



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

A starter guide for developers

Version 1.0, August 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

|   |           |
|---|-----------|
| <b>NOTICE .....</b>   | <b>2</b>  |
| <b>ABOUT THIS GUIDE .....</b>   | <b>3</b>  |
| GUIDE ELEMENTS.....   | 8         |
| GUIDE PREREQUISITES.....  | 8         |
| <b>MODULE 1: DC-SERIES VM SETUP WITH OPEN ENCLAVE.....</b>  | <b>10</b> |
| OVERVIEW.....   | 10        |
| STEP-BY-STEP DIRECTIONS .....   | 11        |
| Deploying a DC-series VM on Azure .....   | 11        |
| Connecting to your DC-series VM.....  | 16        |
| Using the Open Enclave SDK .....  | 19        |
| <b>MODULE 2: TEE-BASED APPLICATION DEVELOPMENT WITH THE OPEN ENCLAVE SDK.....</b>                       | <b>22</b> |
| OVERVIEW.....   | 22        |
| IMPORTANT CONCEPTS.....   | 23        |
| Terminology.....  | 23        |
| Enclave interface definition .....  | 24        |
| Data marshalling.....   | 25        |
| STEP-BY-STEP DIRECTIONS .....   | 26        |
| Building a TEE-based Linux application on Intel SGX .....   | 26        |
| Building a TEE-based Linux application on a simulated ARM TrustZone environment .....                   | 33        |
| <b>MODULE 3: CONFIDENTIAL CONSORTIUM FRAMEWORK (CCF) SETUP AND CONFIGURATION ON DC-SERIES VMS .....</b> | <b>45</b> |
| OVERVIEW.....   | 45        |
| IMPORTANT CONCEPTS.....   | 46        |
| Blockchain network.....   | 46        |
| Confidential Computing Framework (CCF) .....  | 47        |
| STEP-BY-STEP DIRECTIONS .....   | 48        |
| Deploying a 2-node network.....   | 48        |
| Installing CCF on your nodes.....   | 50        |
| Sending requests to CCF .....   | 53        |
| <b>APPENDIX .....</b>   | <b>58</b> |
| NETWORK.PY FILE.....  | 58        |

# 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 Open Enclave SDK in C and C++. The 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

Version 1.0

© 2019 Microsoft France. All rights reserved

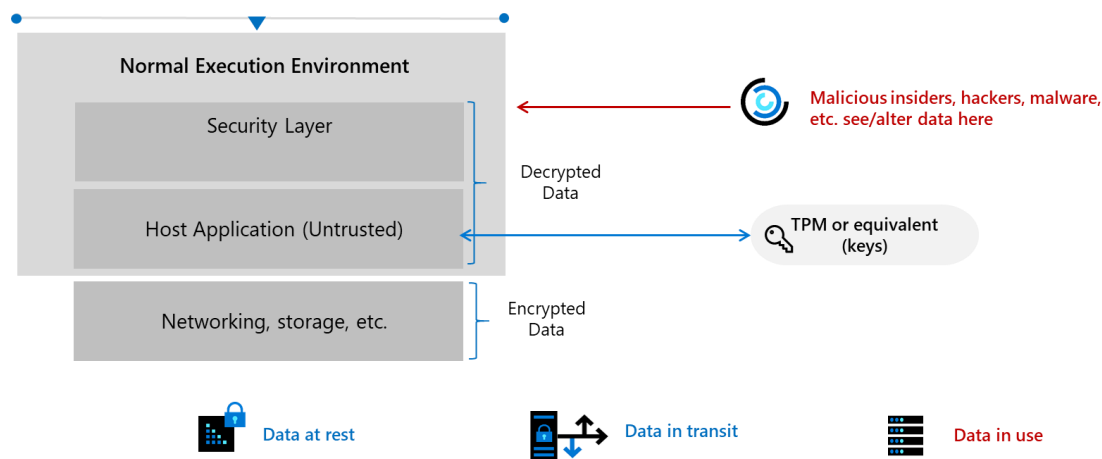
# About this guide

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 pharmaceutical 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)<sup>1</sup>, should allow to fusion sensitive data sources across different organizations while not revealing data to participants or the cloud platform.

---

<sup>1</sup> OBLIVIOUS MULTI-PARTY MACHINE LEARNING ON TRUSTED PROCESSORS: <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/paper.pdf>

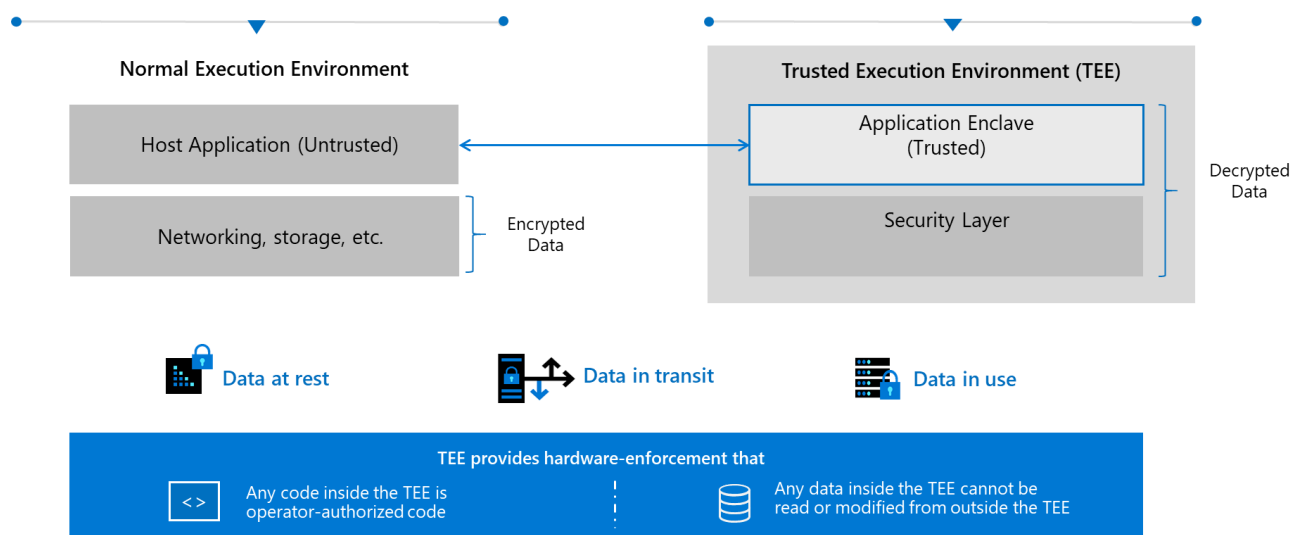
**Note** For more information on privacy-preserving multi-party machine learning, see the various presentations of the one-day workshop [NIPS 2016 Workshop Private Multi-Party Machine Learning](https://pmpml.github.io/PMPML16/)<sup>2</sup>.

