values that you copied from your Azure container registry, i.e. the values registry-username and registry-password, see section § **Creating an Azure Container Registry** in Appendix A. Prerequisites and additional configuration.
* Save your changes to the *.env* file.

#### Reviewing the generated host program code for the Azure IoT Edge module

As aforementioned, the host include some code that implements the required Azure IoT Hub communication.

This code simply receives messages and then passes them on. The pipeline functionality demonstrates here an important concept in IoT Edge, which is how modules communicate with each other.

Each module can have multiple input and output queues declared in their code. The IoT Edge hub running on the device routes messages from the output of one module into the input of one or more modules. The specific language for declaring inputs and outputs varies between languages, but the concept is the same across all modules.

**Note** For more information about routing between modules, see article [Declare routes](https://docs.microsoft.com/en-us/azure/iot-edge/module-composition#declare-routes)[[106]](#footnote-107)..

The C code that comes with the project template uses the [IoTHubModuleClient\_LL module](https://docs.microsoft.com/en-us/azure/iot-hub/iot-c-sdk-ref/iothub-module-client-ll-h/iothubmoduleclient-ll-setinputmessagecallback)[[107]](#footnote-108) from the Azure IoT C SDK:

* Open the ***main.c*** file, which is inside the folder ***modules/EdgeOpenEnclave/host***.
* In the *main.c* file, find the **IoTHubModuleClient\_LL\_SetInputMessageCallback** function. The [IoTHubModuleClient\_LL\_SetInputMessageCallback](https://docs.microsoft.com/en-us/azure/iot-hub/iot-c-sdk-ref/iothub-module-client-ll-h/iothubmoduleclient-ll-setinputmessagecallback)[[108]](#footnote-109) function sets up an input queue callback to receive incoming messages. Review this function call and see how it initializes an input queue callback called **InputQueue1Callback**.

static int SetupCallbacksForModule(IOTHUB\_MODULE\_CLIENT\_LL\_HANDLE iotHubModuleClientHandle)

{

int ret;

if (IoTHubModuleClient\_LL\_SetInputMessageCallback(iotHubModuleClientHandle, "input1",

InputQueue1Callback, (void\*)iotHubModuleClientHandle) != IOTHUB\_CLIENT\_OK)

{

printf("ERROR:

IoTHubModuleClient\_LL\_SetInputMessageCallback(\"input1\")..........FAILED!\r\n");

ret = -1;

}

else

{

ret = 0;

}

return ret;

}

* Next, find the **IoTHubModuleClient\_LL\_SendEventToOutputAsync** function. The [IoTHubModuleClient\_LL\_SendEventToOutputAsync](https://docs.microsoft.com/en-us/azure/iot-hub/iot-c-sdk-ref/iothub-module-client-ll-h/iothubmoduleclient-ll-sendeventtooutputasync)[[109]](#footnote-110) function processes received messages and sets up an output queue to pass them along. Review this call and see that it initializes an output queue called **output1**.

static IOTHUBMESSAGE\_DISPOSITION\_RESULT SendEnclaveResponse(IOTHUB\_MODULE\_CLIENT\_LL\_HANDLE iotHubModuleClientHandle, char\* messageBodyStr)

{

IOTHUBMESSAGE\_DISPOSITION\_RESULT result;

IOTHUB\_CLIENT\_RESULT clientResult;

char\* enclaveMessage = (char\*)malloc(512 \* sizeof(char));

int enclaveResult = call\_enclave(messageBodyStr, enclaveMessage, 512);

if (enclaveResult != 0)

{

result = IOTHUBMESSAGE\_ABANDONED;

}

else

{

// This message should be sent to next stop in the pipeline, namely "output1". What happens

// at "outpu1" is determined by the configuration of the Edge routing table setup.

MESSAGE\_INSTANCE \*messageInstance = CreateMessageInstance(enclaveMessage);

if (NULL == messageInstance)

{

result = IOTHUBMESSAGE\_ABANDONED;

}

else

{

printf("Sending message (%zu) to the next stage in pipeline\n",

messagesReceivedByInput1Queue);

clientResult = IoTHubModuleClient\_LL\_SendEventToOutputAsync(iotHubModuleClientHandle,

messageInstance->messageHandle, "output1", SendConfirmationCallback,

(void \*)messageInstance);

if (clientResult != IOTHUB\_CLIENT\_OK)

{

IoTHubMessage\_Destroy(messageInstance->messageHandle);

free(messageInstance);

printf("IoTHubModuleClient\_LL\_SendEventToOutputAsync failed on sending msg#=%zu,

err=%d\n", messagesReceivedByInput1Queue, clientResult);

result = IOTHUBMESSAGE\_ABANDONED;

}

else

{

result = IOTHUBMESSAGE\_ACCEPTED;

}

}

}

free(enclaveMessage);

return result;

}

* Now, open the ***deployment.template.json*** file.
* Find the **modules** property of the $edgeAgent desired properties. There should be two modules listed here:

1. The first is **SimulatedTemperatureSensor**, which is included in all the templates by default to provide simulated temperature data that you can use to test your modules.
2. The second is the **EdgeOpenEnclave** module that you created as part of this solution.

* At the bottom of the file, find the desired properties for the **$edgeHub** module.

One of the functions of the IoT Edge hub module is to route messages between all the modules in a deployment. Review the values in the **routes** property:

1. The first route, i.e. **EdgeOpenEnclaveToIoTHub**, uses a wildcard character (**\***) to indicate any messages coming from any output queues in the EdgeOpenEnclave module. These messages go into $upstream, which is a reserved name that indicates IoT Hub.
2. The second route, i.e. sensorToEdgeOpenEnclave, takes messages coming from the SimulatedTemperatureSensor module and routes them to the input1 input queue that you saw initialized in the EdgeOpenEnclave code.

"$edgeHub": {

"properties.desired": {

"schemaVersion": "1.0",

"routes": {

"EdgeOpenEnclaveToIoTHub": "FROM /messages/modules/EdgeOpenEnclave/outputs/\* INTO $upstream",

"sensorToEdgeOpenEnclave": "FROM /messages/modules/tempSensor/outputs/temperatureOutput INTO

BrokeredEndpoint(\"/modules/EdgeOpenEnclave/inputs/input1\")"

},

"storeAndForwardConfiguration": {

"timeToLiveSecs": 7200

}

}

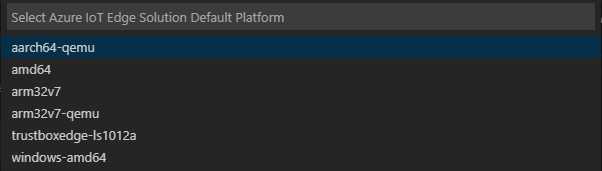
#### Selecting your target architecture for the Azure IoT Edge module

Currently, Visual Studio Code can develop C# modules for Linux ARMv7-A, AArch64/ARMv8-A, and TrustBoxEdge (LS1012a) devices.

You need to select which architecture you're targeting with each solution, because that affects how the Linux container is built and runs.

Proceed with the following steps:

1. Use Azure IoT Edge: Set Default Target Platform for Edge Solution - commands can be found using F1 or CTRL-Shift-P -.



1. In the command palette, select the target architecture from the list of options. Select aarch64-qemu for your Ubuntu VM. (If you were using an actual Scalys TrustBox Edge device, you would have selected here trustboxedge-ls1012a instead.)



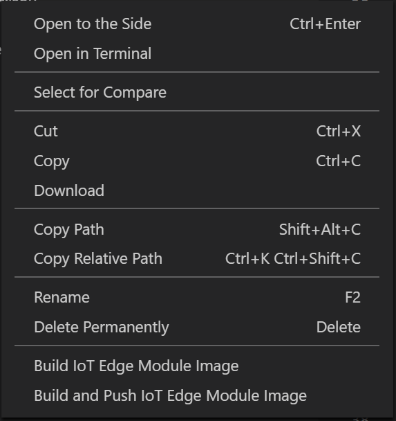
#### Building the Azure IoT Edge module

You've reviewed the module code for the host, and the deployment template to understand some key module and deployment concepts. Now, you're ready to build the EdgeOpenEnclave container image.

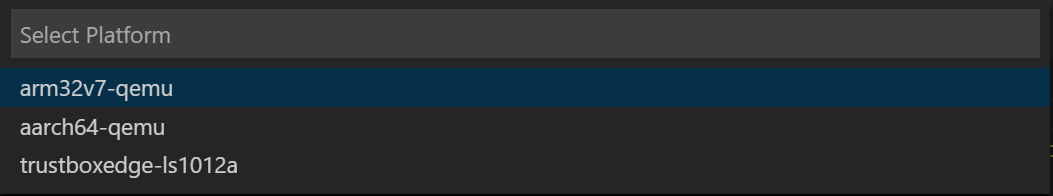
As already noticed, and considering previous requirements, the underlying system used in Visual Studio Code to build is CMake. For Azure IoT Edge projects, Ubuntu containers are used to configure and build. The build task will invoke docker and leverage project dockerfiles.

To build the Azure IoT Edge Module, perform the following steps:

1. Right-click on *modules/EdgeOpenEnclave/module.json*.



1. Select **Build IoT Edge Module Image**.



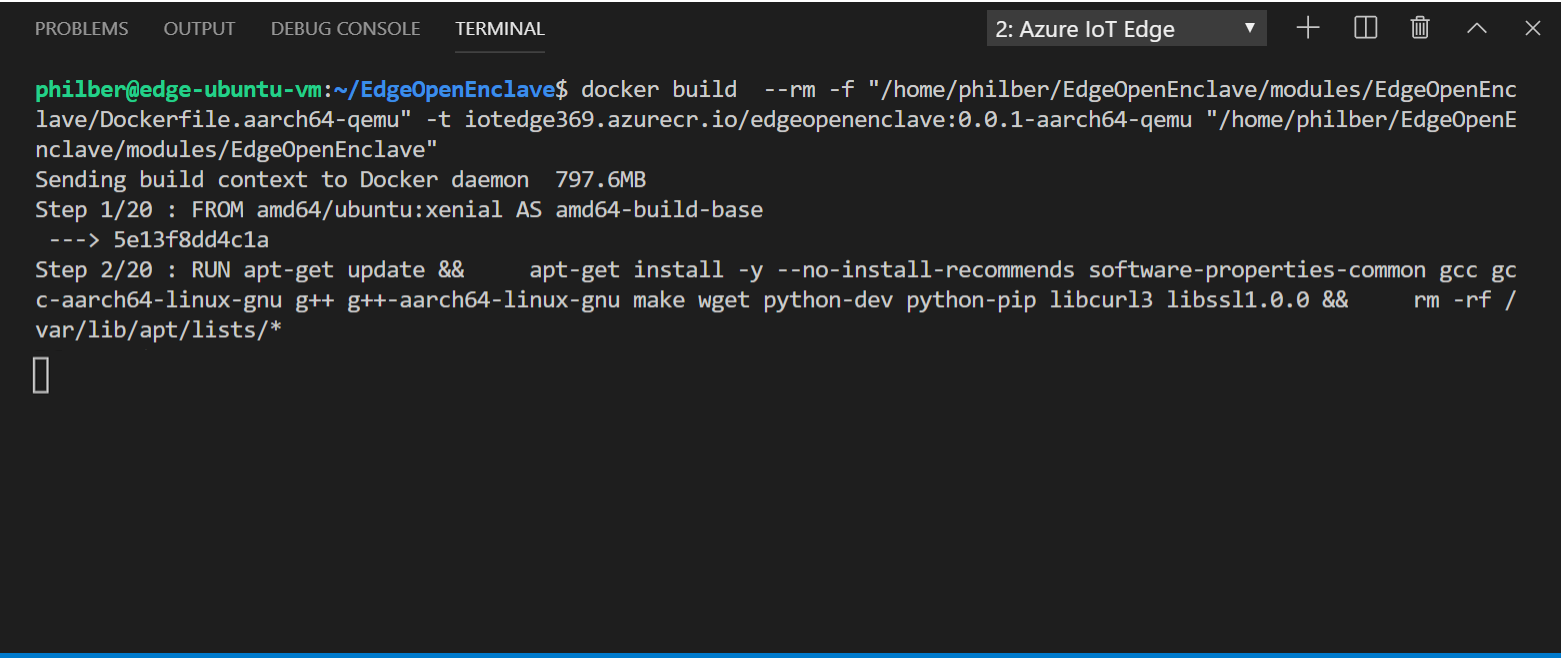
1. You should see tasks configured to build for each target: **ARMv7-A**, **AArch64/ARMv8-A**, and **TrustBoxEdge(LS1012a)**.
2. Select aarch64-quemu for your Ubuntu VM. (If you were using an actual Scalys TrustBox Edge device, you would have selected here trustboxedge-ls1012a instead.)

This “build” command starts two operations:

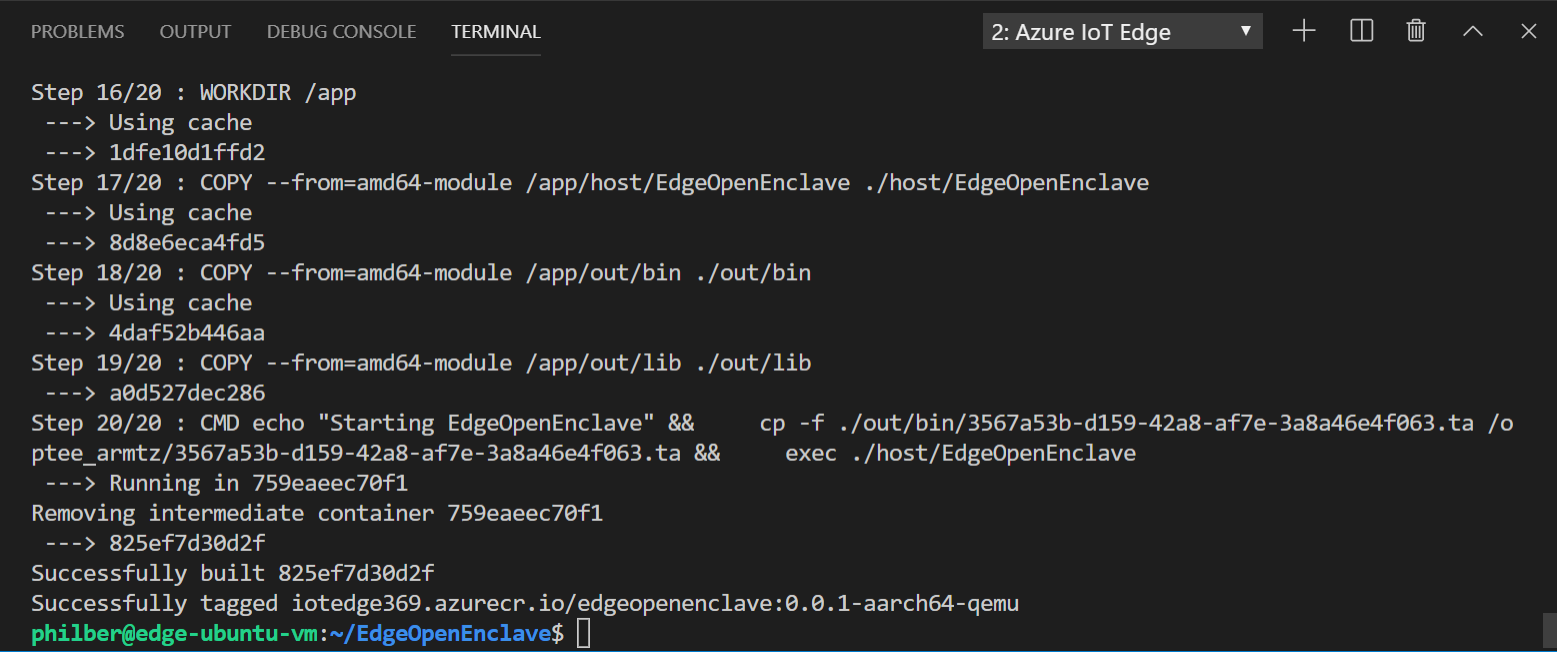
* 1. First, it creates a new folder in the solution called *config* that holds the full deployment manifest, built out of information in the deployment template and other solution files.
  2. Second, it runs docker build to build the container image based on the appropriate dockerfile for your target architecture: i.e. *Dockerfile.aarch64-qemu* here.

As far as the latter are concerned, this will result in executing the following command:

docker build --rm -f "/home/philber/EdgeOpenEnclave/modules/EdgeOpenEnclave/Dockerfile.aarch64-qemu" -t iotedge369.azurecr.io/edgeopenenclave:0.0.1-aarch64-qemu "/home/philber/EdgeOpenEnclave/modules/EdgeOpenEnclave"



This process may take several minutes the first time but is faster the next time that you run the commands.



#### Pushing the Azure IoT Edge module to your container registry

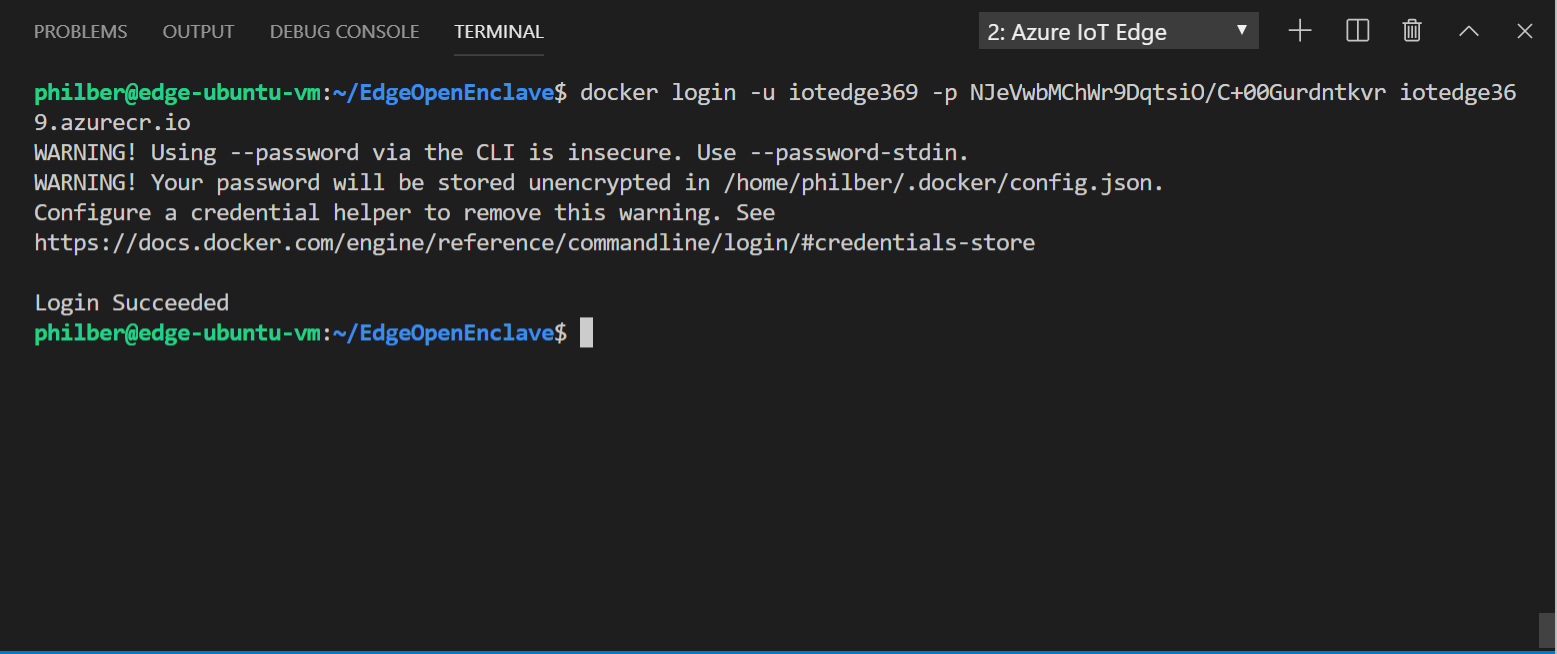
At this stage, you're ready to build the EdgeOpenEnclave container image and push it to your container registry. With the IoT tools extension for Visual Studio Code, this step also generates the deployment manifest based on the information in the template file and the module information from the solution files.

Deploying your Edge Container project's enclave is easy as you will see now.

Perform the following steps:

1. Provide your container registry credentials to Docker so that it can push your container image to be stored in your container registry:
   1. Open the Visual Studio Code integrated terminal by selecting **View** > **Terminal**.
   2. Sign into Docker with the Azure container registry credentials that you saved after creating the registry, see section § **Creating an Azure Container Registry** in Appendix A. Prerequisites and additional configuration.

docker login -u <registry-username> -p <registry-password> <registry-login-server>



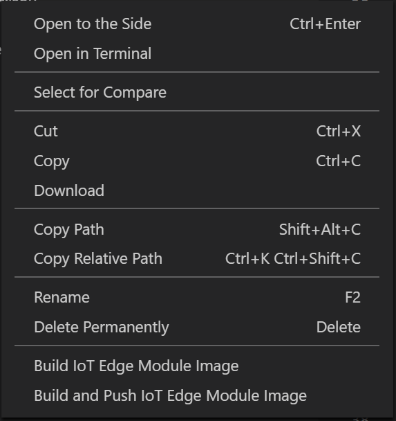
**Note** You may receive a security warning recommending the use of --password-stdin option. The --password-stdin option prevents your password from appearing in the command line history. While that best practice is recommended for production scenarios, it's outside the scope of this walkthrough. For more information, see article [docker login](https://docs.docker.com/engine/reference/commandline/login/#provide-a-password-using-stdin)[[110]](#footnote-111).

Once you've logged into docker, you can log out (and remove your credentials from the system) by:

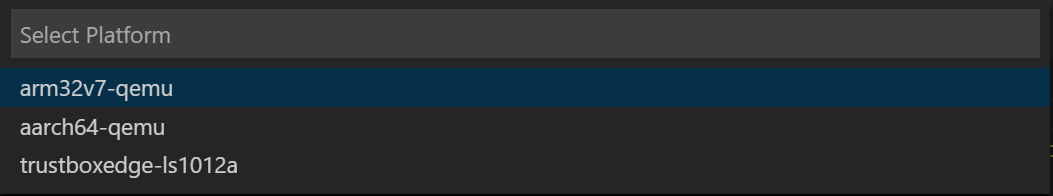
docker logout <*container-url*>

Visual Studio Code now has access to your container registry, so it's time to turn the solution code into a container image.

1. Right-click on *modules/EdgeOpenEnclave/module.json*.



1. Select **Build and Push IoT Edge Module Image**.



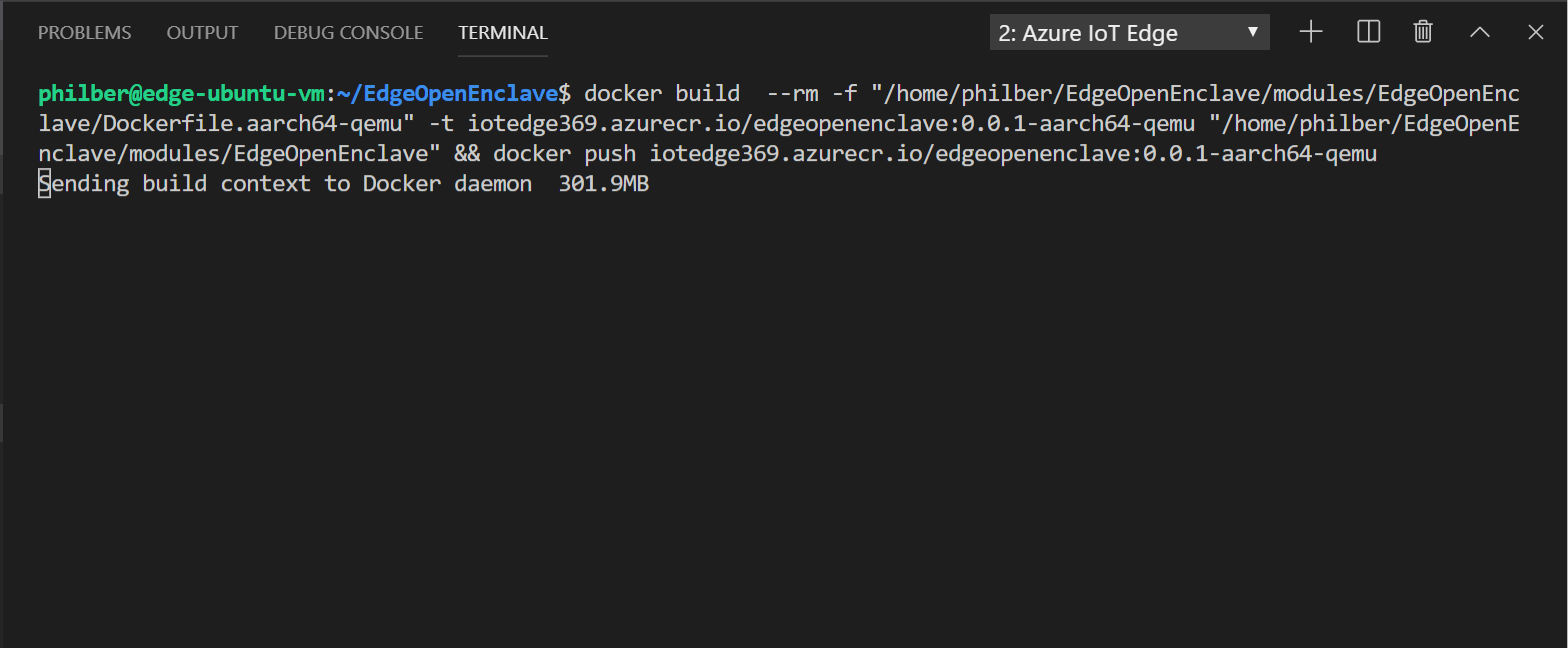
1. You should see tasks configured to build for each target: **ARMv7-A**, **AArch64/ARMv8-A**, and **TrustBoxEdge(LS1012a)**.
2. Select aarch64-quemu for your Ubuntu VM. (If you were using an actual Scalys TrustBox Edge device, you would have selected here trustboxedge-ls1012a instead.)

This “build and push” command starts three operations:

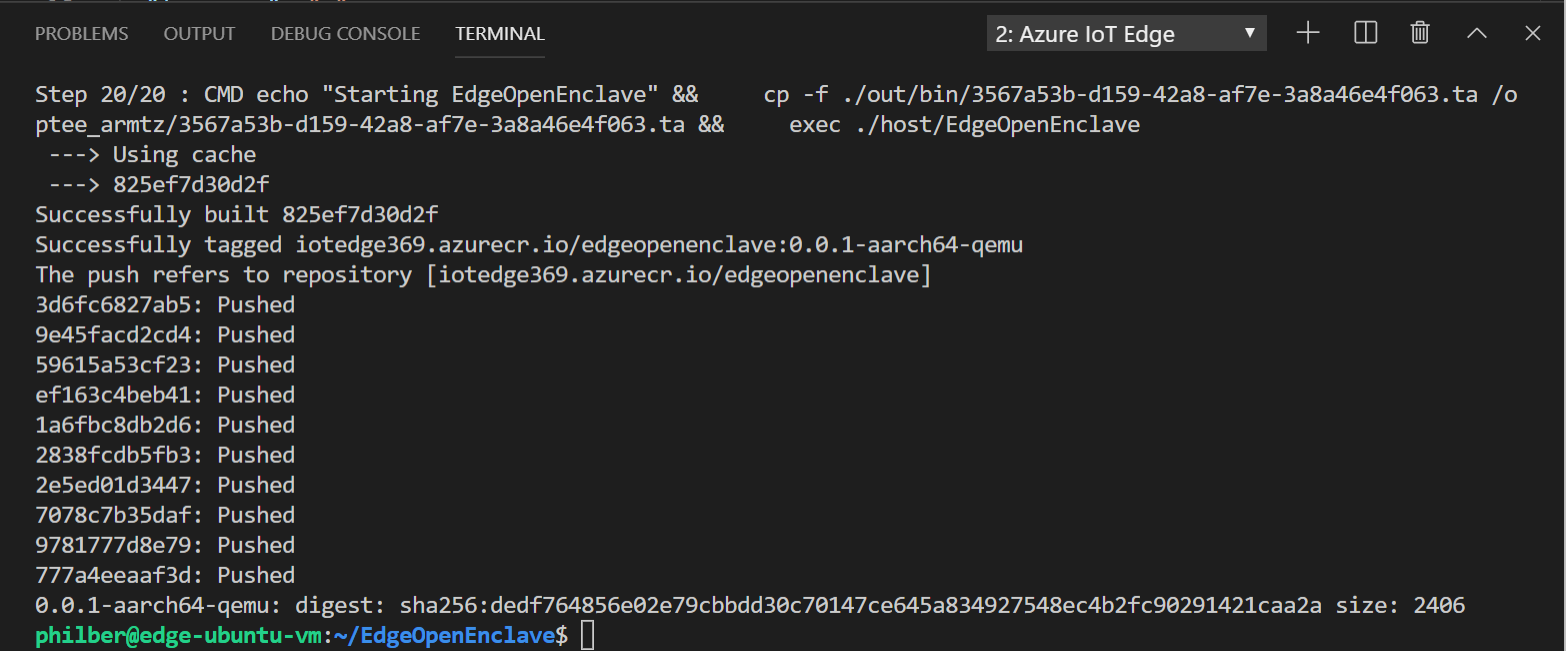
* 1. First, it creates a new folder in the solution called *config* that holds the full deployment manifest, built out of information in the deployment template and other solution files.
  2. Second, it runs docker build to build the container image based on the appropriate dockerfile for your target architecture: i.e. *Dockerfile.aarch64-qemu* here.
  3. Then, it runs docker push to push the image repository to your container registry, i.e. iotedge369.azure.io.

As far as the latter are concerned, this will result in executing the following command:

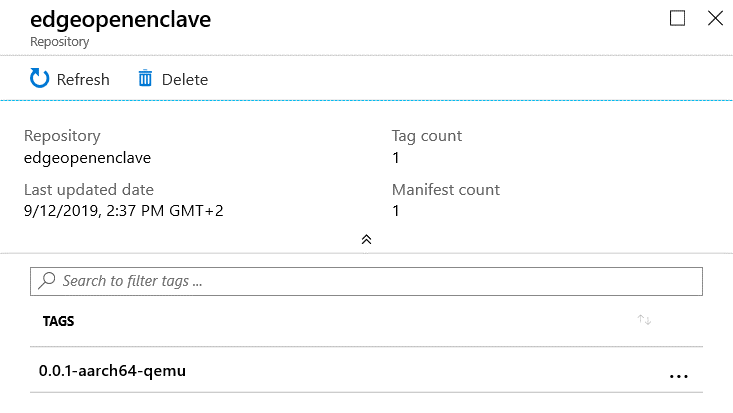
docker build --rm -f "/home/philber/EdgeOpenEnclave/modules/EdgeOpenEnclave/Dockerfile.aarch64-qemu" -t iotedge369.azurecr.io/edgeopenenclave:0.0.1-aarch64-qemu "/home/philber/EdgeOpenEnclave/modules/EdgeOpenEnclave" && docker push iotedge369.azurecr.io/edgeopenenclave:0.0.1-aarch64-qemu



This process may take several minutes the first time but is faster the next time that you run the commands.



1. At this stage, the container image has been pushed to your container registry in Azure.
   1. To further verify what the above build and push command did, go to the Azure portal at <https://portal.azure.com> and navigate to your container registry.
   2. In your container registry, select **Repositories** then **edgeopenenclave**.



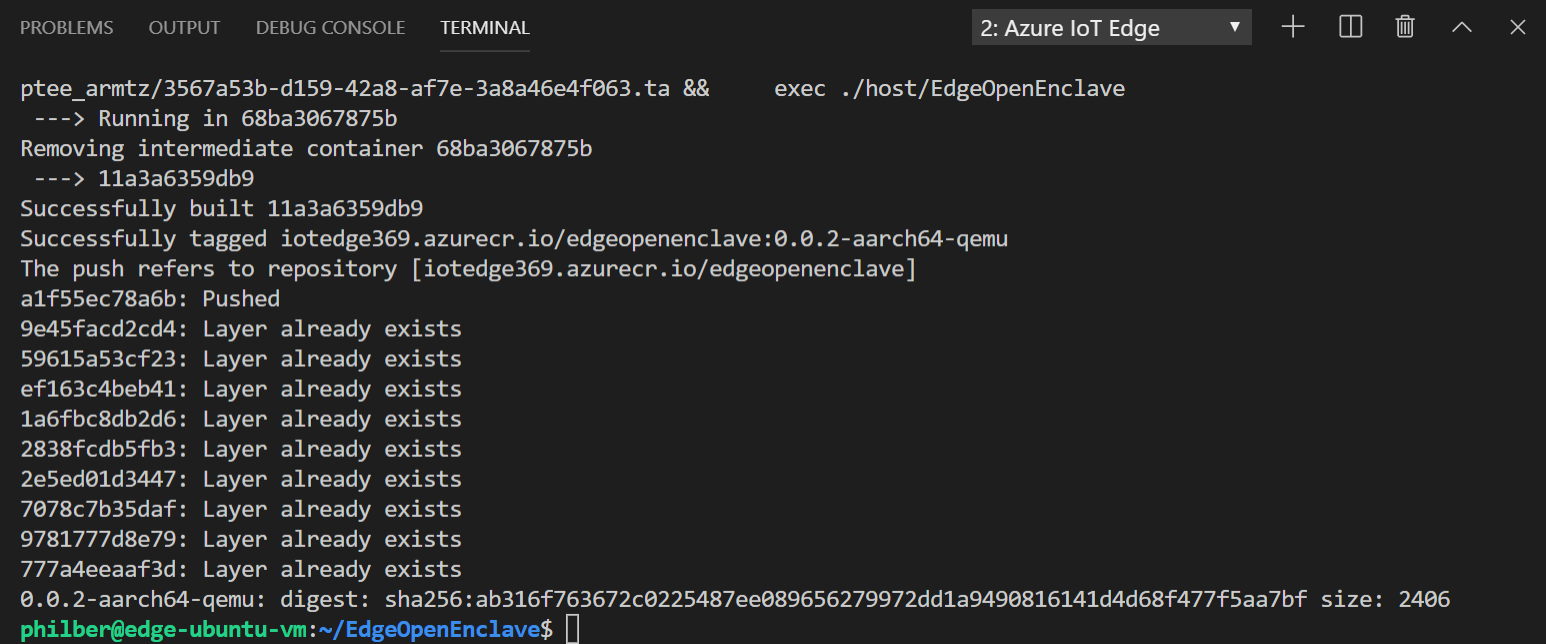
Likewise, Azure IoT Edge deployment template files have been provided.

Perform the following steps:

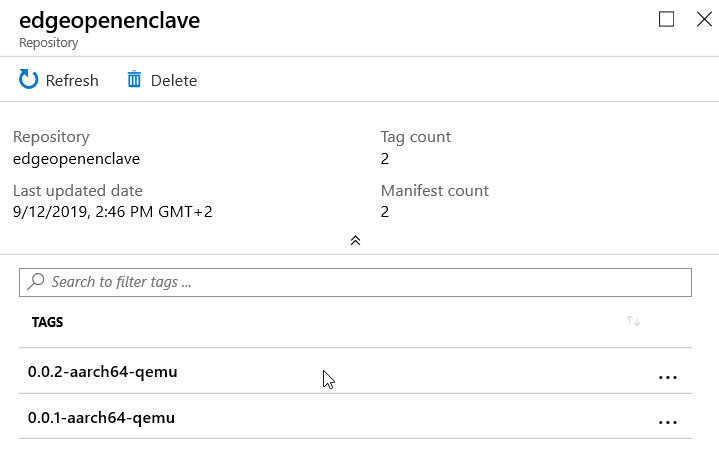
1. Open the ***deployment.aarch64-quemu.json*** file in newly created *config* folder. The filename reflects the target architecture, so it will be different if you chose a different architecture, for example for a real Scalys TrustBox Edge device.
2. Notice that the two parameters that had placeholders now are filled in with their proper values:
   1. The **registryCredentials** section has your registry username and password pulled from the .env file.
   2. The **EdgeOpenEnclave** has the full image repository with the name, version, and architecture tag from the module.json file.
3. Open the ***module.json*** file in the *EdgeOpenEnclave* folder.
4. Change the version number for the module image. (The version, not the $schema-version.) For example, increment the patch version number to **0.0.2** as though we had made a small fix in the module code.

**Note** Module versions enable version control and allow you to test changes on a small set of Azure IoT Edge devices before deploying updates to production. If you don't increment the module version before building and pushing, then you overwrite the repository in your container registry.

1. Save your changes to the *module.json* file.
2. Right-click the ***deployment.template.json*** file again, and select **Build and Push IoT Edge Solution**.



1. Open the ***deployment.aarch64-quemu.json*** file again. Notice that a new file wasn't created when you ran the build and push command again. Rather, the same file was updated to reflect the changes. The EdgeOpenEnclave image now points to the 0.0.2 version of the container.
2. To further verify what the build and push command did, in your container registry in Azure, select again **Repositories** then **edgeopenenclave**. Verify that both versions of the image were pushed to the registry.



If you encounter errors when building and pushing your module image, it often has to do with Docker configuration on your development machine. Use the following check list to review your configuration:

* Did you run the docker login command using the credentials that you copied from your container registry? These credentials are different than the ones that you use to sign in to Azure.
* Is your container registry correct? Does it have your correct container registry name and your correct module name? Open the ***module.json*** file in the *modules/EdgeOpenEnclave/*folder to check. The registry value should look like **<your registry name>.azurecr.io/edgeopenenclave**.
* If you used a different name than default **EdgeOpenEnclave** for your module, is that name consistent throughout the solution?
* You verified that the built container images are stored in your container registry, so it's time to deploy them to a (virtual) Azure IoT Edge device.

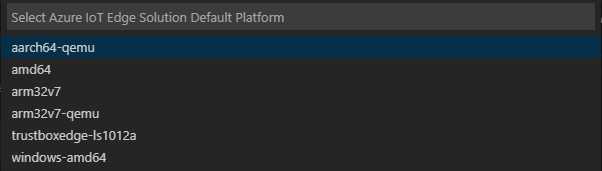
If everything ran well, you verified that the built container images are stored in your container registry, so it's time to deploy them to a (virtual) Azure IoT Edge device.