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 (TEE).

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 Trust Execution Environments (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:

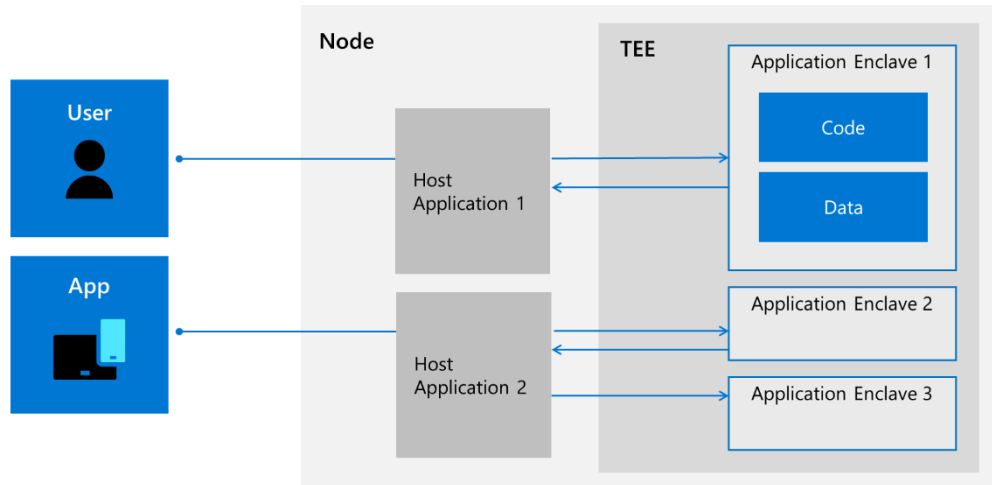
1. The host component runs unmodified on the untrusted operating system (OS), while the trusted component runs within the enclave. It's a normal user mode application that loads an enclave into its address space before starting to interact with an enclave.
2. The enclave is a secured container provided 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.

<sup>2</sup> NIPS 2016 Workshop Private Multi-Party Machine Learning: <https://pmpml.github.io/PMPML16/>

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 sum-up, code and data 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](#)<sup>3</sup> (SGX) - The Intel SGX instruction extension was introduced with 7<sup>th</sup> Generation Intel Core processor platforms and Intel Xeon processor E3 v5 for data center servers back in 2015 -, [ARM TrustZone](#)<sup>4</sup> (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](#)<sup>5</sup>.

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 [Open Enclave SDK](#)<sup>6</sup> available in open source on GitHub. The Open Enclave SDK aims at creating a single unified enclaving abstraction for developer to build TEE-based applications.

The Open Enclave SDK provides an open source consistent API surface across enclave technologies and platforms from the cloud to the edge.

<sup>3</sup> Intel Software Extension Guard: <https://software.intel.com/en-us/sgx>

<sup>4</sup> Layered Security for Your Next SoC: <https://www.arm.com/products/silicon-ip-security>

<sup>5</sup> SGX AND TRUSTZONE: [https://github.com/openenclave/openenclave/blob/feature.new\\_platforms/new\\_platforms/docs/sgx\\_trustzone\\_arch.md](https://github.com/openenclave/openenclave/blob/feature.new_platforms/new_platforms/docs/sgx_trustzone_arch.md)

<sup>6</sup> Open Enclave SDK: <https://OpenEnclave.io/sdk/>





**Note** For more information on Azure Confidential Computing, see blog post [INTRODUCING AZURE CONFIDENTIAL COMPUTING](#)<sup>11</sup> and the webcast [AZURE CONFIDENTIAL COMPUTING UPDATES WITH MARK RUSSINOVICH | BEST OF MICROSOFT IGNITE 2018](#)<sup>12</sup>.

**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](#)<sup>13</sup> and the webcast [DEEP DIVE: CONFIDENTIAL COMPUTING IN IOT USING OPEN ENCLAVE SDK](#)<sup>14</sup>.

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.

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.

---

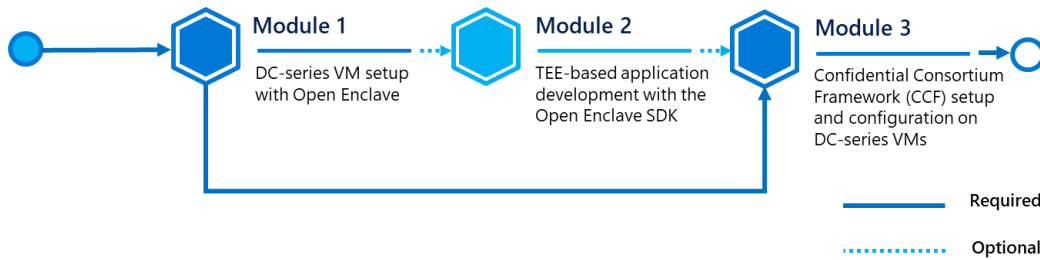
<sup>11</sup> INTRODUCING AZURE CONFIDENTIAL COMPUTING: <https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/>

<sup>12</sup> AZURE CONFIDENTIAL COMPUTING UPDATES WITH MARK RUSSINOVICH | BEST OF MICROSOFT IGNITE 2018: <https://www.youtube.com/watch?v=Qu6sP0XDMU8>

<sup>13</sup> 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/>

<sup>14</sup> 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>

**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 VMs well-suited for trusted applications development,
- Setup a full-pledged development environment with Visual Studio and Visual Studio Code,
- Create new trusted applications 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)<sup>15</sup>.
- An Azure subscription. If you don't have an Azure subscription, create a [free account](https://azure.microsoft.com/en-us/free/?WT.mc_id=A261C142F)<sup>16</sup> before you begin.
- A windows 10 local machine.