#### Deploying the Azure IoT Edge module to your actual Azure IoT Edge device

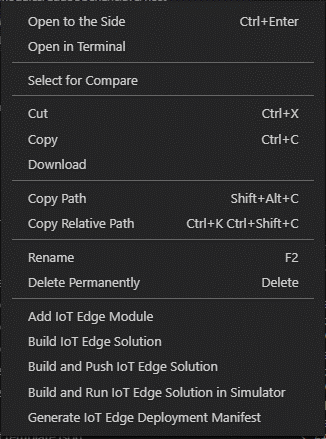
Make sure that your IoT Edge device is up and running.

To create a new deployment configuration based on the current settings in *module.json*, perform the following steps:

1. Use Azure IoT Edge: Set Default Target Platform for Edge Solution - commands can be found using F1 or CTRL-Shift-P - to create your first new Azure IoT Edge Module project.



1. Select aarch64-qemu for your Ubuntu VM. (If you were using an actual Scalys TrustBox Edge device, you would have selected here trustboxedge-ls1012a instead.)
2. Right-click on *deployment.template.json* (or *deployment.debug.template.json*).



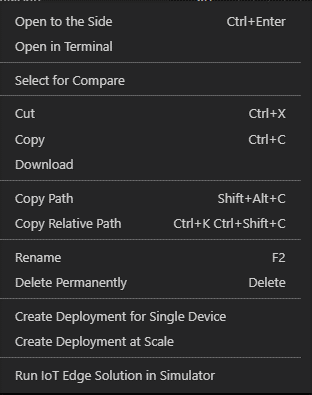
1. Select Generate IoT Edge Deployment Manifest. This will generate or replace the appropriate deployment .JSON file in the *config* folder.



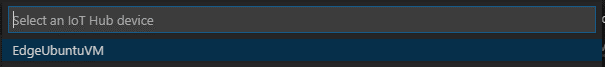
Once your deployment JSON file has been created in the *config* folder, you can deploy to an Azure Edge device.

Perform the following steps:

1. Navigate into to the *config* folder.
2. Right-click on the ***deployment.aarch64-quemu.json*** file. Do not use the *deployment.template.json* file, which doesn't have the container registry credentials or module image values in it.



1. Select Create Deployment for Single Device or Create Deployment at Scale.



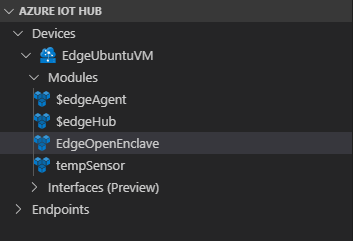
1. Select your Azure IoT Edge device and press ENTER.



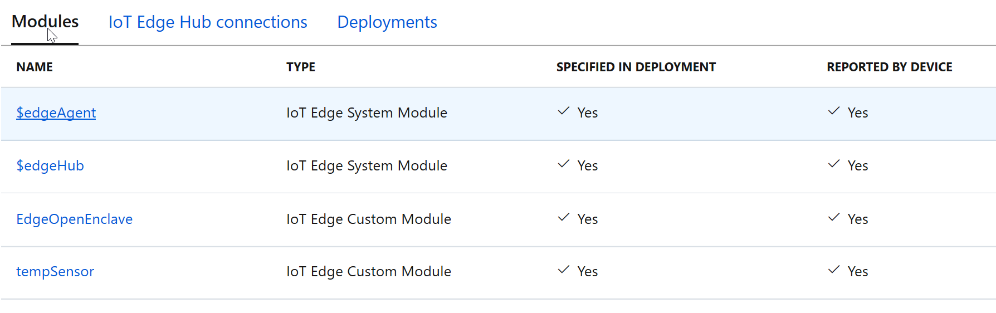
1. Et voila! This is all what it takes to deploy your Azure IoT Edge module onto your actual device.

It may take a few minutes for both modules to start. The Azure IoT Edge runtime needs to receive its new deployment manifest, pull down the module images from the container runtime, then start each new module.

1. In the Azure IoT Hub extension in Visual Studio Code:
   1. Expand the details for your Azure IoT Edge device, then expand the **Modules** list for your device.
   2. Use the refresh button to update the device view until you see the tempSensor (SimulatedTemperatureSensor) and EdgeOpenEnclave modules running on your device.



1. To further verify what the deployment command did against your Azure IoT Edge device:
   * Go again to the Azure portal at <https://portal.azure.com> and navigate this time to your Azure IoT Hub.
   * In your Azure IoT Hub, select **IoT Edge**, your device in list, and then **Set Modules**.
   * select again **Repositories** then **edgeopenenclave**. Verify that both versions of the image were pushed to the registry.



This concludes the third illustration of this second module.

**Now that you know the basics on how to get an up and running TEE-based application/module in the Intelligent Cloud or in the Intelligent Edge, let’s see how you can leverage such an approach to build trusted network. This is the purpose of the next module.**

# Module 3: Confidential Consortium Framework (CCF) setup and configuration on DC-series VMs

## Overview

In this last module, you will learn about the [Confidential Consortium Framework](https://aka.ms/ccf)[[111]](#footnote-112) (CCF). This project, led by Microsoft Research, is a Microsoft's open source framework based on Open Enclave (OE) for building a new category of secure, highly available, and performant applications that focus on multi-party compute and data.

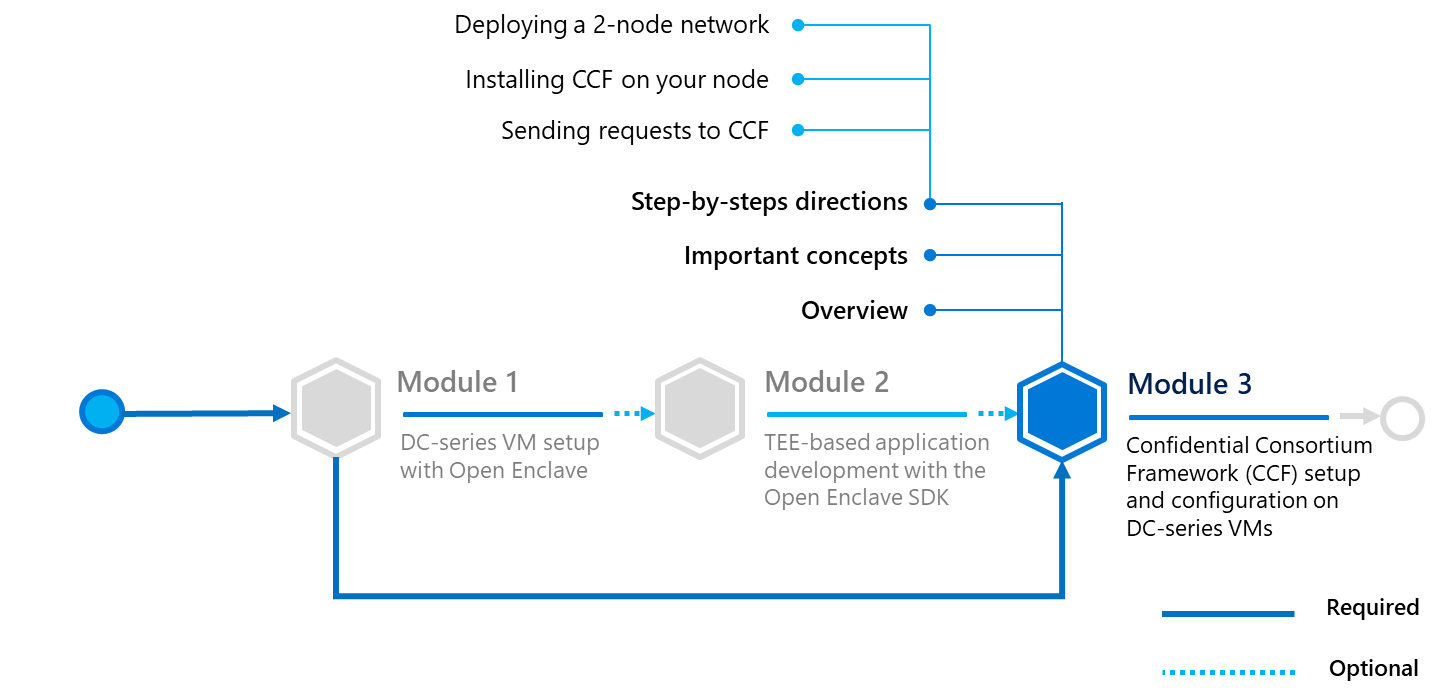
**Note** For an overview, watch the episodes [Confidential Consortium Framework (CCF) Overview](https://channel9.msdn.com/Shows/Blocktalk/Confidential-Consortium-Framework-CCF-Overview)[[112]](#footnote-113) and [Confidential Consortium Framework (CCF) Part II: A deeper look and demos](https://channel9.msdn.com/Shows/Blocktalk/Confidential-Consortium-Framework-CCF-Part-II-A-deeper-look-and-demos)[[113]](#footnote-114) on the Block Talk series on Channel 9.

Deployed as a blockchain network between parties, every node of the network is running a Trusted Execution Environment (TEE) or enclaves - you can re-read the introduction of this guide to learn about enclaves -.

This approach gives to CCF some key benefits, such as a great throughput, a low latency of transactions (approaching database speeds), confidentiality features, and last but not least a welcome ability to establish governance between parties.

You will see in this module how to:

* Create CCF nodes (by leveraging all the work you previously done work in section **Module 1: DC-series VM setup with Open Enclave**),
* Start and leverage the blockchain network between your newly created nodes.



Before doing that, let’s consider some important concepts.

## Important concepts

Before starting to create your nodes, let’s take the time to (shortly) remind or present you what blockchain (network) is and how enclaves are keeping the network safe from fraudulent transactions or external attacks in CCF.

### Blockchain network

A blockchain (network) is a tamper-proof, highly available, decentralized ledger.

It’s a system that maintains cryptographically chained blocks of transactions (containing cryptocurrencies, code or assets) across nodes that are linked in a peer-to-peer network:

* It records **what happened** and the **order** it happened in.
* There isn’t any central server or organization, which rules the network. It’s a fully decentralized environment.

There are 2 types of blockchain: **public** vs. **private**.

On a public blockchain, members are usually unknown - you only know public addresses - and everybody can join the network. This is a **permissionless network**.

Conversely, on a private blockchain, everybody knows each other and when someone tries to join the blockchain, he usually needs to be accepted inside the network by other peers. This is a **permissioned network**.

To keep the network tamper-proof, a blockchain is usually based on/govern by algorithms. The goal of those algorithms aims at validating that a transaction is valid and spreading this transaction across the peer network. Indeed, every peer of the network runs its own copy of the transaction ledger, so such an algorithm is in charge of keeping every ledger consistent across nodes.

On a public blockchain, the algorithm requires that a peer, which wants to validate a transaction, needs to bet a lot of “something” (computation power, cryptocurrency, etc.). So, considering that, at least half of the members inside the network are not trying to attack or steal something on the network, this part of peers will overwhelm cheaters, which will lost their “bets”. This is why a public blockchain can be considered as secure: if you try to attack the blockchain, you likely will lose more than what you could win as potential rewards.

Those algorithms such as the well-known [Proof of work](https://en.wikipedia.org/wiki/Proof_of_work)[[114]](#footnote-115) (PoW) or [Proof of stake](https://en.wikipedia.org/wiki/Proof_of_stake)[[115]](#footnote-116) (PoS), provide reliability but there are some drawbacks, such as a low transaction throughput, a high computation cost, or an impossibility to scale up. To illustrate the point, Bitcoin currently uses approximately 69 TWh/year of electricity, about the same as Austria, to process between 2 and 3 transactions (Tx)/second. Determining if a transaction was successful is probabilistic rather than deterministic and takes at least an hour. In addition, transactions are not confidential.

This is (currently) not “enterprise ready”, resulting in the rise of private or consortium blockchains (in the meantime). In so far as you know every member inside the network, you can implement lighter algorithms (with less confidence) to verify transactions. Indeed, if someone tries to cheat, he/she will simply be kicked out of the blockchain. But this is not a perfect solution either, as this is still possible to cheat at least one time before being kicked, or, even if performances of the network are better, this isn’t as good as a database.

Furthermore, governance is a difficult problem, making changes and bug fixes problematic.

### Confidential Computing Framework (CCF)

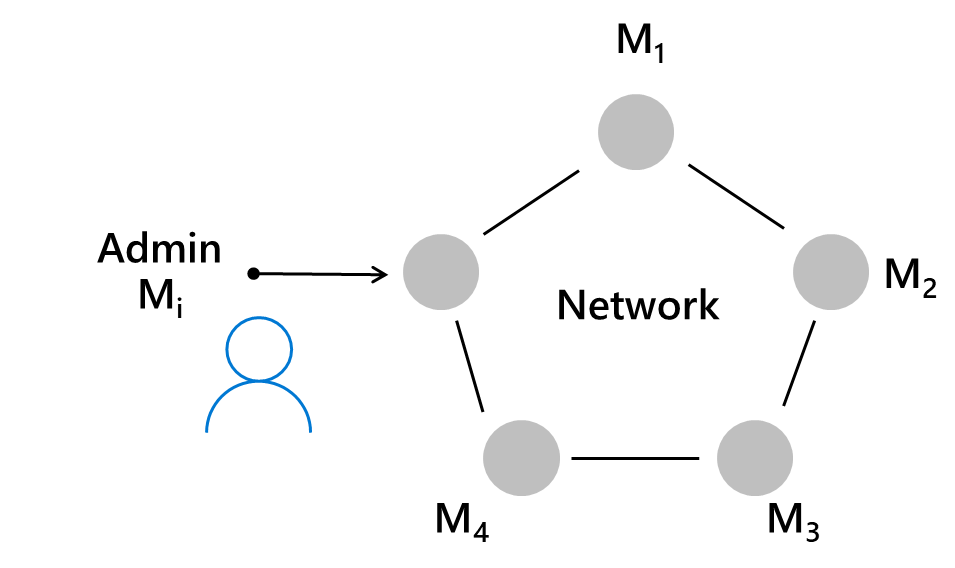
In this context, Microsoft tried another approach: what if the algorithm were just a way to establish a consensus between nodes, and the security part would be managed by something else, such as Trusted Execution Environments (TEEs) or enclaves.

As you saw before, an enclave is a secure area inside the processor, in a fully isolated environment. A user connected to the machine (even an administrator) can’t see what is running and processing inside this enclave. However, the user can get an attestation about the code running in that enclave. This means that if you put some (trusted) code inside an enclave, you can verify at any moment if your code was modified or not.

And this is basically how CCF works: by putting the code in charge of verifying transactions inside enclaves, you can be sure that a transaction signed by the code inside the enclave and the code itself are correct and not fraudulent.

Also, since you can verify transaction by yourselves, you can make them private if needed, as other nodes, which will append their ledger with this transaction, will only have to know if the transaction is signed, they don’t have to know about their content anymore.

With the use of Trusted Execution Environments (TEEs) or enclaves, CCF creates a trusted distributed blockchain network of physical nodes on which to run a distributed ledger, providing secure, reliable components for the protocol to use.



CCF is an infrastructure framework that leverages HW based (e.g. Intel SGX via ACC) Trusted Execution Environments (TEEs), [Paxos](https://en.wikipedia.org/wiki/Paxos_%28computer_science%29)[[116]](#footnote-117) and similar consensus mechanisms, a governance model, standard cryptography to enable arbitrary blockchain protocols to deliver high-throughput, secure computation with confidential transaction models for consortium blockchain networks.

**The Confidential Consortium Framework (CCF) doesn’t compete with blockchain’s technologies. It rather provides the foundation, the groundwork for turning these technologies into high-performance, confidential implementations.**

All of these simplify consensus and transaction processing for high throughput and confidentiality. Among other benefits, it provides for blockchain and multi-party application builders:

* **High transaction throughput**. Currently running 50,000+ Tx/sec. For context, Visa averages 2,000 Tx/sec, with 50,000 Tx/sec peaks.
* **Low latency**. Transaction latency in 10s of milliseconds.
* **Full confidentiality across for the entire workload**. Transactions are confidential.
* **Decentralized governance capabilities**. Strong, smart contract-based governance.
* **Attestation at scale**. Maintains integrity, resilience, and accountability properties of existing blockchains’ technologies.

If you want to learn more about CCF, feel free to explore the GitHub repository available [here](https://github.com/microsoft/CCF)[[117]](#footnote-118), or the website [at this address](https://microsoft.github.io/CCF/).[[118]](#footnote-119)

Now that you’re equipped with an understanding of the core capabilities of CCF, you are ready to build our own network, based on that framework.

## Step-by-step directions

This module covers the following three activities:

1. Deploying a 2-node network.
2. Installing CCF on your nodes.
3. Sending requests to CCF.

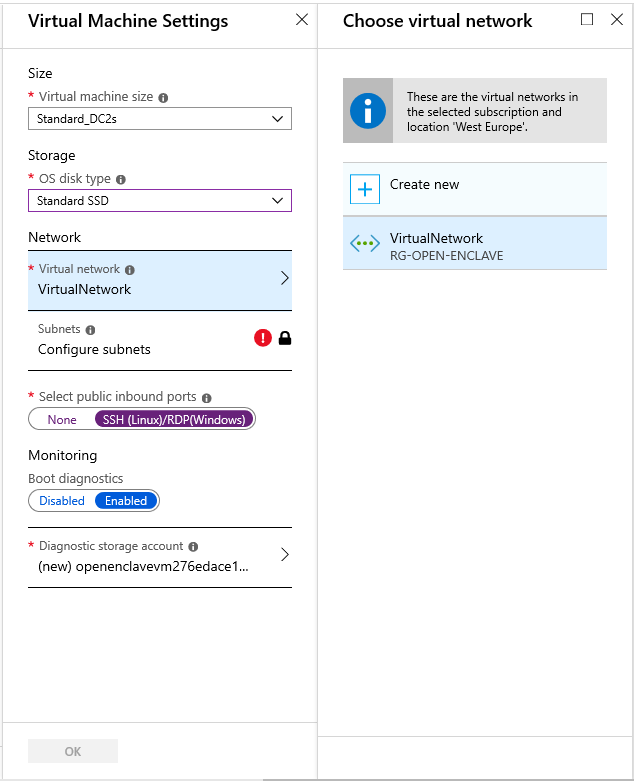
Each activity is described in order in the next sections.

### Deploying a 2-node network

For this sake of this guide, you will create a simple 2-node blockchain network. As CCF runs on top of Open Enclave, a node will unsurprisingly be a DC-series VM which is enclave-capable.

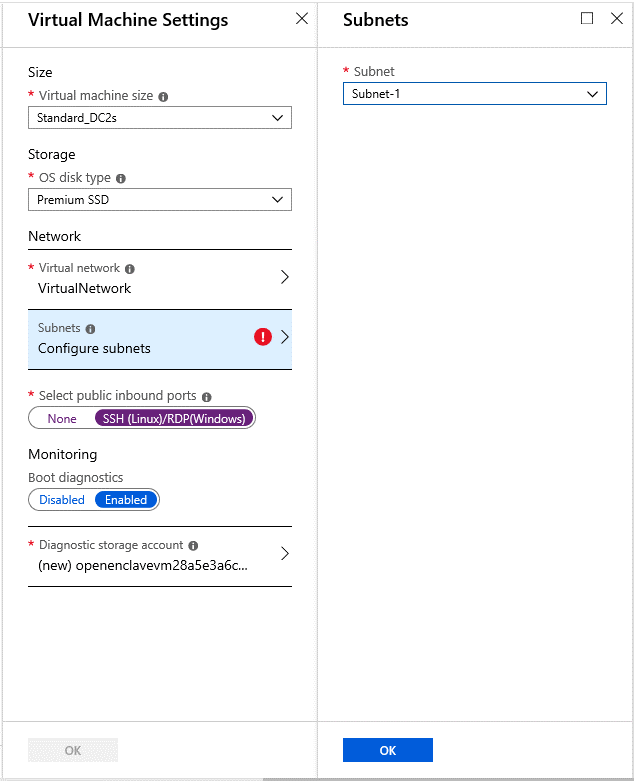
As you already covered how to deploy this type of machine, you should have one up-and-running DC-series VM at this stage. Please refer to section § **Deploying a DC-series VM on Azure in**  deploy another machine.

Although you will need to create a new resource group or point to an empty one, make sure that you select the same virtual network (VNet) for your second VMs when instantiating it, e.g. **RG-OPEN-ENCLAVE** in our illustration.



You can then place the VM on the same subnet:

1. Click on **Configure subnets**.
2. Select the subnet, i.e. **Subnet-1** in our illustration and click on **OK**.



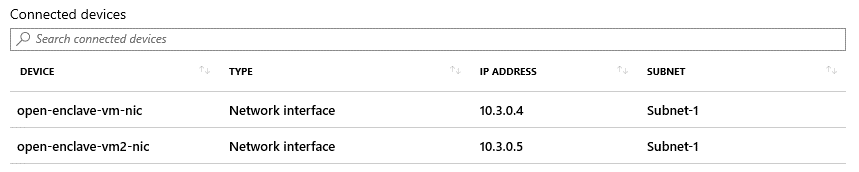
Proceed with the rest of the configuration.

Once completed, you are now ready to install CCF on your two nodes.

### Installing CCF on your nodes

In your Azure subscription, you should now have two DC-series VMs and a common virtual network that connect them together, e.g. in our illustration: **open-enclave-vm** and **open-enclave-vm2** on **Subnet-1**.

The following snapshot shows the two network interface cards connected onto the subnet.



Perform the following:

1. Open a remote terminal console on your first DC-series VM as per section § **Connecting to your DC-series VM**.
2. Repeat above step for your second DC-series VM, as you will need to be remotely connected to both of them.
3. Now that you are logged in on the two VMs, first make sure that you’re in your home directory on each of the VMs. Type the following command if not:

cd ~

1. It’s now time to download CCF. To do so, on each of the VMs, type the following command:

git clone https://github.com/microsoft/CCF

1. Before building CCF, you will need to install a few dependencies. Fortunately, a Shell script is available inside the CCF project to do that for you.
   1. Simply type the following command:

cd CCF/getting\_started/setup\_vm

* 1. and then, execute the script:

./setup.sh

Don’t forget to execute the commands on both of your VMs.

1. The next step consists in building CCF from its source files.
   1. Go back into the root directory of CCF by typing:

cd ../..

* 1. Create a *build* directory:

mkdir build

* 1. Then, go inside it:

cd build

* 1. You have two choices there: you can build CCF by enabling all log messages, or not. Enabling them can be useful to understand what is going on when the node is online, but it can be a bit overwhelming as a lot of log messages are displayed on the screen. Type one of the following commands:

cmake -GNinja ..

(without all logs)

-or-

cmake -DVERBOSE\_LOGGING=1 -GNinja ..

(with all logs)

* 1. In both of the cases, simply type the following command after that to build the solution:

ninja

**Note** To learn more about CCF installation (build arguments, etc.), see article [Getting Started](https://microsoft.github.io/CCF/getting_started.html)[[119]](#footnote-120) of the CCF documentation.

Congratulations! CCF is installed at this stage, you’re now able to run it. There are multiple steps to do so:

1. Launch two instances of CCF: one on each VM. They will become nodes of the network.
2. Each node will then have a quote file (used for remote attestation) and a cert file (associated to its public key). As such, the second node (the one which will join the network later) need to send its quote and cert file to the first one (over a TLS connection).
3. The first node (i.e. the one which will launch the network) need to generate a file called *nodes.json*, which describes the initial structure of the network when launched for the first time. It contains nodes IPs, Raft & TLS ports, and quote and cert files (which takes the form of a binary array).
4. Based to the previous file, the first node must generate the genesis transaction, which will be the anchor point of the starting blockchain. This will also generate a json script used to launch the network.
5. Finally, the first node will launch the blockchain, and the second node will get a signal to automatically join it.

**Note** Manually doing this step is quite complicated, but if you want to deep dive into CCF, see article [Starting up the network](https://microsoft.github.io/CCF/start_network.html)[[120]](#footnote-121) of the CCF documentation.

In this guide, you will use instead a Python script. This script aims at easing the deployment of your test infrastructure, by executing each required command automatically, providing that you give suitable VMs information and credentials.

To get it, simply go on the CCF sample repository (available [here](https://aka.ms/CCDevGuideSamples)[[121]](#footnote-122)), then click on clone and copy the .git URL which appeared. Then, in your VMs terminals, type the following command and press ENTER:

git clone <.git URL>

The python script is located inside the newly created directory *ccf-samples/CCF install files*. Provided that you’re inside the build directory, type the following command in your terminal console to copy the launching script from the repository to your current build directory:

cp “ccf-samples/CCF install files/network.py”

Then, run it by typing:

python3 network.py run

You will have to provide the following information:

* **IP address of the first node**. You can use the public VM IP or private VM IP (10.x.x.x) if you created a virtual network.
* **IP address of the second node**. (Same rules as the first node IP), username and password.
* **Raft and TLS ports**. They will be used by your nodes to communicate, e.g. 7465, 47585, etc.

**Important note** The script will not work if your CCF directory isn’t located in *~/CCF/*, but it should be at the right place if you correctly followed the guide.

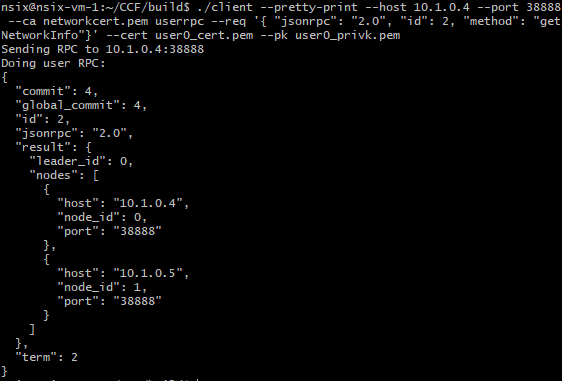
If your network is online, you should be able to run a JSON-RPC request to get online nodes’ list.

Perform the following steps:

1. Open a remote terminal console on your first DC-series VM as per section § **Connecting to your DC-series VM**.
2. Type the following command:

./client –pretty-print –host <your node IP> --port <your node TLS port> --ca networkcert.pem userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "getNetworkInfo"}' --cert user0\_cert.pem --pk user0\_privk.pem

Hereafter is an example of the expected result; you can see that 2 nodes are participating to the network.



Also, this is an example of a JSON-RPC request. This protocol is used with CCF to perform operations over the network. In the next section, you’re going to dive into all those operations to understand (some of) the CCF capabilities.

### Sending requests to CCF

#### Overview

Now that your network is online, you are able to communicate with it, to send transactions or requests over a JSON-RPC endpoint, which takes as an input a JSON message and return another JSON output.

If you want the complete list of allowed methods, feel free to read the JSON-RPC documentation [here](https://microsoft.github.io/CCF/rpc_api.html)[[122]](#footnote-123).

To send a request to a node, execute the following command from the terminal console:

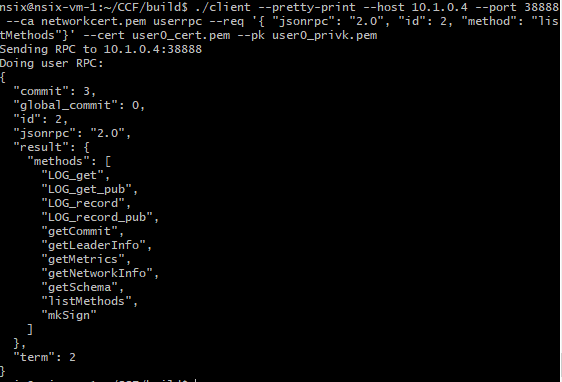
./client –pretty-print –host <your node IP> --port <your node TLS port> --ca networkcert.pem userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "<method name>", “params”: {<method parameters>}}' --cert user0\_cert.pem --pk user0\_privk.pem

Please note that params is optional, it depends of the executed method.

For some commands, like *User management* methods, you will directly use some programs built at the same time as CCF. We’re going to came across those executables later in this guide.

#### Common methods

To start with these commands, we’re going to execute the method listMethods to get in return the list of all allowed methods. As you launched our network over the CCF Logging sample application, you will see that four methods exists to handle this application:



We’re going to use them later, in a dedicated part. For the moment, let’s see interesting common methods. One of them is getLeaderInfo.

CCF is based on an algorithm called [Raft](https://en.wikipedia.org/wiki/Raft_(computer_science))[[123]](#footnote-124), which is in charge of electing a Leader between all nodes in the network. The Leader is the only node allowed to process transaction inside the network, as other nodes are called Followers and are waiting for orders from the Leader node.

This guarantee that there is a consensus between all nodes, as they only add transactions validated by the Leader.

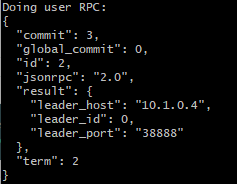
In your context, your know that our Leader is vm-1 since you’ve used it to start the network, but sometimes it could be useful to learn how to request information about the leader from any node of the network, to do some transactions later.

Let’s execute this command on one of our nodes, to see the result:

./client --pretty-print --host 10.1.0.4 --port 38888

--ca networkcert.pem userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "get

LeaderInfo"}' --cert user0\_cert.pem --pk user0\_privk.pem



As you can see, we’re getting back his IP and TLS port, which is enough to send him transaction.

#### User management methods

When someone wants to administrate members and users, member methods should be called. To illustrate this part, you’re going to add a user. First, your need to generate him/her his/her cert and his/her private key:

./genesisgenerator cert –name “newmember”

You should see those new files when doing ls inside your directory.

Adding a member in the network is not like another request. This type of request must be accepted by the majority of the members (aka reach the quorum).

To do so, a proposal must be submitted to the network, using memberclient executable. Type the following command:

./memberclient add\_member --ca=networkcert.pem --member\_cert=<New member cert> --cert=<Already accepted member cert> --privk=<Already accepted member private key> --host=<Node IP> --port=<TLS port>

If it worked, you should see the following message, indicating an id which is the proposal id:



Therefore, members can choose between accepting or denying access to the network to the new member.

This is also possible to add new nodes from this executable. To do that, call memberclient executable with the add\_node command, provide as a parameter a JSON object containing information about nodes to be added (exactly like the file *nodes.json*), but also your cert/privk and node IP/port like examples above:

./memberclient add\_node --nodes\_to\_add node.json --ca=networkcert.pem --cert=<Already accepted member cert> --privk=<Already accepted member private key> --host=<Node IP> --port=<TLS port>

The process is the same as members, you will get a proposal id and you can accept or reject it with the accept\_node and retire\_node commands.

#### Logging application methods

As you saw before, the Logging application of our network is constituted with 4 methods:

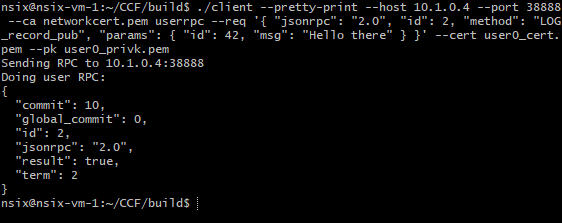
1. LOG\_record. Post a private log (a small text message) on the blockchain at a given index. As it is a private log, it will be encrypted on the distributed ledger and therefore only readable by nodes on the network.
2. LOG\_record\_pub. Perform the same action as LOG\_recordbut will not be encrypted after sending it. Anybody which can open the ledger file will see the log in plain text.
3. LOG\_get. Return a log from its index.
4. LOG\_get\_pub. Return a log from its id, only works if the log is public.

Let’s try to send a transaction which posts a new log:

./client –pretty-print –host <your node IP> --port <your node TLS port> --ca networkcert.pem userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "LOG\_record\_pub", “params”: { “id”: 42, “msg”: “Hello there”}}' --cert user0\_cert.pem --pk user0\_privk.pem

You can see that the method name is LOG\_record\_pub, and parameters are the newly created log index, and log content.

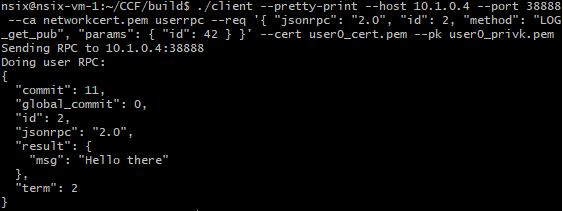
**Important note** The host pointed with that command must be the leader, as this is the only node which can process transactions. If you try to send this transaction to a follower node, you will get an error in return.



If everything went well, you should get this JSON in return. Let’s execute the get command to see if you can get back your log:

./client –pretty-print –host <your node IP> --port <your node TLS port> --ca networkcert.pem userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "LOG\_get\_pub", “params”: { “id”: 42 }}' --cert user0\_cert.pem --pk user0\_privk.pem

Following is the result:



You can see that in the result field, our message appeared. You can also have a look at the commit field: this id keeps track of transactions done with the network. Each time you send something to it, this it is incremented by 1.

**This concludes this starter guide for developers.**

# Appendix A. Prerequisites and additional configuration

## Setting up a core Azure IoT environment

To setup a core Azure IoT environment, perform the following four activities:

1. Create an Azure Container Registry.
2. Create an Azure IoT Hub.
3. Register an Azure IoT Edge device to your IoT hub.
4. Install and start the Azure IoT Edge runtime on your edge device.

Each activity is detailed in order in the next sections.

### Creating an Azure Container Registry

In section § **Building a TEE-based Linux module on an Edge ARM TrustZone device**, you will build a TEE-based module and create a container image from the related project files. Then you will push this image to a registry that stores and manages your images. Finally, you will deploy your image from your registry to run on your IoT Edge device.

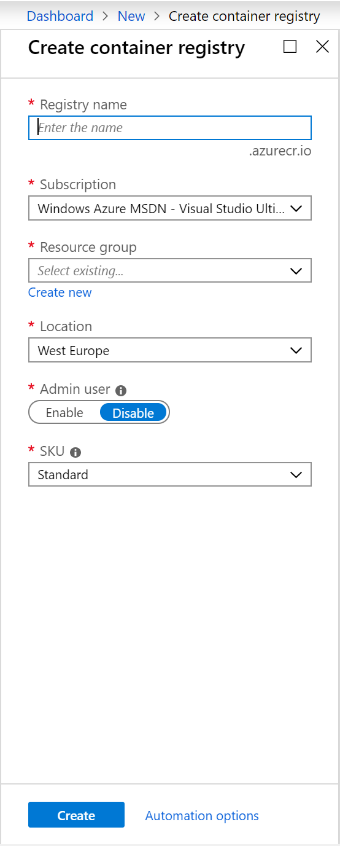
You can use any Docker-compatible registry to hold your container images. Two popular Docker registry services are [Azure Container Registry](https://azure.microsoft.com/en-us/services/container-registry/)[[124]](#footnote-125) and [Docker Hub](https://www.docker.com/products/docker-hub)[[125]](#footnote-126). You will use here Azure Container Registry (ACR).

Azure Container Registry allows you to manage a Docker private registry in Azure where you can store and manage your private Docker container images.

Perform the following steps:

**Note** For more information, see article [Quickstart: Create a private container registry using the Azure portal](https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-portal)[[126]](#footnote-127).

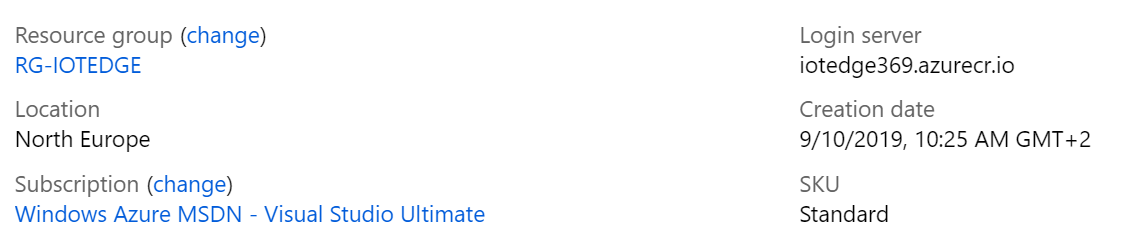
1. Open a browser session and go to the Azure portal at <https://portal.azure.com>.
2. Sign in with your Azure account.
3. Select **Create a resource** > **Containers** > **Container Registry**.



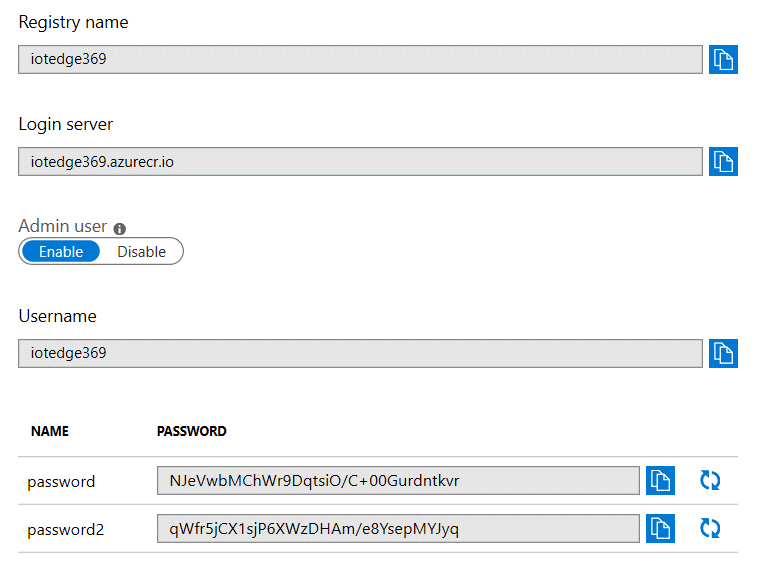
1. Specify the required settings.

|  |  |
| --- | --- |
| Setting | Description |
| Registry name | Provide a unique name. The registry name must be unique within Azure and contain 5-50 alphanumeric characters. For example, iotedge369 in our illustration. |
| Subscription | Select your subscription from the drop-down list if not already selected. |
| Resource group | You can create a new resource group or use an existing one.  To create a new one, click on **Create** new and fill in the name you want to use. For example, in our illustration, RG-IOTEDGE in our illustration.  To use instead an existing resource group, click on **Use existing** and select the resource group from the dropdown list. |
| Location | Choose a location close to you. |
| Admin user | Set to **Enable**. If enabled, you can use the registry name as username and admin user access key as password to docker login to your container registry, see section § **Providing your registry credentials to the IoT Edge agent**. |
| SKU | Select **Basic**. The Basic registry, which is a cost-optimized option for developers learning about Azure Container Registry. For details on available service tiers, see article [Azure Container Registry SKUs](https://docs.microsoft.com/en-us/azure/container-registry/container-registry-skus)[[127]](#footnote-128). |

1. Click on **Create** to deploy the container registry instance.



1. After your container registry is created, browse to it, and then select **Access keys** under **Settings**.



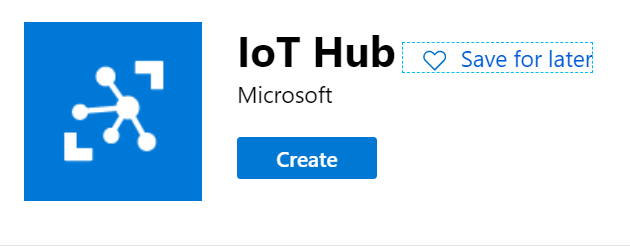
1. Make a note of the values for **Login server** (iotedge369.azurecr.io in our illustration), **Username** (iotedge369 in our illustration) and **Password** and save them somewhere convenient. You use these values throughout this module to provide access to your container registry. These values will be respectively referred as to registry-login-server, registry-username, and registry-password.