<sup>15</sup> Microsoft Account: <https://account.microsoft.com/account?lang=en-us>

<sup>16</sup> Create your Azure free account today: [https://azure.microsoft.com/en-us/free/?WT.mc\\_id=A261C142F](https://azure.microsoft.com/en-us/free/?WT.mc_id=A261C142F)

- A code editor of your choice, such as [Visual Studio](#)<sup>17</sup> or [Visual Studio Code](#)<sup>18</sup>, 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](#)<sup>19</sup>, [Git for Windows](#)<sup>20</sup> (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](#)<sup>21</sup>.

---

<sup>17</sup> Visual Studio: <https://visualstudio.microsoft.com/>

<sup>18</sup> Visual Studio Code: <https://code.visualstudio.com/>

<sup>19</sup> PuTTY: <https://www.chiark.greenend.org.uk/~sgtatham/putty/>

<sup>20</sup> Git for Windows: <https://git-for-windows.github.io/>

<sup>21</sup> WHAT'S NEW FOR THE COMMAND LINE IN WINDOWS 10 VERSION 1803:  
<https://blogs.msdn.microsoft.com/commandline/2018/03/07/windows10v1803/>

# 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](#)<sup>22 23</sup> 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:

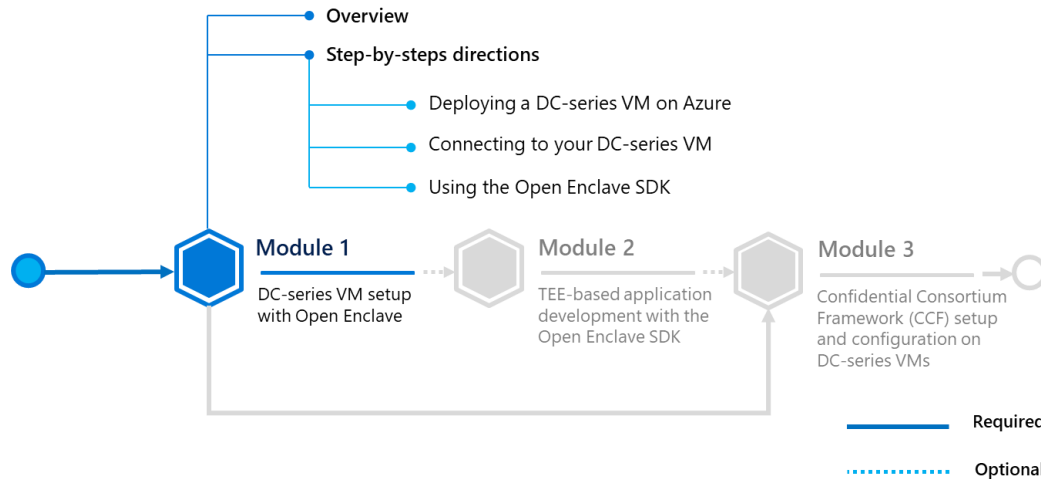
1. Windows Server 2016 Datacenter,
2. Ubuntu Server 16.04 LTS,
3. Ubuntu Server 18.04 TLS.

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

---

<sup>22</sup> GENERAL PURPOSE VIRTUAL MACHINE SIZES: <https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-general#dc-series>

<sup>23</sup> AZURE LAUNCHES DC-SERIES CONFIDENTIAL COMPUTE VM PREVIEW: <https://www.petri.com/azure-launches-dc-series-confidential-compute-vm-preview>



## 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 be found in the directory in the [Azure Marketplace](#)<sup>24</sup> or by searching “*Confidential Compute*” in the search bar in Azure.

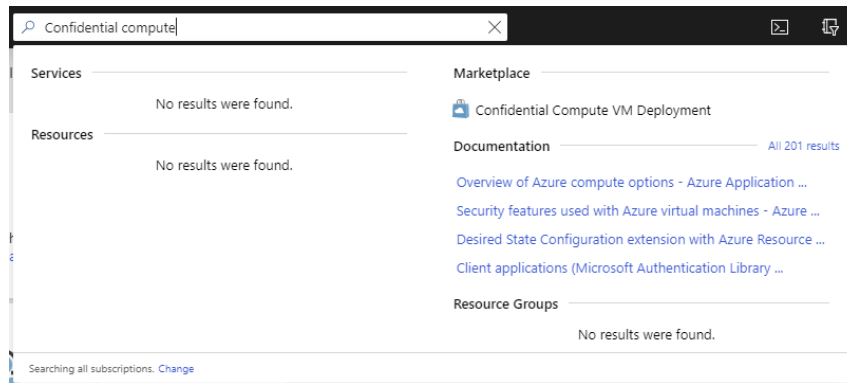
To deploy a DC-series VM in your Azure subscription, perform the following steps:

**Note** For more information, see article [GET STARTED WITH MICROSOFT AZURE\\* CONFIDENTIAL COMPUTING](#)<sup>25</sup>.

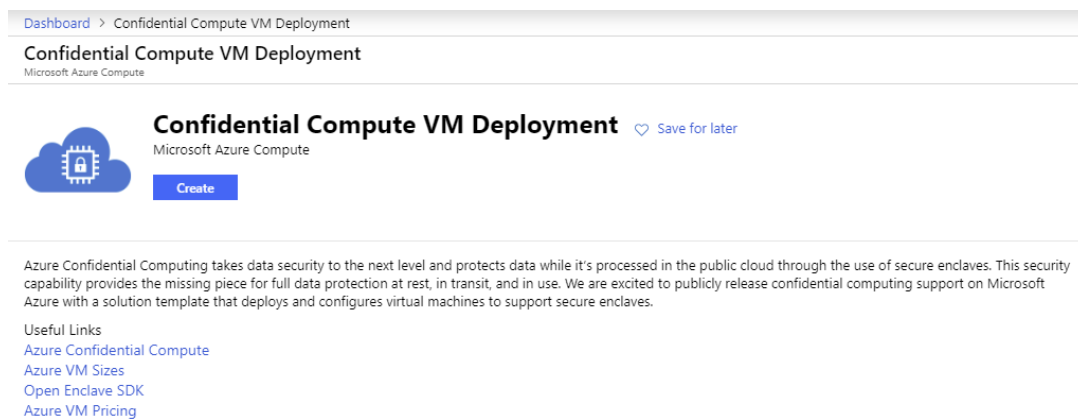
1. Open a browser session and go to the Azure portal at <https://portal.azure.com>.
2. Sign in with your Azure account.
3. First search for “*Confidential Compute*” in the search bar in the Azure portal.

<sup>24</sup> Confidential Compute VM Deployment: <https://azuremarketplace.microsoft.com/marketplace/apps/microsoft-azure-compute.confidentialcompute>

<sup>25</sup> GET STARTED WITH MICROSOFT AZURE\* CONFIDENTIAL COMPUTING: <https://software.intel.com/en-us/articles/get-started-with-azure-confidential-computing>



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



5. Click **Create**. You will have to fill two subsequent pages to complete the deployment. The first one is the **Basics** page.

Dashboard > Confidential Compute VM Deployment > Create Confidential Compute VM Deployment >

### Create Confidential Compute ...

- 1 Basics**  
Configure basic settings
- 2 Virtual Machine Settings  
Configure the virtual machine's re...
- 3 Summary  
Confidential Compute VM Deploy...
- 4 Buy

#### Basics

ACC VMs are only available in East US and West Europe regions currently.

\* Image  
Ubuntu Server 18.04 LTS

\* Name  
open-enclave-vm

\* Username  
philber

\* Authentication type  
Password **SSH public key**

\* SSH public key  
Pp/X7nSs38t1gg9qPMAebCwWcEy2Jstj  
n8QD8oi9koi8pxQkhFnu0LaO4Uj3pDp  
txa+hD/MzIW9C/c8Q== rsa-key-

\* Include Open Enclave SDK  
Yes

Subscription  
Windows Azure MSDN - Visual Studio Ulti...

\* Resource group  
(New) RG-OPEN-ENCLAVE  
[Create new](#)

\* Location  
(Europe) West Europe

OK

6. Specify the required settings.

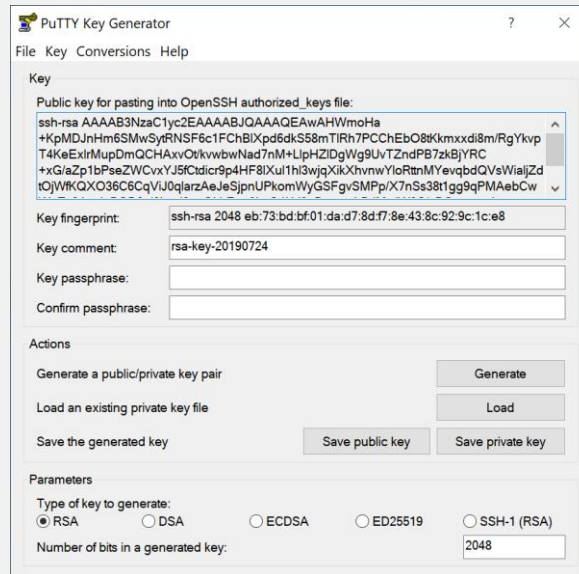
| Setting                    | Description  |
|----------------------------|--|
| <i>Image</i>               | Select <b>Ubuntu Server 18.04 LTS</b> .  |
| <i>Name</i>                | Provide the VM a hostname (as a resource, which will be displayed in Azure).                   |
| <i>Username</i>            | Specify a username for the privileged user account of the VM.                                  |
| <i>Authentication type</i> | Select <b>SSH public key</b> 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 -.

**Note**  
machine.

You can generate SSH keys by using [PuTTYGen](https://www.puttygen.com/)<sup>26</sup> on a Windows 10 local



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](#)<sup>27</sup>.

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 \(UBUNTU 18.04\)](#)<sup>28</sup>.

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 center 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.

<sup>26</sup> PuTTYGen: <https://www.puttygen.com/>

<sup>27</sup> HOW TO USE SSH KEYS WITH WINDOWS ON AZURE: <https://docs.microsoft.com/en-us/azure/virtual-machines/linux/ssh-from-windows>

<sup>28</sup> 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](https://github.com/openenclave/openenclave/blob/v0.6.x/docs/GettingStartedDocs/install_oe_sdk-Ubuntu_18.04.md)



7. 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**.

Virtual Machine Settings

Size

\* Virtual machine size ⓘ

Standard\_DC2s

Storage

\* OS disk type ⓘ

Standard SSD

Network

\* Virtual network ⓘ

(new) VirtualNetwork

\* Subnets ⓘ

Review subnet configuration

\* Select public inbound ports ⓘ

None SSH (Linux)/RDP(Windows)

Monitoring

Boot diagnostics

Disabled Enabled

\* Diagnostic storage account ⓘ

(new) openenclavevmaca54cb878

OK

| Setting         | Description  |
|-----------------|--|
| Size            | Azure Confidential Computing (ACC) public preview VMs come in two sizes. Let <b>Standard_DC2s</b> selected by default.   |
| Storage         | Select your preferred storage type. Select <b>Standard SSD</b> .   |
| 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 <b>OK</b> to accept the default values Click on the option, then click on <b>OK</b> below the newly created subnet.   |
| Inbound ports   | Select <b>SSH (Linux)/RDP (Windows)</b> as you're going to use SSH. <div><b>Important note</b> 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 <b>none</b>; 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.</div> |

|                                   |  |
|-----------------------------------|--|
| <i>Boot diagnostics</i>           | Leave <b>Enabled</b> by default.       |
| <i>Diagnostic storage account</i> | Leave the default parameter untouched. |

- Click on **OK** to continue. Validation of your configuration settings will occur on the **Summary** page.
- If validation has passed, review your configuration and click on **OK** to continue.
- 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.