### Creating an Azure IoT Hub

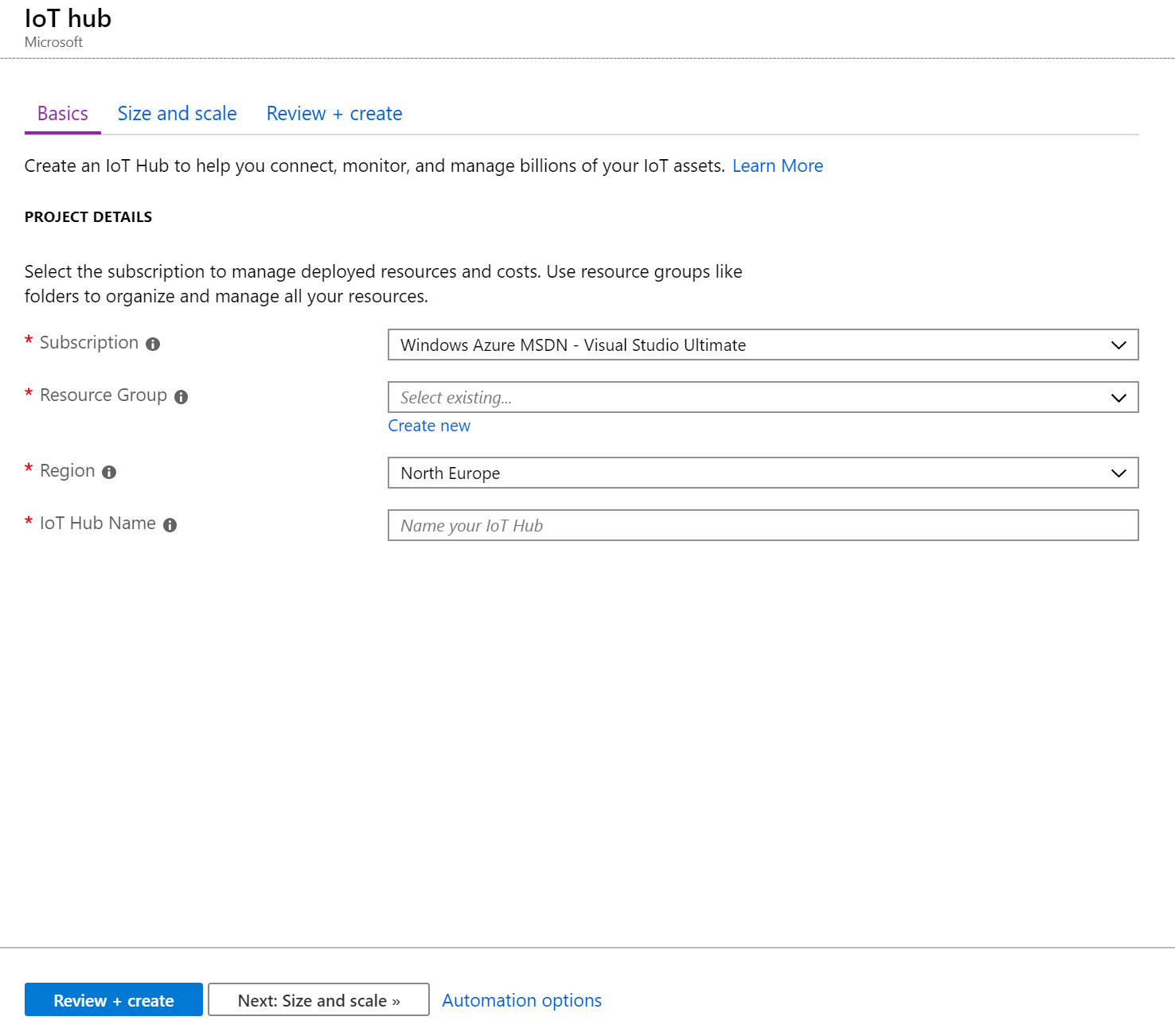
Perform the following steps:

**Note** For more information, see article [Create an IoT hub using the Azure portal](https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-create-through-portal)[[128]](#footnote-129).

1. Still from the Azure portal at <https://portal.azure.com>, in the left pane, select **Create a resource**. Search for “*IoT Hub*”in the **Search the Marketplace** search bar.



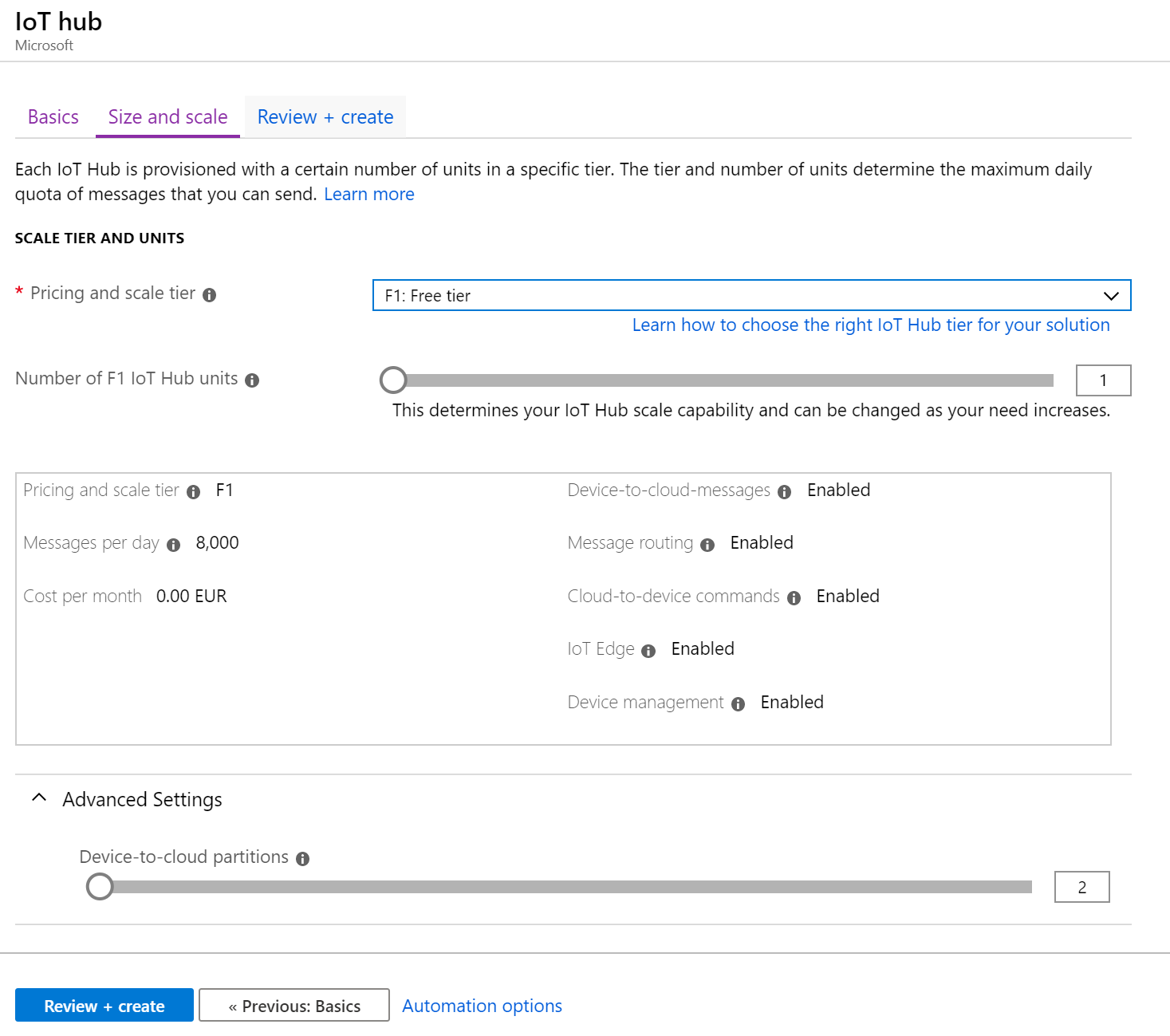
1. Select **Create.**



1. Specify the required settings.

|  |  |
| --- | --- |
| Setting | Description |
| Subscription | Select your subscription from the drop-down list if not already selected. |
| Resource Group | You can create a new resource group or use an existing one.  To create a new one, click on **Create** new and fill in the name you want to use.  To use instead an existing resource group, like the previous resource group, click on **Use existing** and select the resource group from the dropdown list.  We recommend that you use the same resource group for all the test resources that you create for your Azure IoT platform as part of this walkthrough. For example, RG-IOTEDGE in our illustration. |
| Region | This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list. |
| IoT Hub Name | Specify a name for your Azure IoT Hub, for example iothub369 in our illustration. This name must be globally unique. If the name you enter is available, a green check mark appears. |

1. Click on **Next: Size and scale** to continue creating your Azure IoT Hub.



1. Specify the required settings.

|  |  |
| --- | --- |
| Setting | Description |
| Pricing and scale tier | Specify the tier to use. You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free level of Azure IoT Hub works for this guide. If you've used Azure IoT Hub in the past and already have a free hub created, you can use that Azure IoT hub. Keep in mind that each subscription can only have one free IoT hub. Select **F1: Free tier** or **S1: Standard tier** for the pricing tier. |
| Number of F1|S1 IoT Hub units | Specify the number of units. The number of messages allowed per unit per day depends on your hub's pricing tier. You don’t need more than one unit for this guide. |
| Device-to-cloud partition | Specify the number of partitions. This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most Azure IoT Hubs only need four partitions, you can keep it by default. |

1. Select **Review + Create** to review your choices.
2. If validation has passed, click on **Create** to create your new Azure IoT hub. Creating the hub takes a few minutes.

This is it! Your Azure IoT Hub is ready, let’s now set up and connect our edge device to our hub.

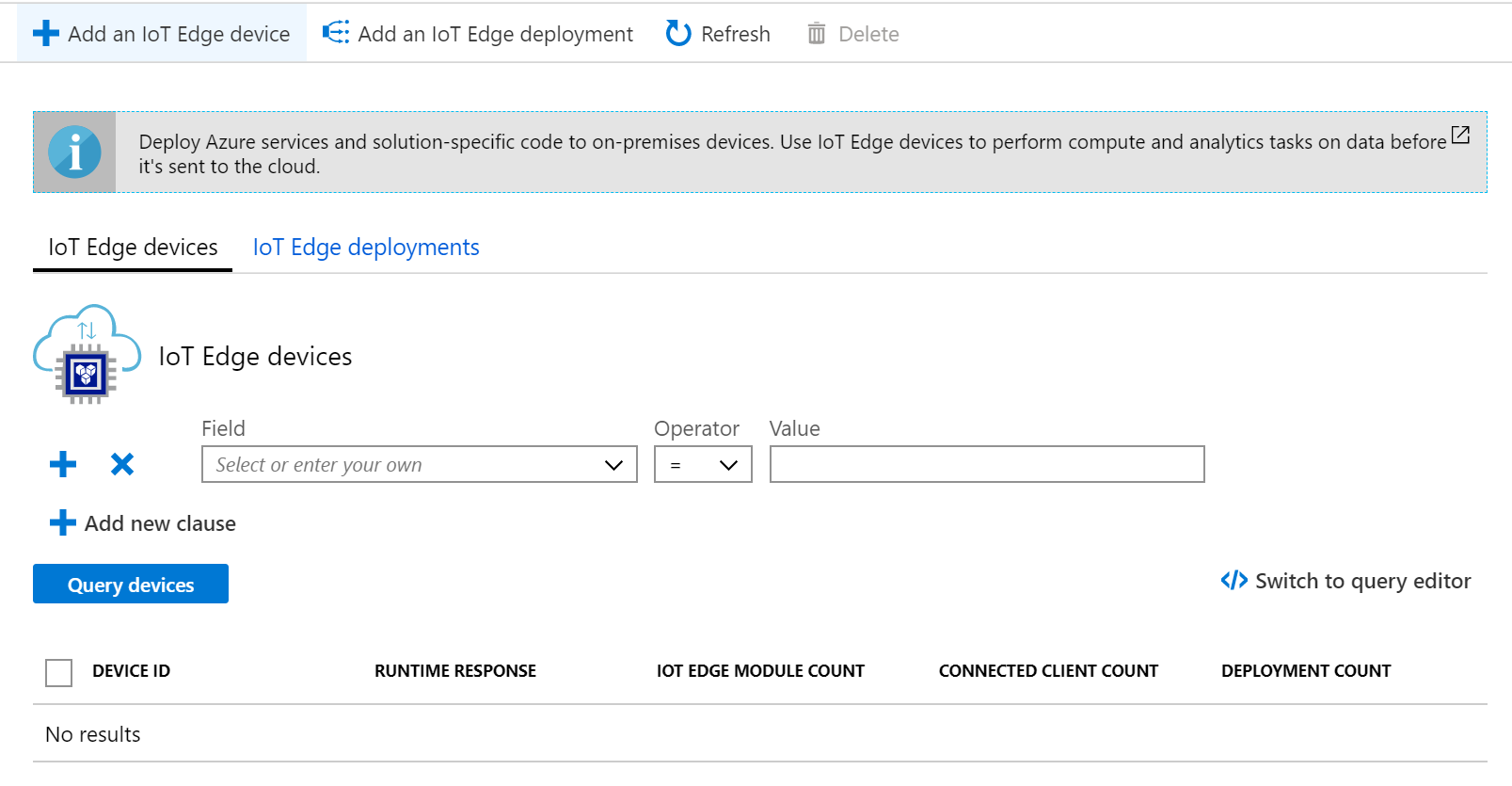
### Registering an Azure IoT Edge device to your Azure IoT hub

Let’s now create a device identity for your Azure IoT Edge device on your Azure IoT Hub so that it can communicate with your Azure IoT hub. The device identity lives in the cloud, and you use a unique device connection string to associate a physical device to a device identity.

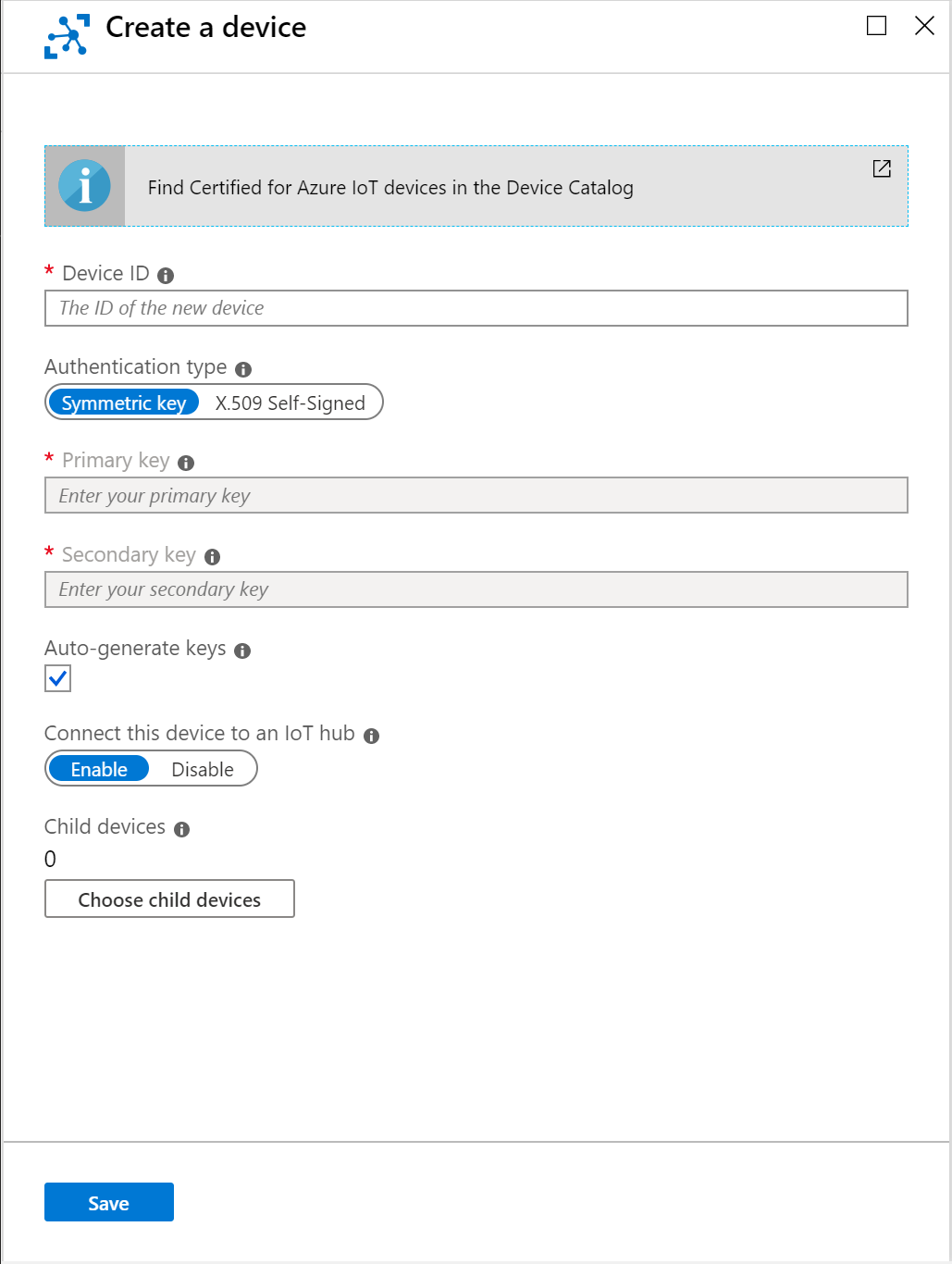
**Note** For more information, see article [Register a new Azure IoT Edge device from the Azure portal](https://docs.microsoft.com/en-us/azure/iot-edge/how-to-register-device-portal)[[129]](#footnote-130).

To manually create a device identity, perform the following steps:

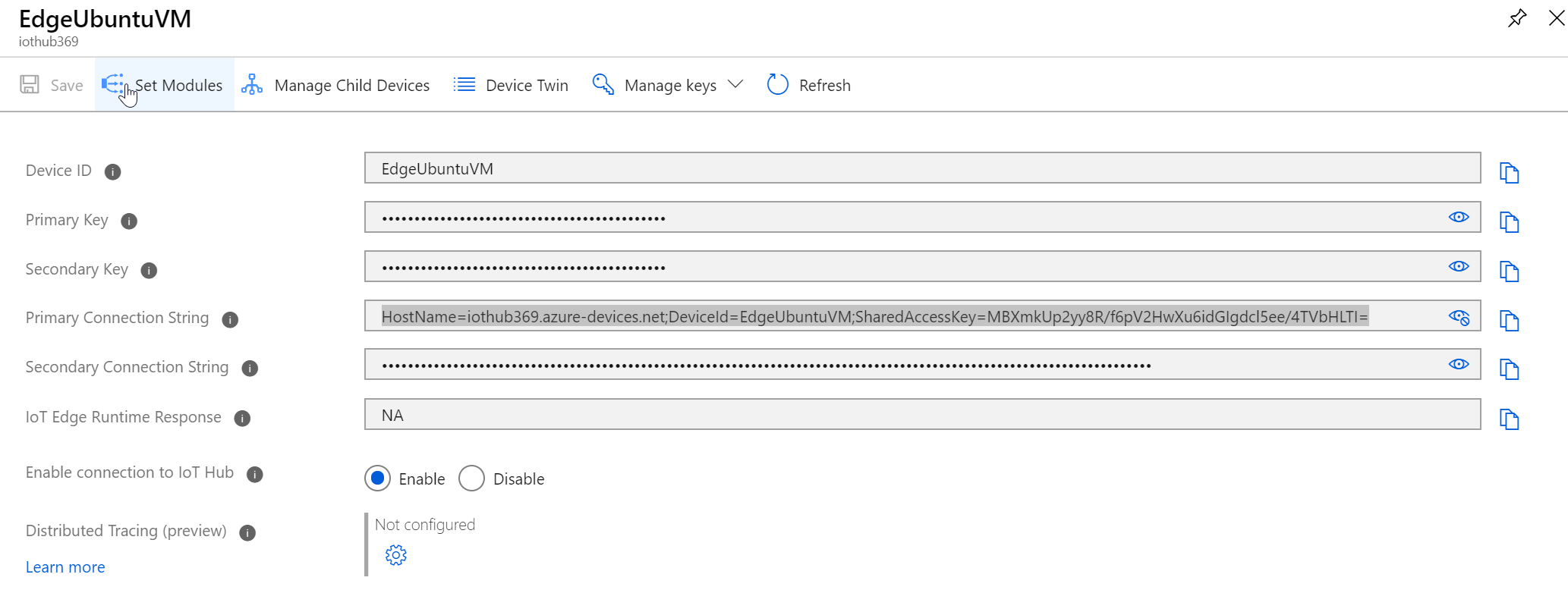
1. From Azure portal, search for “*IoT Hub*” in the search bar and select your **IoT Hub**.
2. Click on **IoT Edge** under **Automatic Device Management**.



1. Select **Add an IoT Edge Device.**



1. Specify the required settings. Enter a unique **Device ID**, for example, EdgeUbuntuVM in our illustration, and then click on **Save.**
2. Now click on the newly created device in the list of Azure IoT Edge devices.



You should look out for **Primary Connection String**. This value is the device connection string. You'll use this connection string to connect your actual device, and thus configure for that purpose the Azure IoT Edge runtime in the next section on your Azure IoT Edge device, save it.

It should look like this:

HostName=YourIoTHubName.azure-devices.net;DeviceId=SimulatedDevice;SharedAccessKey={YourSharedAccessKey}

Make a note of it. It will be further referred as to the string your\_iothub\_edge\_connection\_string.

Now we are all set! Let’s now configure your actual (virtual) Azure IoT Edge device.

### Installing and starting the Azure IoT Edge runtime on your device

The Azure IoT Edge runtime is deployed on all Azure IoT Edge devices. It has three components:

1. The [Azure IoT Edge security daemon](https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-security-manager)[[130]](#footnote-131) starts each time an Azure IoT Edge device boots and bootstraps the device by starting the IoT Edge agent.
2. The Azure IoT Edge agent facilitates deployment and monitoring of modules on the Azure IoT Edge device, including the IoT Edge hub.
3. The Azure IoT Edge hub manages communications between modules on the Azure IoT Edge device, and between the device and your Azure IoT Hub.

During the runtime configuration, you provide a device connection string.

A real edge device can typically be the [Scalys TrustBox Edge](https://scalys.com/trustbox-industrial/)[[131]](#footnote-132) device, which is an Industrial grade, tamper-resistant secured Azure IoT Edge device optimized for confidential computing using TEEs.

However, as mentioned earlier, for the sake of this guide, and to lessens the barrier for this walkthrough, not requiring some specific hardware, you will use instead an Ubuntu VM in Azure to act as your IoT Edge device, which allows you to quickly create a test machine with all prerequisites installed and then delete it when you're finished with this guide.

For that purpose, you should use the Microsoft-provided [Azure IoT Edge on Ubuntu](https://azuremarketplace.microsoft.com/marketplace/apps/microsoft_iot_edge.iot_edge_vm_ubuntu)[[132]](#footnote-133) VM, which preinstalls everything you need to run IoT Edge on a device. This virtual machine will install the latest Azure IoT Edge runtime and its dependencies on startup and makes it easy to connect to your IoT Hub.

**Note** For instructions on how to run the Azure IoT Edge runtime on your own (virtual) device, see article [Install the Azure IoT Edge runtime on Debian-based Linux systems](https://docs.microsoft.com/en-us/azure/iot-edge/how-to-install-iot-edge-linux)[[133]](#footnote-134).

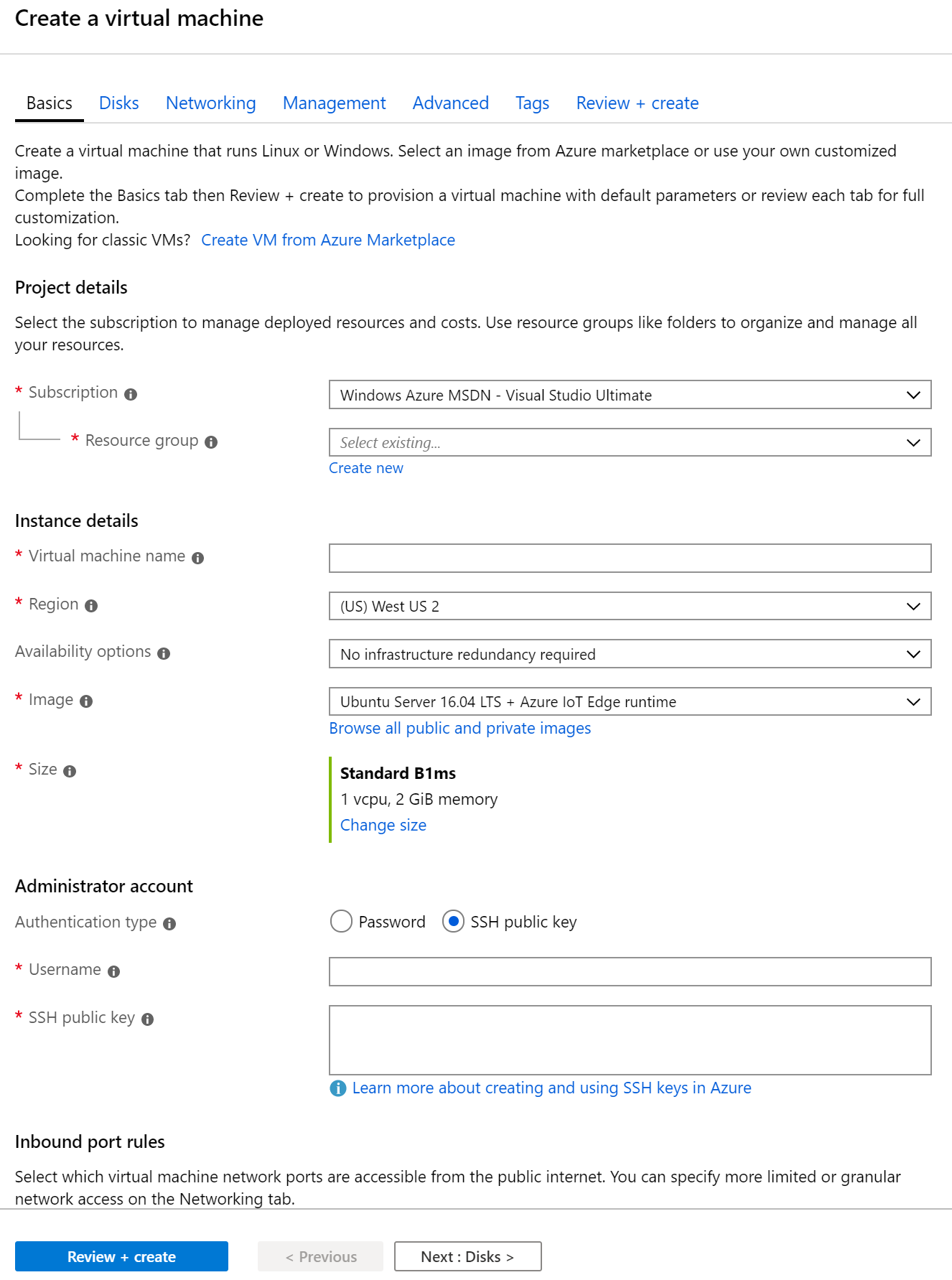
Perform the following steps:

**Note** For instructions on how to run the Azure IoT Edge runtime on your own (virtual) device, see article [Install the Azure IoT Edge runtime on Debian-based Linux systems](https://docs.microsoft.com/en-us/azure/iot-edge/how-to-install-iot-edge-linux)[[134]](#footnote-135).

1. Still from the Azure portal at <https://portal.azure.com>, in the left pane, select **Create a resource**. Search for “*Azure IoT Edge on Ubuntu*”in the **Search the Marketplace** search bar.



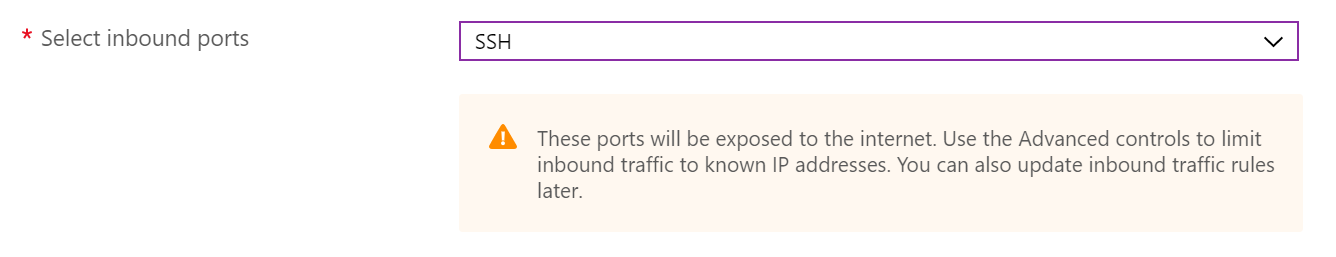
1. Click on **Create** and follow the wizard to deploy the VM.



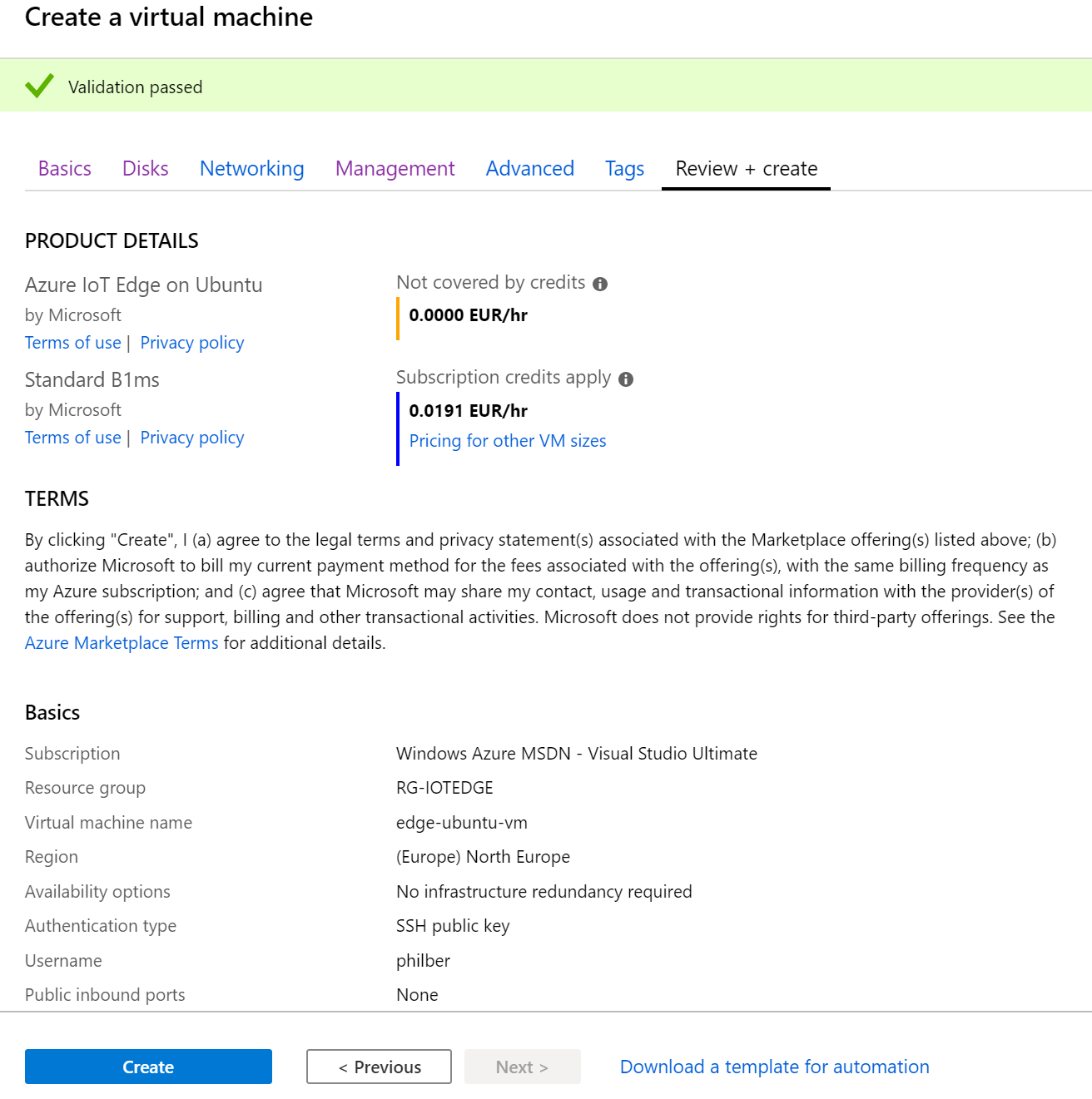
1. Specify the required settings.

|  |  |
| --- | --- |
| Setting | Description |
| Subscription | Select your subscription from the drop-down list if not already selected. |
| Resource group | You can create a new resource group or use an existing one.  To create a new one, click on **Create** new and fill in the name you want to use.  To use instead an existing resource group, like the previous resource group, click on **Use existing** and select the resource group from the dropdown list.  We recommend that you use the same resource group for all the test resources that you create for your Azure IoT platform as part of this walkthrough. For example, RG-IOTEDGE in our illustration. |
| Virtual machine name | Provide the VM a hostname (as a resource, which will be displayed in Azure). For example, edge-ubuntu-vm in our illustration. |
| Region | Select the Azure location where you want to deploy the VM. |
| Availability options | Leave **No infrastructure redundancy required** selected. THIS IS FOR DEMO PURPOSE ONLY |
| Image | Leave **Ubuntu Server 18.04 LTS + Azure IoT Edge runtime** selected. |
| Size | Leave **Standard B1ms** selected. |
| Username | Specify a username for the privileged user account of the VM. |
| Authentication type | Select **SSH public key** for stronger authentication to later remotely connect to your VM. |
| SSH public key | Specify a RSA public key in the single-line format beginning with “*ssh-rsa*” - you can use instead the multi-line PEM format -. You can use the same key as the one previously created for your DC-series VM |

1. Select **Networking**.
2. In **Select inbound ports**, ensure **SSH** is selected. THIS IS FOR DEMO PURPOSE ONLY.

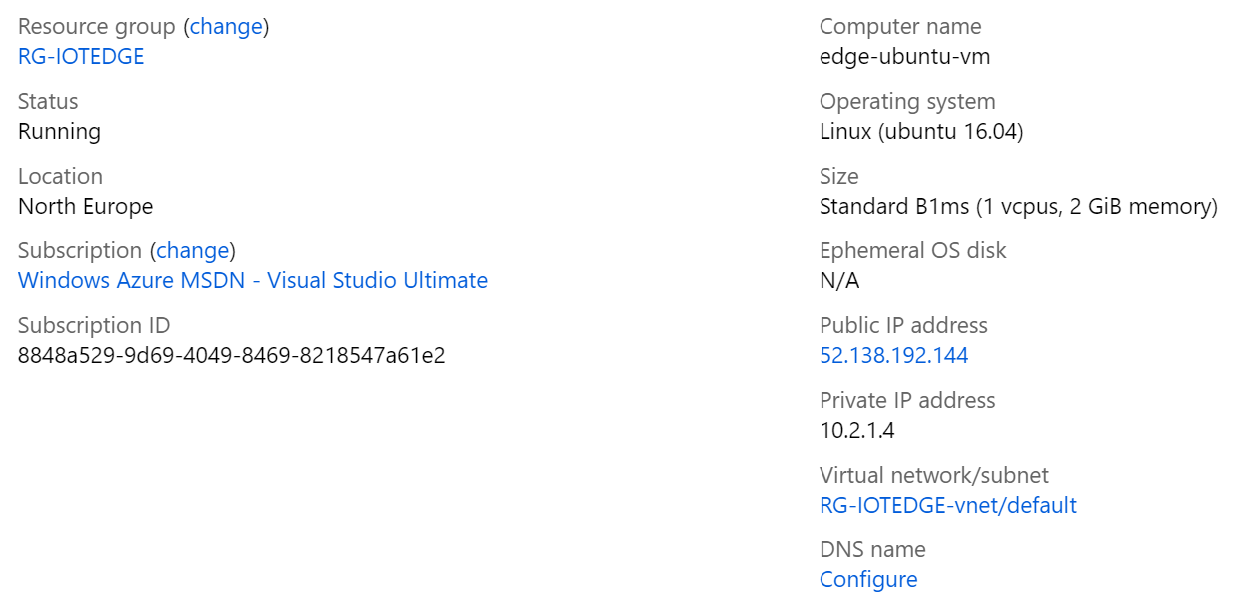


1. Click on **Review + create**.



1. Click on **Create**.

It may take a few minutes to create and start the new VM.



Once your Ubuntu VM is deployed, you need to provision it, i.e. connect it to your Azure IoT Hub. An Azure IoT Edge device can be provisioned manually using a device connection string provided by your Azure IoT Hub or automatically using the Device Provisioning Service (DPS), which is helpful when you have many devices to provision. For the sake of simplicity here, you will provision your device, i.e. VM, manually (since you only deal with a single device here).