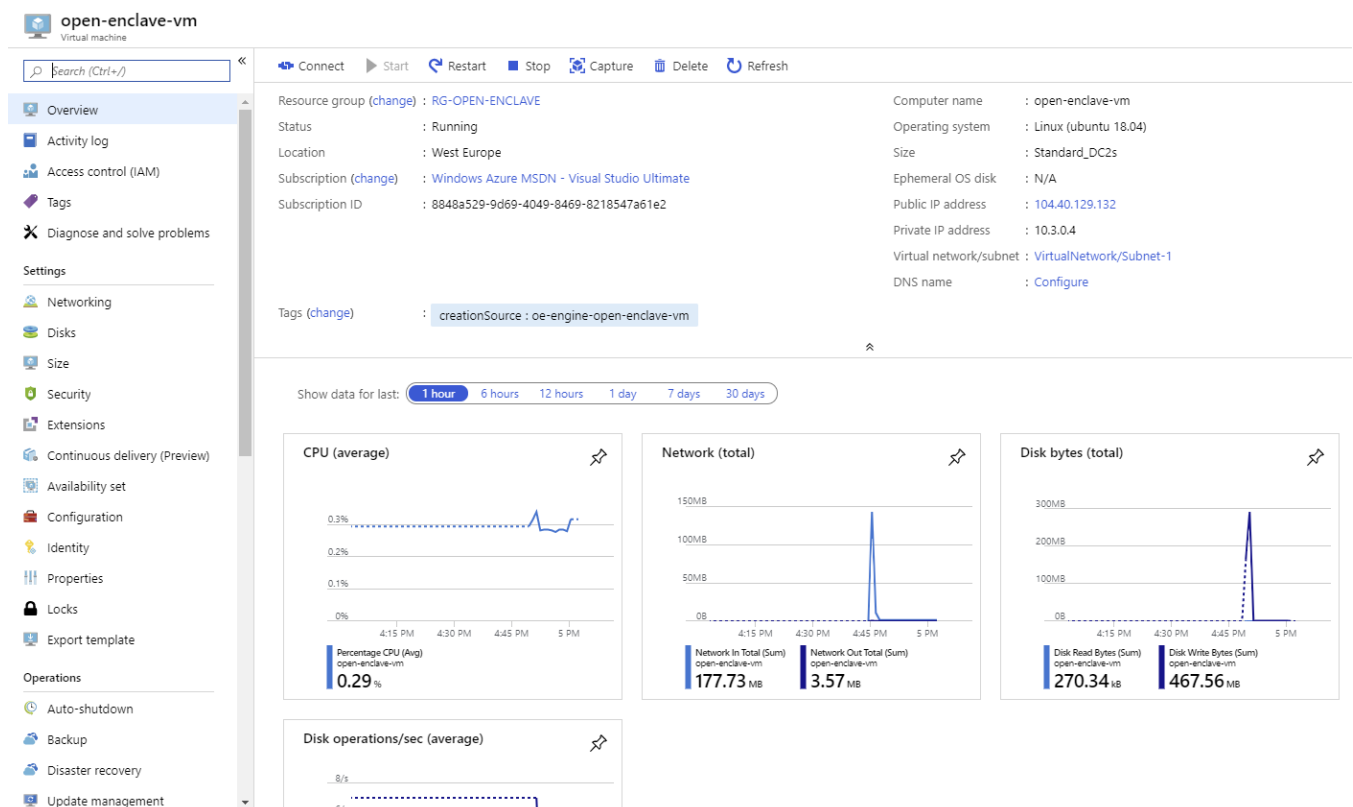
## Connecting to your DC-series VM

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:

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



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

Connect to virtual machine

open-enclave-vm

To improve security, enable just-in-time access on this VM. →

RDP SSH

To connect to your virtual machine via SSH, select an IP address, optionally change the port number, and use one of the following commands:

- 

\* IP address

Public IP address (104.40.129.132)

\* Port number

22

Login using VM local account

ssh philber@104.40.129.132

Having trouble connecting to this VM?

- [Diagnose and solve problems](#)
- [Troubleshoot connection](#)
- [Serial console](#)

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

PuTTY Configuration

Category:

- Session
- Logging
- Terminal
  - Keyboard
  - Bell
  - Features
- Window
  - Appearance
  - Behaviour
  - Translation
  - Selection
  - Colours
- Connection
  - Data
  - Proxy
  - Telnet
  - Rlogin
  - SSH
    - Kex
    - Host keys
    - Cipher
    - Auth
      - GSSAPI
      - TTY
      - X11

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address) 104.40.129.132 Port 22

Connection type:

☐ Raw ☐ Telnet ☐ Rlogin ☒ SSH ☐ Serial

Load, save or delete a stored session

Saved Sessions

Default Settings

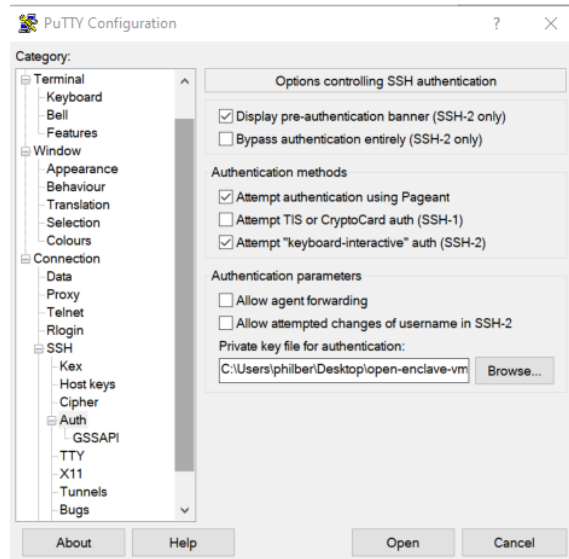
Load Save Delete

Close window on exit:

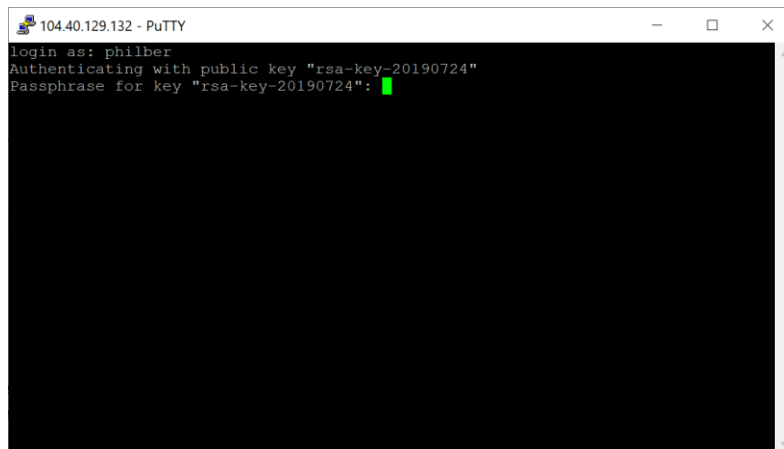
☐ Always ☐ Never ☒ Only on clean exit

About Help Open Cancel

5. In the **Category** pane, expand **Connection**, expand **SSH**, and then click on **Auth**. Complete the followings:
  - a. Choose **Browse**.
  - b. 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**.