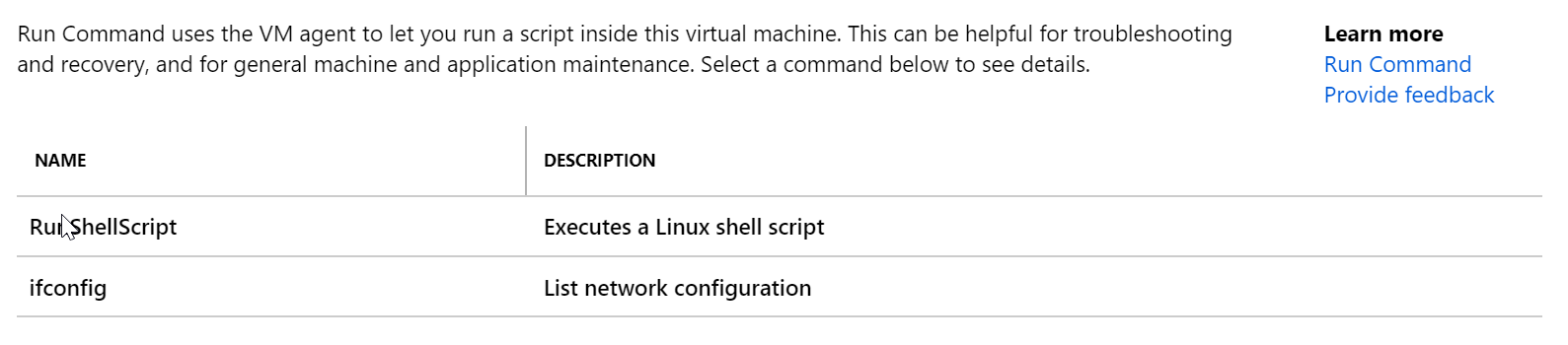
To do so, you now need to configure your Ubuntu VM with the device connection string your\_iothub\_edge\_connection\_string, you made a copy of in the previous section, which is of the form:

HostName=YourIoTHubName.azure-devices.net;DeviceId=SimulatedDevice;SharedAccessKey={YourSharedAccessKey}

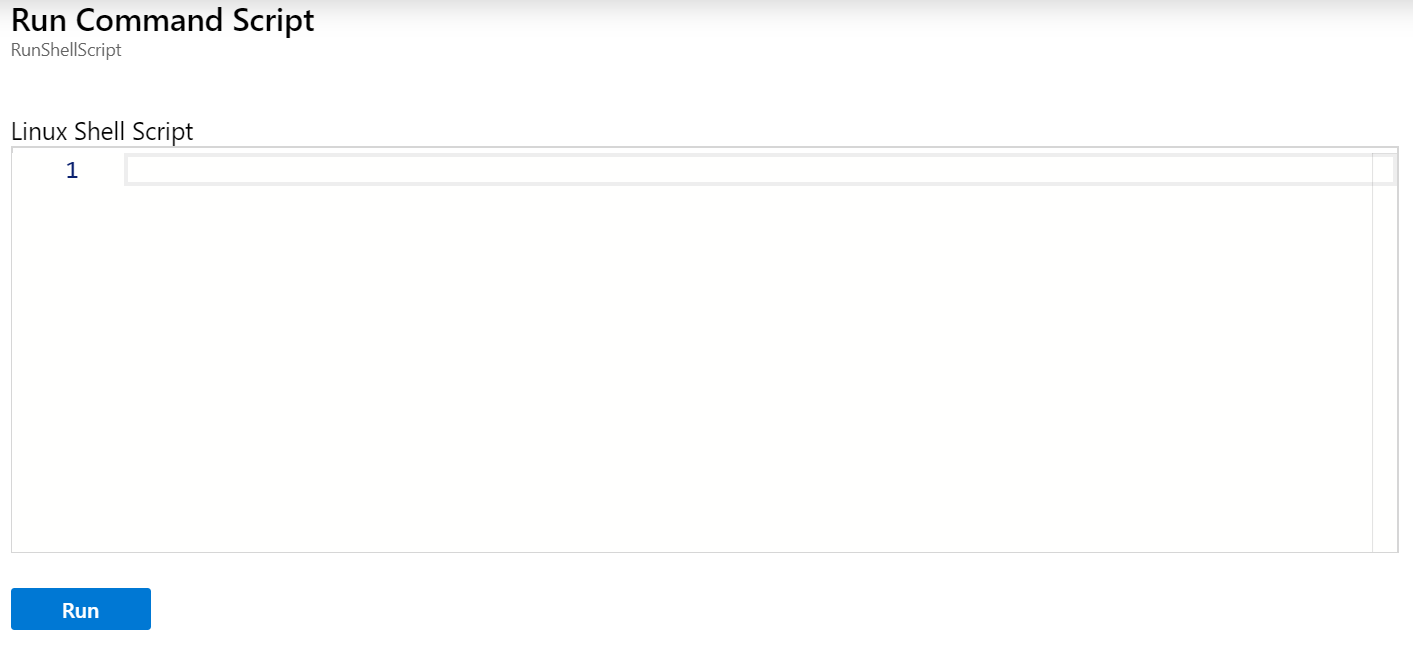
You can do this remotely without having to connect to the Ubuntu VM with the run command **RunShellScript** feature via the Azure portal to execute: /etc/iotedge/configedge.sh "<your\_iothub\_edge\_connection\_string>"

To do so, perform the following steps:

1. Select your newly created virtual machine resource from the Azure portal and click on **Run command** under **Operations**.



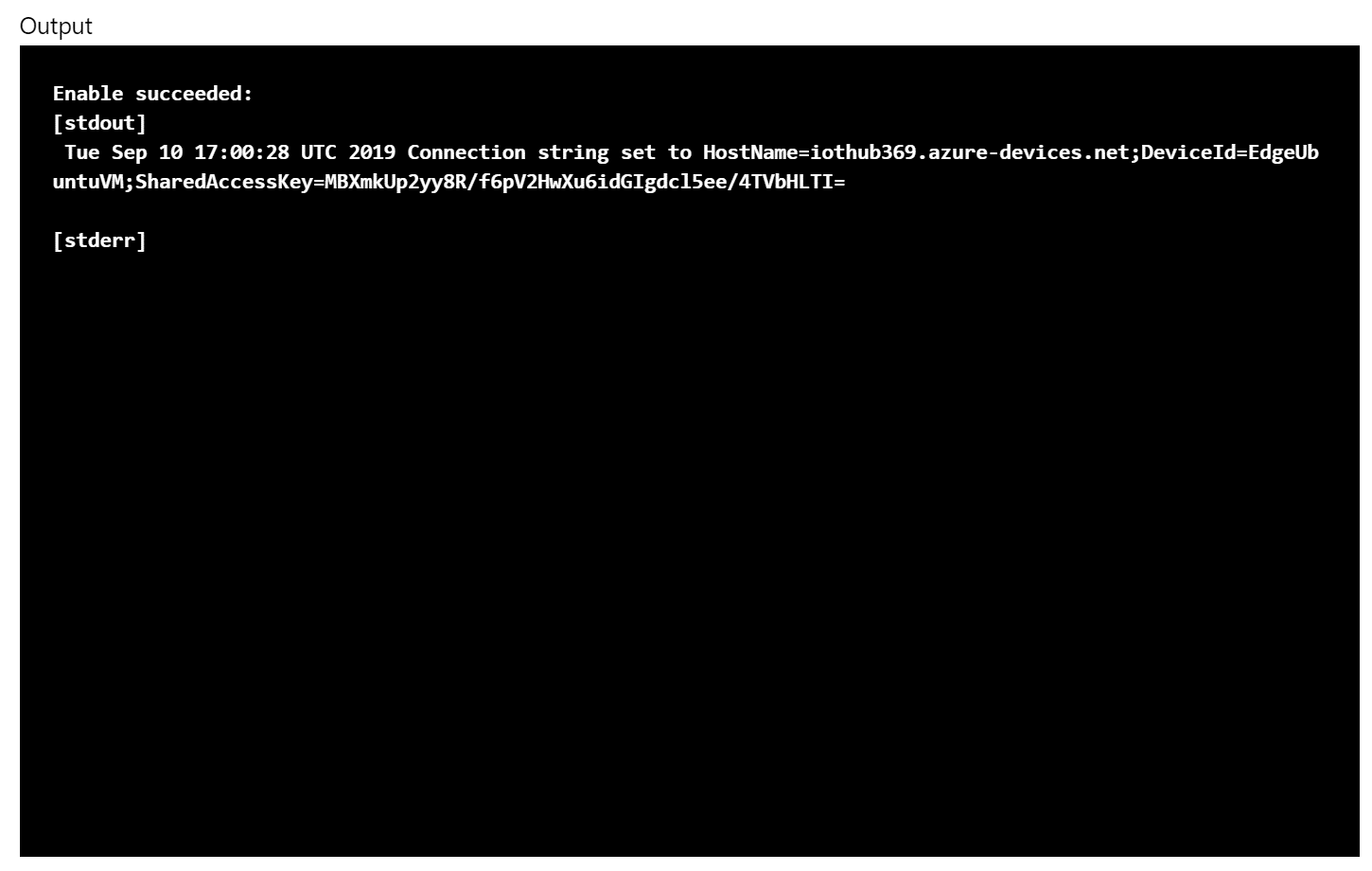
1. Select **RunShellScript**.



1. Execute the script below via the command window with your device connection string:

/etc/iotedge/configedge.sh "<your\_iothub\_edge\_connection\_string>"

1. Click on **Run**.
2. Wait a few moments, and the screen should then provide a success message indicating the connection string was set successfully.



# Appendix B. Support files for the guide

## network.py file

import os

import json

import datetime

import getpass

import paramiko

import sys

DEBUG = True

def log\_text(text):

if(DEBUG):

print(str(datetime.datetime.now()) + ' : ' + str(text))

def cert\_bytes(cert\_file\_name):

"""

Parses a pem certificate file into raw bytes and appends null character.

"""

with open(cert\_file\_name, "rb") as pem:

chars = []

for c in pem.read():

chars.append(ord(c))

# mbedtls demands null-terminated certs

chars.append(0)

return chars

def quote\_bytes(quote\_file\_name):

"""

Parses a binary quote file into raw bytes.

"""

with open(quote\_file\_name, "rb") as quote:

chars = []

for c in quote.read():

chars.append(ord(c))

return chars

def reset\_workspace():

os.system("sudo pkill cchost")

os.system("rm -rf tx0\* gov\* \*.pem quote\* nodes.json startNetwork.json joinNetwork.json ledger parsed\_\* sealed\_\*")

def reset\_remote\_workspace(c):

c.exec\_command("sudo pkill cchost")

c.exec\_command("cd ~/CCF/build && rm -rf tx0\* gov\* \*.pem quote\* nodes.json startNetwork.json joinNetwork.json ledger parsed\_\* sealed\_\*")

def generate\_members\_certs(nb\_members\_certs, nb\_users\_certs):

for x in range(0, nb\_members\_certs):

os.system("./genesisgenerator cert --name=./member" + str(x))

for x in range(0, nb\_users\_certs):

os.system("./genesisgenerator cert --name=./user" + str(x))

def generate\_members\_certs\_on\_remote(c, nb\_members\_certs, nb\_users\_certs):

for x in range(0, nb\_members\_certs):

c.exec\_command("cd ~/CCF/build && ./genesisgenerator cert --name=member" + str(x))

for x in range(0, nb\_users\_certs):

c.exec\_command("cd ~/CCF/build && ./genesisgenerator cert --name=user" + str(x))

def generate\_nodes\_json(info):

with open('./nodes.json', 'w') as outfile:

json.dump(

[

{

"host": info["node\_address\_1"],

"raftport": info["raft\_port"],

"pubhost": info["node\_address\_1"],

"tlsport": info["tls\_port"],

"cert": cert\_bytes("./0.pem"),

"quote": quote\_bytes("./quote0.bin"),

"status": 0,

},

{

"host": info["node\_address\_2"],

"raftport": info["raft\_port"],

"pubhost": info["node\_address\_2"],

"tlsport": info["tls\_port"],

"cert": cert\_bytes("./1.pem"),

"quote": quote\_bytes("./quote1.bin"),

"status": 0,

},

]

, outfile)

def start\_remote\_node(info, c):

reset\_remote\_workspace(c)

c.exec\_command("cd ./CCF/build && ./cchost "

"--enclave-file=./libloggingenc.so.signed "

"--raft-election-timeout-ms=100000 "

"--raft-host=" + info["node\_address\_2"] + " "

"--raft-port=" + info["raft\_port"] + " "

"--tls-host=" + info["node\_address\_2"] + " "

"--tls-pubhost=" + info["node\_address\_2"] + " "

"--tls-port=" + info["tls\_port"] + " "

"--ledger-file=./ledger "

"--node-cert-file=./1.pem "

"--enclave-type=debug "

"--log-level=info "

"--quote-file=./quote1.bin &")

def start\_node(info):

os.system("./cchost "

"--enclave-file=./libloggingenc.so.signed "

"--raft-election-timeout-ms=100000 "

"--raft-host=" + info["node\_address\_1"] + " "

"--raft-port=" + info["raft\_port"] + " "

"--tls-host=" + info["node\_address\_1"] + " "

"--tls-pubhost=" + info["node\_address\_1"] + " "

"--tls-port=" + info["tls\_port"] + " "

"--ledger-file=./ledger "

"--node-cert-file=./0.pem "

"--enclave-type=debug "

"--log-level=info "

"--quote-file=./quote0.bin & ")

def retrieve\_remote\_node\_certs(c):

try:

sftp = c.open\_sftp()

sftp.get("./CCF/build/quote1.bin", "./quote1.bin")

sftp.get("./CCF/build/1.pem","./1.pem")

return True

except:

print("Error: failed to retrieve remote node certs. Check node health, then retry.")

return False

def send\_network\_info\_to\_remote\_node(c):

try:

sftp = c.open\_sftp()

sftp.put("./networkcert.pem", "./CCF/build/networkcert.pem")

return True

except:

print("Error: failed to send network cert. Check node health, then retry.")

return False

def connect\_remote\_node(info):

try:

c = paramiko.SSHClient()

c.load\_system\_host\_keys()

c.set\_missing\_host\_key\_policy(paramiko.WarningPolicy)

c.connect(info["node\_address\_2"], port=22, username=info["node\_user\_2"], password=info["node\_pwd\_2"])

return c

except:

print("Error: unable to connect to remote node. Check node health and credentials, then retry.")

return False

def get\_light\_node\_info():

node\_address\_2 = str(raw\_input("Remote node IP address: "))

node\_user\_2 = str(raw\_input("Remote node IP username: "))

node\_pwd\_2 = getpass.getpass()

return {

"node\_address\_2": node\_address\_2,

"node\_user\_2": node\_user\_2,

"node\_pwd\_2": node\_pwd\_2,

}

def get\_node\_info():

node\_address\_1 = str(raw\_input("Local node IP address: "))

node\_address\_2 = str(raw\_input("Remote node IP address: "))

node\_user\_2 = str(raw\_input("Remote node IP username: "))

node\_pwd\_2 = getpass.getpass()

raft\_port = str(raw\_input("Raft port: "))

tls\_port = str(raw\_input("TLS port: "))

return {

"node\_address\_1": node\_address\_1,

"node\_address\_2": node\_address\_2,

"node\_user\_2": node\_user\_2,

"node\_pwd\_2": node\_pwd\_2,

"raft\_port": raft\_port,

"tls\_port": tls\_port,

}

def reset\_workspaces():

info = get\_light\_node\_info()

log\_text("Cleaning older files and resetting server on local node ...")

reset\_workspace()

log\_text("Done.")

log\_text("Connecting to remote node ...")

c = connect\_remote\_node(info)

log\_text("Done.")

log\_text("Cleaning older files and resetting server on remote node ...")

reset\_remote\_workspace(c)

log\_text("Done.")

c.close()

def run(args=None):

info = get\_node\_info()

log\_text("Cleaning older files and resetting server")

reset\_workspace()

log\_text("Starting local node ...")

start\_node(info)

log\_text("Connecting to remote node ...")

c = connect\_remote\_node(info)

if(c):

log\_text("Starting remote node ...")

start\_remote\_node(info, c)

log\_text("Waiting for nodes to start ...")

os.system("sleep 8")

log\_text("Retrieving remote node certs ...")

res = retrieve\_remote\_node\_certs(c)

if(res):

log\_text("Generating nodes.json file ...")

generate\_nodes\_json(info)

log\_text("Done.")

log\_text("Generating members and users certs on local node ...")

generate\_members\_certs(1,1)

log\_text("Done.")

log\_text("Generating members and users certs on remote node ...")

generate\_members\_certs\_on\_remote(c,1,1)

log\_text("Done.")

log\_text("Generating genesis transaction ...")

os.system("./genesisgenerator tx --gov-script=../src/runtime\_config/gov.lua")

log\_text("Starting blockchain ...")

os.system("./client --host=" + info["node\_address\_1"] + " --port=" + info["tls\_port"] + " --ca=./0.pem startnetwork --req=@startNetwork.json")

log\_text("Sending network cert to remote node ...")

send\_network\_info\_to\_remote\_node(c)

log\_text("Done.")

log\_text("Connecting remote node to blockchain ...")

c.exec\_command("cd ~/CCF/build && ./genesisgenerator joinrpc --network-cert=./networkcert.pem --host=" + info["node\_address\_1"] + " --port=" + info["tls\_port"])

#print("cd ~/CCF/build && ./genesisgenerator joinrpc --network-cert=./1.pem --host=" + info["node\_address\_1"] + " --port=" + info["tls\_port"])

c.exec\_command("cd ~/CCF/build && ./client --host=" + info["node\_address\_2"] + " --port=" + info["tls\_port"] + " --ca=./1.pem joinnetwork --req=joinNetwork.json")

#print("./client --host=" + info["node\_address\_2"] + " --port=" + info["tls\_port"] + " --ca=./1.pem joinnetwork --req=joinNetwork.json")

log\_text("Done.")

log\_text("Network online, setup complete.")

c.close()

else:

os.system("sudo pkill cchost")

os.system("cd ~/CCF/build && rm -rf tx0\* gov\* \*.pem quote\* nodes.json startNetwork.json joinNetwork.json 0 parsed\_\* sealed\_\*")

if \_\_name\_\_ == "\_\_main\_\_":

if sys.argv[1] == "run":

run()

elif sys.argv[1] == "clean":

reset\_workspaces()

else:

print("How to use the script : ")

print("-> python network.py run : launch the sample network")

print("-> python network.py clean : clean all files generated from previous tests, on VMs used")

|  |
| --- |
| Copyright © 2019 Microsoft France. All right reserved.  Microsoft France  39 Quai du Président Roosevelt  92130 Issy-Les-Moulineaux  The reproduction in part or in full of this document, and of the associated trademarks and logos, without the written permission of Microsoft France, is forbidden under French and international law applicable to intellectual property.  MICROSOFT EXCLUDES ANY EXPRESS, IMPLICIT OR LEGAL GUARANTEE RELATING TO THE INFORMATION IN THIS DOCUMENT.  Microsoft, Azure, Office 365, Microsoft 365, Dynamics 365 and other names of products and services are, or may be, registered trademarks and/or commercial brands in the United States and/or in other countries. |