6. Click on **Open**.
7. 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.
8. Choose **Yes**. A window opens.
9. Enter your username and press ENTER.



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

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

```
philber@open-enclave-vm: ~  
  
System information as of Wed Jul 24 15:59:57 UTC 2019  
System load:  0.01      Processes:      116  
Usage of /:   6.9% of 28.90GB  Users logged in:  0  
Memory usage: 5%      IP address for eth0: 10.3.0.4  
Swap usage:   0%  
  
1 package can be updated.  
1 update is a security update.  
  
The programs included with the Ubuntu system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*/copyright.  
  
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by  
applicable law.  
  
To run a command as administrator (user "root"), use "sudo <command>".  
See "man sudo_root" for details.  
philber@open-enclave-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  |
|---|--|
| <code>bin</code>                          | All the developer tools for developing, debugging and signing TEE-based applications using the Open Enclave SDK.   |
| <code>include/openenclave</code>          | Open Enclave runtime headers for use in your application enclave ( <code>enclave.h</code> ) and its host application ( <code>host.h</code> )   |
| <code>include/openenclave/3rdparty</code> | Headers for <code>libc</code> , <code>libcxx</code> and <code>mbedtls</code> libraries for use inside the enclave.   |
| <code>lib/openenclave/cmake</code>        | 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. |
| <code>lib/openenclave/enclave</code>      | Libraries for linking into the enclave, including the <code>libc</code> , <code>libcxx</code> and <code>mbedtls</code> libraries for Open Enclave.   |
| <code>lib/openenclave/host</code>         | Library for linking into the host application process of the enclave.  |
| <code>lib/openenclave/debugger</code>     | Libraries used by the <code>gdb</code> plug-in for debugging enclaves.   |
| <code>share/pkgconfig</code>              | Pkg-config files for header and library includes when building TEE-based applications using the Open Enclave SDK.  |
| <code>share/openenclave/samples</code>    | Sample applications’ code showing how to use the Open Enclave SDK.   |

**Note** For more information, see article [USING THE OPEN ENCLAVE SDK](#)<sup>29</sup>.

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  |
|--|--|
| <a href="#">HelloWorld</a> <sup>30</sup>         | Minimum code needed for an Open Enclave application. Help understand the basic components a TEE-based application with the Open Enclave SDK.   |
| <a href="#">File-Encryptor</a> <sup>31</sup>     | Show how to encrypt and decrypt data inside an enclave.  |
| <a href="#">Data-Sealing</a> <sup>32</sup>       | Introduce the Open Enclave sealing and unsealing features.   |
| <a href="#">Remote Attestation</a> <sup>33</sup> | Explain how the Open Enclave attestation works.<br>Demonstrate an implementation of such a remote attestation between two enclaves running on different machines.  |
| <a href="#">Local Attestation</a> <sup>34</sup>  | Explain the concept of Open Enclave local attestation.<br>Demonstrate an implementation of local attestation between two enclaves on the same machine.   |
| <a href="#">Attested TLS</a> <sup>35</sup>       | Explain what an Attested TLS channel is. See article <a href="#">WHAT IS AN ATTESTED TLS CHANNEL</a> <sup>36</sup> .<br>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](#)<sup>37</sup>.

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
```

<sup>29</sup> USING THE OPEN ENCLAVE SDK: [https://github.com/openenclave/openenclave/blob/master/docs/GettingStartedDocs/using\\_oe\\_sdk.md](https://github.com/openenclave/openenclave/blob/master/docs/GettingStartedDocs/using_oe_sdk.md)

<sup>30</sup> HelloWorld sample: <https://github.com/openenclave/openenclave/blob/master/samples/helloworld/README.md>

<sup>31</sup> File-Encryptor sample: <https://github.com/openenclave/openenclave/blob/master/samples/file-encryptor/README.md>

<sup>32</sup> Data-Sealing sample: <https://github.com/openenclave/openenclave/blob/master/samples/data-sealing/README.md>

<sup>33</sup> Remote Attestation sample: [https://github.com/openenclave/openenclave/blob/master/samples/remote\\_attestation/README.md](https://github.com/openenclave/openenclave/blob/master/samples/remote_attestation/README.md)

<sup>34</sup> Local Attestation sample: [https://github.com/openenclave/openenclave/blob/master/samples/local\\_attestation/README.md](https://github.com/openenclave/openenclave/blob/master/samples/local_attestation/README.md)

<sup>35</sup> Attested TLS sample: [https://github.com/openenclave/openenclave/blob/master/samples/attested\\_tls/README.md](https://github.com/openenclave/openenclave/blob/master/samples/attested_tls/README.md)

<sup>36</sup> WHAT IS AN ATTESTED TLS CHANNEL:

[https://github.com/openenclave/openenclave/blob/master/samples/attested\\_tls/AttestedTLSREADME.md#what-is-an-attested-tls-channel](https://github.com/openenclave/openenclave/blob/master/samples/attested_tls/AttestedTLSREADME.md#what-is-an-attested-tls-channel)

<sup>37</sup> OPEN ENCLAVE SDK SAMPLES: <https://github.com/openenclave/openenclave/blob/master/samples/README.md>

3. Change the owner of the sample applications' code directory from root to your account:

```
sudo chown -R philber ~/mysamples/
```

4. Before building any sample application code, you first need to source the file *openenclaverc* to setup environment variables for sample building. 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.

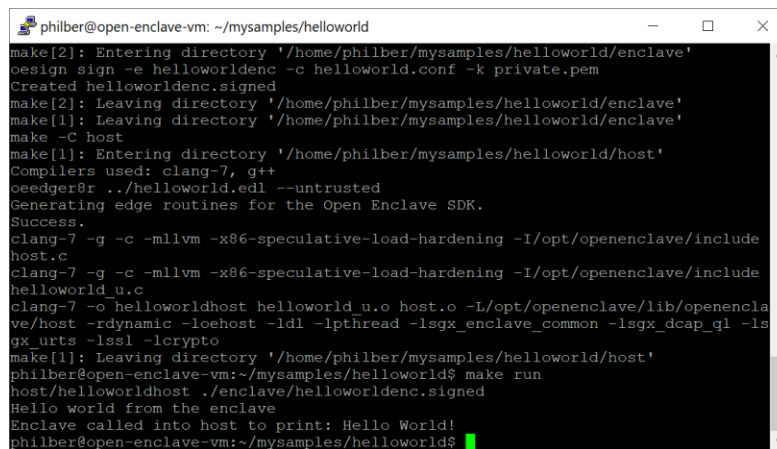
5. Go to the sample applications' directory:

```
cd ~/mysamples
```

6. 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!



```
philber@open-enclave-vm: ~/mysamples/helloworld
make[2]: Entering directory '/home/philber/mysamples/helloworld/enclave'
oesign sign -e helloworldenc -c helloworld.conf -k private.pem
Created helloworldenc.signed
make[2]: Leaving directory '/home/philber/mysamples/helloworld/enclave'
make[1]: Leaving directory '/home/philber/mysamples/helloworld/enclave'
make -C host
make[1]: Entering directory '/home/philber/mysamples/helloworld/host'
Compilers used: clang-7, g++
oeedger8r ../helloworld.edl --untrusted
Generating edge routines for the Open Enclave SDK.
Success.
clang-7 -g -c -mllvm -x86-speculative-load-hardening -I/opt/openenclave/include
host.c
clang-7 -g -c -mllvm -x86-speculative-load-hardening -I/opt/openenclave/include
helloworld_u.c
clang-7 -o helloworldhost helloworld_u.o host.o -L/opt/openenclave/lib/openencla
ve/host -rdynamic -loehost -ldl -lpthread -lsgx_enclave_common -lsgx_dcap_q1 -ls
gx_urts -lssl -lcrypto
make[1]: Leaving directory '/home/philber/mysamples/helloworld/host'
philber@open-enclave-vm:~/mysamples/helloworld$ make run
host/helloworldhost ./enclave/helloworldenc.signed
Hello world from the enclave
Enclave called into host to print: Hello World!
philber@open-enclave-vm:~/mysamples/helloworld$
```

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 for Linux with the Open Enclave SDK in C and C++ and will help you understand the key characteristics of such applications.

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

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.

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

In the former 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](#)<sup>38</sup>.

Visual Studio will be used 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 Linux machine.
- A Linux VM running on your Windows 10 development machine.

**Note** To install the Open Enclave SDK, see articles [INSTALL THE OPEN ENCLAVE SDK \(UBUNTU 18.04\)](#)<sup>39</sup> or [INSTALL THE OPEN ENCLAVE SDK \(UBUNTU 16.04\)](#)<sup>40</sup>.

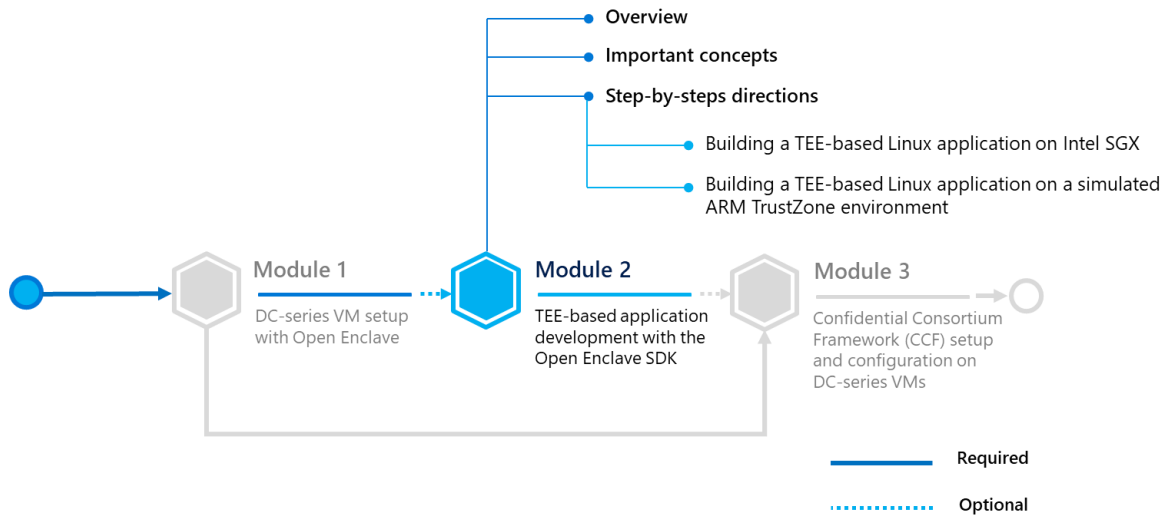
---

<sup>38</sup> DETERMINE THE SGX SUPPORT LEVEL: <https://github.com/openenclave/openenclave/blob/master/docs/GettingStartedDocs/SGXSupportLevel.md>

<sup>39</sup> INSTALL THE OPEN ENCLAVE SDK (UBUNTU 18.04):  
[https://github.com/microsoft/openenclave/blob/master/docs/GettingStartedDocs/install\\_oe\\_sdk-Ubuntu\\_18.04.md](https://github.com/microsoft/openenclave/blob/master/docs/GettingStartedDocs/install_oe_sdk-Ubuntu_18.04.md)

<sup>40</sup> INSTALL THE OPEN ENCLAVE SDK (UBUNTU 16.04):  
[https://github.com/microsoft/openenclave/blob/master/docs/GettingStartedDocs/install\\_oe\\_sdk-Ubuntu\\_16.04.md](https://github.com/microsoft/openenclave/blob/master/docs/GettingStartedDocs/install_oe_sdk-Ubuntu_16.04.md)





Before dive in, let's consider some important concepts notably regarding some of the data marshalling and unmarshalling principles and how to transfer control between the host application and the secure enclave.