1. Oblivious Multi-Party Machine Learning on Trusted Processors: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/paper.pdf [↑](#footnote-ref-2)
2. NIPS 2016 Workshop Private Multi‑Party Machine Learning: https://pmpml.github.io/PMPML16/ [↑](#footnote-ref-3)
3. Trusted execution environment: https://en.wikipedia.org/wiki/Trusted\_execution\_environment [↑](#footnote-ref-4)
4. Intel Software Extension Guard: https://software.intel.com/en-us/sgx [↑](#footnote-ref-5)
5. Layered Security for Your Next SoC: https://www.arm.com/products/silicon-ip-security [↑](#footnote-ref-6)
6. SGX and TrustZone: https://github.com/openenclave/openenclave/blob/feature.new\_platforms/new\_platforms/docs/sgx\_trustzone\_arch.md [↑](#footnote-ref-7)
7. Open Enclave SDK: https://Open Enclave.io/sdk/ [↑](#footnote-ref-8)
8. The future of computing: intelligent cloud and intelligent edge: https://azure.microsoft.com/en-us/overview/future-of-cloud [↑](#footnote-ref-9)
9. Intelligent video surveillance: an untapped source of value: https://blog.deepomatic.com/en/intelligent-video-surveillance-an-untapped-source-of-value/4098/ [↑](#footnote-ref-10)
10. Edge Technologies Give Oil, Gas Operators Extra Computing Power: https://www.rigzone.com/news/edge\_technologies\_give\_oil\_gas\_operators\_extra\_computing\_power-30-aug-2018-156789-article/ [↑](#footnote-ref-11)
11. Azure Confidential Computing: https://azure.microsoft.com/solutions/confidential-compute/ [↑](#footnote-ref-12)
12. Azure IoT Edge: https://azure.microsoft.com/en-us/services/iot-edge/ [↑](#footnote-ref-13)
13. Introducing Azure confidential computing: https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/ [↑](#footnote-ref-14)
14. Azure Confidential Computing updates with Mark Russinovich | Best of Microsoft Ignite 2018: https://www.youtube.com/watch?v=Qu6sP0XDMU8 [↑](#footnote-ref-15)
15. Simplifying confidential computing: Azure IoT Edge security with enclaves – Public preview: https://azure.microsoft.com/en-us/blog/simplifying-confidential-computing-azure-iot-edge-security-with-enclaves-public-preview/ [↑](#footnote-ref-16)
16. Deep Dive: Confidential Computing in IoT using Open Enclave SDK: https://channel9.msdn.com/Shows/Internet-of-Things-Show/Deep-Dive-Confidential-Computing-in-IoT-using-Open-Enclave-SDK [↑](#footnote-ref-17)
17. Confidential Computing Consortium: https://confidentialcomputing.io/ [↑](#footnote-ref-18)
18. New Cross-Industry Effort to Advance Computational Trust and Security for Next-Generation Cloud and Edge Computing: https://www.linuxfoundation.org/press-release/2019/08/new-cross-industry-effort-to-advance-computational-trust-and-security-for-next-generation-cloud-and-edge-computing/ [↑](#footnote-ref-19)
19. Microsoft Account: https://account.microsoft.com/account?lang=en-us [↑](#footnote-ref-20)
20. Create your Azure free account today: https://azure.microsoft.com/en-us/free/?WT.mc\_id=A261C142F [↑](#footnote-ref-21)
21. Visual Studio: https://visualstudio.microsoft.com/ [↑](#footnote-ref-22)
22. Visual Studio Code: https://code.visualstudio.com/ [↑](#footnote-ref-23)
23. PuTTY: https://www.chiark.greenend.org.uk/~sgtatham/putty/ [↑](#footnote-ref-24)
24. Git for Windows: https://git-for-windows.github.io/ [↑](#footnote-ref-25)
25. What’s new for the Command Line in Windows 10 version 1803: https://blogs.msdn.microsoft.com/commandline/2018/03/07/windows10v1803/ [↑](#footnote-ref-26)
26. General purpose virtual machine sizes: https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-general#dc-series [↑](#footnote-ref-27)
27. Azure Launches DC-Series Confidential Compute VM Preview: https://www.petri.com/azure-launches-dc-series-confidential-compute-vm-preview [↑](#footnote-ref-28)
28. Confidential Compute VM Deployment: https://azuremarketplace.microsoft.com/marketplace/apps/microsoft-azure-compute.confidentialcompute [↑](#footnote-ref-29)
29. Get Started with Microsoft Azure\* Confidential Computing: https://software.intel.com/en-us/articles/get-started-with-azure-confidential-computing [↑](#footnote-ref-30)
30. Public Key Authentication for SSH: https://www.ssh.com/ssh/public-key-authentication [↑](#footnote-ref-31)
31. PuTTYGen: https://www.puttygen.com/ [↑](#footnote-ref-32)
32. How to use SSH keys with Windows on Azure: https://docs.microsoft.com/en-us/azure/virtual-machines/linux/ssh-from-windows [↑](#footnote-ref-33)
33. Install the Open Enclave SDK (Ubuntu 18.04): https://github.com/openenclave/openenclave/blob/v0.6.x/docs/GettingStartedDocs/install\_oe\_sdk-Ubuntu\_18.04.md [↑](#footnote-ref-34)
34. Using the Open Enclave SDK: https://github.com/openenclave/openenclave/blob/master/docs/GettingStartedDocs/using\_oe\_sdk.md [↑](#footnote-ref-35)
35. HelloWorld sample: https://github.com/openenclave/openenclave/blob/master/samples/helloworld/README.md [↑](#footnote-ref-36)
36. File-Encryptor sample: https://github.com/openenclave/openenclave/blob/master/samples/file-encryptor/README.md [↑](#footnote-ref-37)
37. Data-Sealing sample: https://github.com/openenclave/openenclave/blob/master/samples/data-sealing/README.md [↑](#footnote-ref-38)
38. Remote Attestation sample: https://github.com/openenclave/openenclave/blob/master/samples/remote\_attestation/README.md [↑](#footnote-ref-39)
39. Local Attestation sample: https://github.com/openenclave/openenclave/blob/master/samples/local\_attestation/README.md [↑](#footnote-ref-40)
40. Attested TLS sample: https://github.com/openenclave/openenclave/blob/master/samples/attested\_tls/README.md [↑](#footnote-ref-41)
41. What is an Attested TLS channel: https://github.com/openenclave/openenclave/blob/master/samples/attested\_tls/AttestedTLSREADME.md#what-is-an-attested-tls-channel [↑](#footnote-ref-42)
42. Open Enclave SDK Samples: https://github.com/openenclave/openenclave/blob/master/samples/README.md [↑](#footnote-ref-43)
43. Deep Dive: Confidential Computing in IoT using Open Enclave SDK: https://channel9.msdn.com/Shows/Internet-of-Things-Show/Deep-Dive-Confidential-Computing-in-IoT-using-Open-Enclave-SDK [↑](#footnote-ref-44)
44. IoT Show: https://aka.ms/IoTShow [↑](#footnote-ref-45)
45. Determine the SGX support level: https://github.com/openenclave/openenclave/blob/master/docs/GettingStartedDocs/SGXSupportLevel.md [↑](#footnote-ref-46)
46. Install the Open Enclave SDK (Ubuntu 18.04): https://github.com/microsoft/openenclave/blob/master/docs/GettingStartedDocs/install\_oe\_sdk-Ubuntu\_18.04.md [↑](#footnote-ref-47)
47. Install the Open Enclave SDK (Ubuntu 16.04): https://github.com/microsoft/openenclave/blob/master/docs/GettingStartedDocs/install\_oe\_sdk-Ubuntu\_16.04.md [↑](#footnote-ref-48)
48. Windows Subsystem for Linux Documentation: https://aka.ms/wsldocs [↑](#footnote-ref-49)
49. Learn About Windows Console & Windows Subsystem For Linux (WSL): https://aka.ms/learnwsl [↑](#footnote-ref-50)
50. Visual Studio Code Remote Development Extension Pack: https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensionpack [↑](#footnote-ref-51)
51. Scalys TrustBox Edge: https://scalys.com/trustbox-industrial/ [↑](#footnote-ref-52)
52. Azure IoT Edge: https://azure.microsoft.com/en-us/services/iot-edge/ [↑](#footnote-ref-53)
53. QorIQ® Layerscape 1012A Low Power Communication Processor: https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/layerscape-communication-process/qoriq-layerscape-1012a-low-power-communication-processor:LS1012A [↑](#footnote-ref-54)
54. Azure IoT Edge on Ubuntu: https://azuremarketplace.microsoft.com/en-us/marketplace/apps/microsoft\_iot\_edge.iot\_edge\_vm\_ubuntu [↑](#footnote-ref-55)
55. Azure IoT : https://azure.microsoft.com/en-us/overview/iot/ [↑](#footnote-ref-56)
56. Getting started with the Open Enclave edger8r: https://github.com/openenclave/openenclave/blob/feature.new\_platforms/docs/GettingStartedDocs/Edger8rGettingStarted.md [↑](#footnote-ref-57)
57. Enclave Definition Language File Syntax - Intel® Developer Zone: https://software.intel.com/en-us/sgx-sdk-dev-reference-enclave-definition-language-file-syntax [↑](#footnote-ref-58)
58. Meltdown and Spectre: https://meltdownattack.com/ [↑](#footnote-ref-59)
59. Getting started with the Open Enclave edger8r: https://github.com/openenclave/openenclave/tree/master/docs/GettingStartedDocs/Edger8rGettingStarted.md [↑](#footnote-ref-60)
60. Azure IoT Hub: https://azure.microsoft.com/en-us/services/iot-hub/ [↑](#footnote-ref-61)
61. Securing the Intelligent Edge: https://azure.microsoft.com/en-us/blog/securing-the-intelligent-edge/ [↑](#footnote-ref-62)
62. Microsoft Azure IoT Architecture Reference guide: http://download.microsoft.com/download/A/4/D/A4DAD253-BC21-41D3-B9D9-87D2AE6F0719/Microsoft\_Azure\_IoT\_Reference\_Architecture.pdf [↑](#footnote-ref-63)
63. Remote Monitoring Solution with Azure IoT: https://github.com/Azure/azure-iot-pcs-remote-monitoring-dotnet/ [↑](#footnote-ref-64)
64. Azure IoT connected factory preconfigured solution: https://github.com/Azure/azure-iot-connected-factory [↑](#footnote-ref-65)
65. An introduction to creating a sample enclave using Intel Software Guard Extensions: https://software.intel.com/en-us/articles/intel-software-guard-extensions-developing-a-sample-enclave-application [↑](#footnote-ref-66)
66. Using Visual Studio to Develop Enclave Applications for Linux: https://github.com/openenclave/openenclave/blob/feature.new\_platforms/docs/GettingStartedDocs/VisualStudioLinux.md [↑](#footnote-ref-67)
67. Open Enclave Wizard – Preview extension: https://marketplace.visualstudio.com/items?itemName=MS-TCPS.OpenEnclaveSDK-VSIX [↑](#footnote-ref-68)
68. Visual Studio Community: https://visualstudio.microsoft.com/vs/community [↑](#footnote-ref-69)
69. Connect to your target Linux system in Visual Studio: https://docs.microsoft.com/en-us/cpp/linux/connect-to-your-remote-linux-computer?view=vs-2017 [↑](#footnote-ref-70)
70. Deploy, run, and debug your Linux project: https://docs.microsoft.com/en-us/cpp/linux/deploy-run-and-debug-your-linux-project?view=vs-2017 [↑](#footnote-ref-71)
71. Linux debugging walkthrough: https://docs.microsoft.com/en-us/cpp/linux/deploy-run-and-debug-your-linux-project?view=vs-2017 [↑](#footnote-ref-72)
72. Open Enclave extension for Visual Studio Code: https://github.com/openenclave/openenclave/blob/feature.new\_platforms/new\_platforms/vscode-extension/README.md [↑](#footnote-ref-73)
73. Remote Development Extension Pack: https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensionpack [↑](#footnote-ref-74)
74. Visual Studio Code Remote – SSH: https://aka.ms/vscode-remote/download/ssh [↑](#footnote-ref-75)
75. Visual Studio Code Remote – WSL: https://aka.ms/vscode-remote/download/wsl [↑](#footnote-ref-76)
76. Visual Studio Code Remote – Containers: https://aka.ms/vscode-remote/download/containers [↑](#footnote-ref-77)
77. Open Enclave extension for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-Open Enclave [↑](#footnote-ref-78)
78. Open Enclave extension for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave#Requirements [↑](#footnote-ref-79)
79. Visual Studio Code – Insiders: https://code.visualstudio.com/insiders [↑](#footnote-ref-80)
80. Git for Windows: https://git-for-windows.github.io/ [↑](#footnote-ref-81)
81. Installation of OpenSSH For Windows Server 2019 and Windows 10: https://docs.microsoft.com/en-us/windows-server/administration/openssh/openssh\_install\_firstuse [↑](#footnote-ref-82)
82. Remote Development using SSH: https://code.visualstudio.com/docs/remote/ssh [↑](#footnote-ref-83)
83. Developing in WSL: https://code.visualstudio.com/docs/remote/wsl [↑](#footnote-ref-84)
84. Developing inside a Container: https://code.visualstudio.com/docs/remote/containers [↑](#footnote-ref-85)
85. Remote Development Tips and Tricks: https://code.visualstudio.com/docs/remote/troubleshooting#\_installing-a-supported-ssh-client [↑](#footnote-ref-86)
86. ssh\_config: https://linux.die.net/man/5/ssh\_config [↑](#footnote-ref-87)
87. Open Enclave extension for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave#Requirements [↑](#footnote-ref-88)
88. Native Debug extension: https://marketplace.visualstudio.com/items?itemName=webfreak.debug [↑](#footnote-ref-89)
89. CMake download: https://cmake.org/download/ [↑](#footnote-ref-90)
90. QEMU: https://www.qemu.org/ [↑](#footnote-ref-91)
91. Remote Development using SSH: https://code.visualstudio.com/docs/remote/ssh [↑](#footnote-ref-92)
92. TrustZone technology for ARM v8-M Architecture: https://static.docs.arm.com/100690/0100/armv8\_m\_architecture\_trustzone\_technology\_100690\_0100\_00\_en.pdf [↑](#footnote-ref-93)
93. Understand Azure IoT Edge modules: https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-modules [↑](#footnote-ref-94)
94. Tutorial: Develop IoT Edge modules for Linux devices: https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-develop-for-linux [↑](#footnote-ref-95)
95. Tutorial: Develop a C IoT Edge module for Linux devices: https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-c-module [↑](#footnote-ref-96)
96. Azure IoT Tools extension pack: https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-tools [↑](#footnote-ref-97)
97. Azure IoT Hub Toolkit: https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-toolkit [↑](#footnote-ref-98)
98. Azure IoT Edge for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-edge [↑](#footnote-ref-99)
99. Use Visual Studio Code to develop and debug modules for Azure IoT Edge: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-vs-code-develop-module [↑](#footnote-ref-100)
100. Use Visual Studio Code to develop and debug modules for Azure IoT Edge: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-vs-code-develop-module [↑](#footnote-ref-101)
101. Open Enclave extension for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave#Requirements [↑](#footnote-ref-102)
102. Native Debug extension: https://marketplace.visualstudio.com/items?itemName=webfreak.debug [↑](#footnote-ref-103)
103. CMake download: https://cmake.org/download/ [↑](#footnote-ref-104)
104. Open Enclave extension for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-iot.msiot-vscode-openenclave#Requirements [↑](#footnote-ref-105)
105. Iotedgehubdev: https://pypi.org/project/iotedgehubdev/ [↑](#footnote-ref-106)
106. declare routes: https://docs.microsoft.com/en-us/azure/iot-edge/module-composition#declare-routes [↑](#footnote-ref-107)
107. iothub\_module\_client\_ll.h: https://docs.microsoft.com/en-us/azure/iot-hub/iot-c-sdk-ref/iothub-module-client-ll-h [↑](#footnote-ref-108)
108. IoTHubModuleClient\_LL\_SetInputMessageCallback(): https://docs.microsoft.com/en-us/azure/iot-hub/iot-c-sdk-ref/iothub-module-client-ll-h/iothubmoduleclient-ll-setinputmessagecallback [↑](#footnote-ref-109)
109. IoTHubModuleClient\_LL\_SendEventToOutputAsync(): https://docs.microsoft.com/en-us/azure/iot-hub/iot-c-sdk-ref/iothub-module-client-ll-h/iothubmoduleclient-ll-sendeventtooutputasync [↑](#footnote-ref-110)
110. Docker Login: https://docs.docker.com/engine/reference/commandline/login/#provide-a-password-using-stdin [↑](#footnote-ref-111)
111. Confidential Consortium Framework: https://aka.ms/ccf [↑](#footnote-ref-112)
112. Confidential Consortium Framework (CCF) Overview: https://channel9.msdn.com/Shows/Blocktalk/Confidential-Consortium-Framework-CCF-Overview [↑](#footnote-ref-113)
113. Confidential Consortium Framework (CCF) Part II: A deeper look and demos: https://channel9.msdn.com/Shows/Blocktalk/Confidential-Consortium-Framework-CCF-Part-II-A-deeper-look-and-demos [↑](#footnote-ref-114)
114. Proof of work: https://en.wikipedia.org/wiki/Proof\_of\_work [↑](#footnote-ref-115)
115. Proof of stake: https://en.wikipedia.org/wiki/Proof\_of\_stake [↑](#footnote-ref-116)
116. Paxos (computer science): https://en.wikipedia.org/wiki/Paxos\_%28computer\_science%29 [↑](#footnote-ref-117)
117. Confidential Consortium Framework: https://github.com/microsoft/CCF [↑](#footnote-ref-118)
118. Welcome to CCF’s documentation: https://microsoft.github.io/CCF/ [↑](#footnote-ref-119)
119. Getting Started: https://microsoft.github.io/CCF/getting\_started.html [↑](#footnote-ref-120)
120. Starting up the network: https://microsoft.github.io/CCF/start\_network.html [↑](#footnote-ref-121)
121. Guide samples on the GitHub repo: https://aka.ms/CCDevGuideSamples [↑](#footnote-ref-122)
122. RPC API: https://microsoft.github.io/CCF/rpc\_api.html [↑](#footnote-ref-123)
123. Raft (computer science): https://en.wikipedia.org/wiki/Raft\_(computer\_science) [↑](#footnote-ref-124)
124. Azure Container Registry: https://azure.microsoft.com/en-us/services/container-registry/ [↑](#footnote-ref-125)
125. Docker Hub: https://www.docker.com/products/docker-hub [↑](#footnote-ref-126)
126. Quickstart: Create a private container registry using the Azure portal: https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-portal [↑](#footnote-ref-127)
127. Azure Container Registry SKUs: https://docs.microsoft.com/en-us/azure/container-registry/container-registry-skus [↑](#footnote-ref-128)
128. Create an IoT hub using the Azure portal: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-create-through-portal [↑](#footnote-ref-129)
129. Register a new Azure IoT Edge device from the Azure portal: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-register-device-portal [↑](#footnote-ref-130)
130. Azure IoT Edge security manager: https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-security-manager [↑](#footnote-ref-131)
131. Scalys TrustBox Edge: https://scalys.com/trustbox-industrial/ [↑](#footnote-ref-132)
132. Azure IoT Edge on Ubuntu: https://azuremarketplace.microsoft.com/marketplace/apps/microsoft\_iot\_edge.iot\_edge\_vm\_ubuntu [↑](#footnote-ref-133)
133. Install the Azure IoT Edge runtime on Debian-based Linux systems: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-install-iot-edge-linux [↑](#footnote-ref-134)
134. Install the Azure IoT Edge runtime on Debian-based Linux systems: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-install-iot-edge-linux [↑](#footnote-ref-135)