## 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**  
[ENCLAVE EDGER8R](#)<sup>41</sup>.

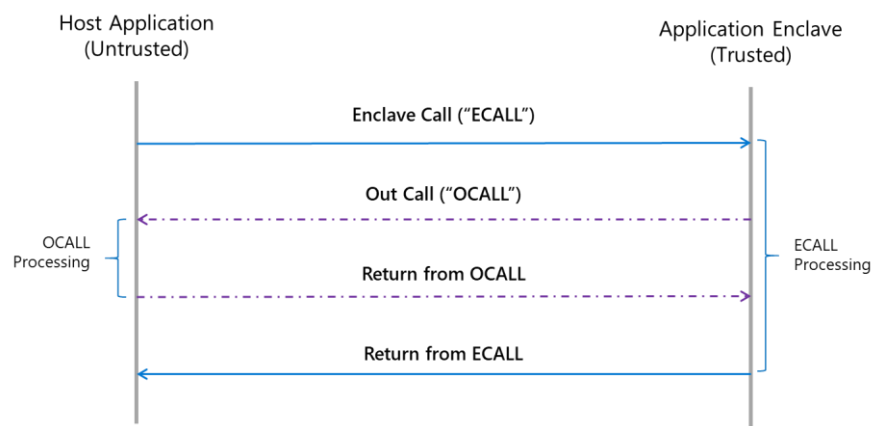
For information about the Open Enclave edger8r tool, see article [GETTING STARTED WITH THE OPEN](#)

## 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](#)<sup>42</sup>. 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-
2. 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.

<sup>41</sup> GETTING STARTED WITH THE OPEN ENCLAVE EDGER8R:

[https://github.com/openenclave/openenclave/blob/feature.new\\_platforms/docs/GettingStartedDocs/Edger8rGettingStarted.md](https://github.com/openenclave/openenclave/blob/feature.new_platforms/docs/GettingStartedDocs/Edger8rGettingStarted.md)

<sup>42</sup> Enclave Definition Language File Syntax - Intel® Developer Zone: <https://software.intel.com/en-us/sgx-sdk-dev-reference-enclave-definition-language-file-syntax>

An EDL file may include other EDL files and is processed using the Open Enclave edger8r tool, i.e. ootedger8r, which generates boilerplate code for you.

## Data marshalling

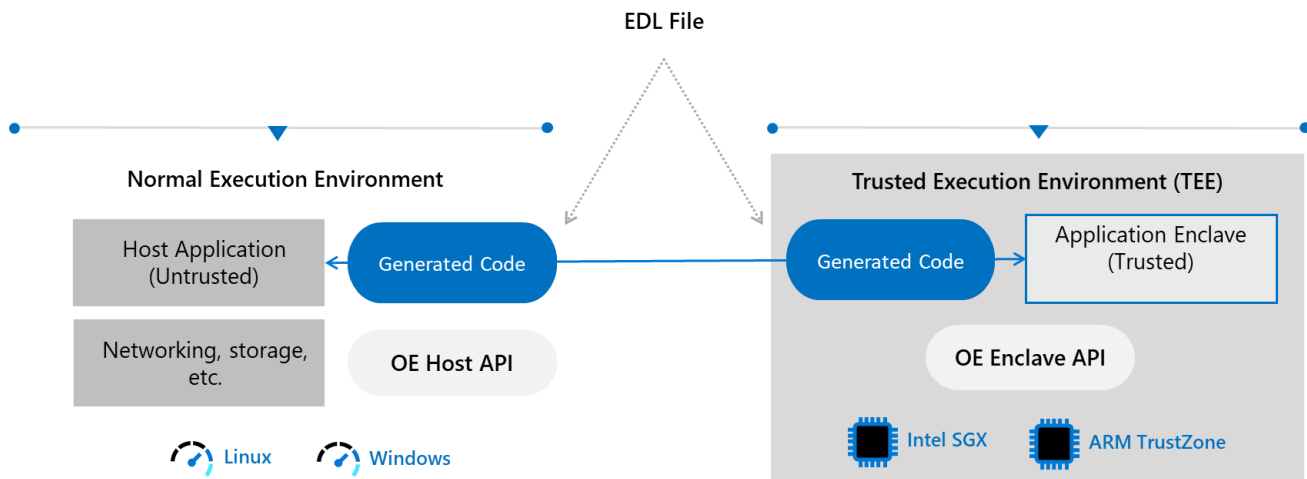
Calling into and out of enclaves is done through special methods that switch into and out of the enclave, along with the marshaling of parameters that are passed into these functions. A lot of the code necessary to handle these calls and parameter marshaling are common to all function calls. Marshaling 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](#)<sup>43</sup>.

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 ootedger8r is executed with the `--untrusted` flag),
- Conversely, a source (`<enclave>_t.c`) and a header file (`<enclave>_t.h`) to be included by the enclave (when ootedger8r 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 ootedger8r tool, see article [GETTING STARTED WITH THE OPEN ENCLAVE EDGER8R](#)<sup>44</sup>.

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.



<sup>43</sup> Meltdown and Spectre: <https://meltdownattack.com/>

<sup>44</sup> GETTING STARTED WITH THE OPEN ENCLAVE EDGER8R:  
<https://github.com/openenclave/openenclave/tree/master/docs/GettingStartedDocs/Edger8rGettingStarted.md>

# Step-by-step directions

This module covers the following two activities:

1. Building a TEE-based Linux application on Intel SGX.
2. Building a TEE-based Linux application on a simulated ARM TrustZone environment.

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](#)<sup>45</sup> and [USING VISUAL STUDIO TO DEVELOP ENCLAVE APPLICATIONS FOR LINUX](#)<sup>46</sup>.

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](#)<sup>47</sup>.

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](#)<sup>48</sup>, 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**.)

---

<sup>45</sup> 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>

<sup>46</sup> USING VISUAL STUDIO TO DEVELOP ENCLAVE APPLICATIONS FOR LINUX: [https://github.com/openenclave/openenclave/blob/feature.new\\_platforms/docs/GettingStartedDocs/VisualStudioLinux.md](https://github.com/openenclave/openenclave/blob/feature.new_platforms/docs/GettingStartedDocs/VisualStudioLinux.md)

<sup>47</sup> Open Enclave Wizard – Preview extension: <https://marketplace.visualstudio.com/items?itemName=MS-TCPS.OpenEnclaveSDK-VSIX>

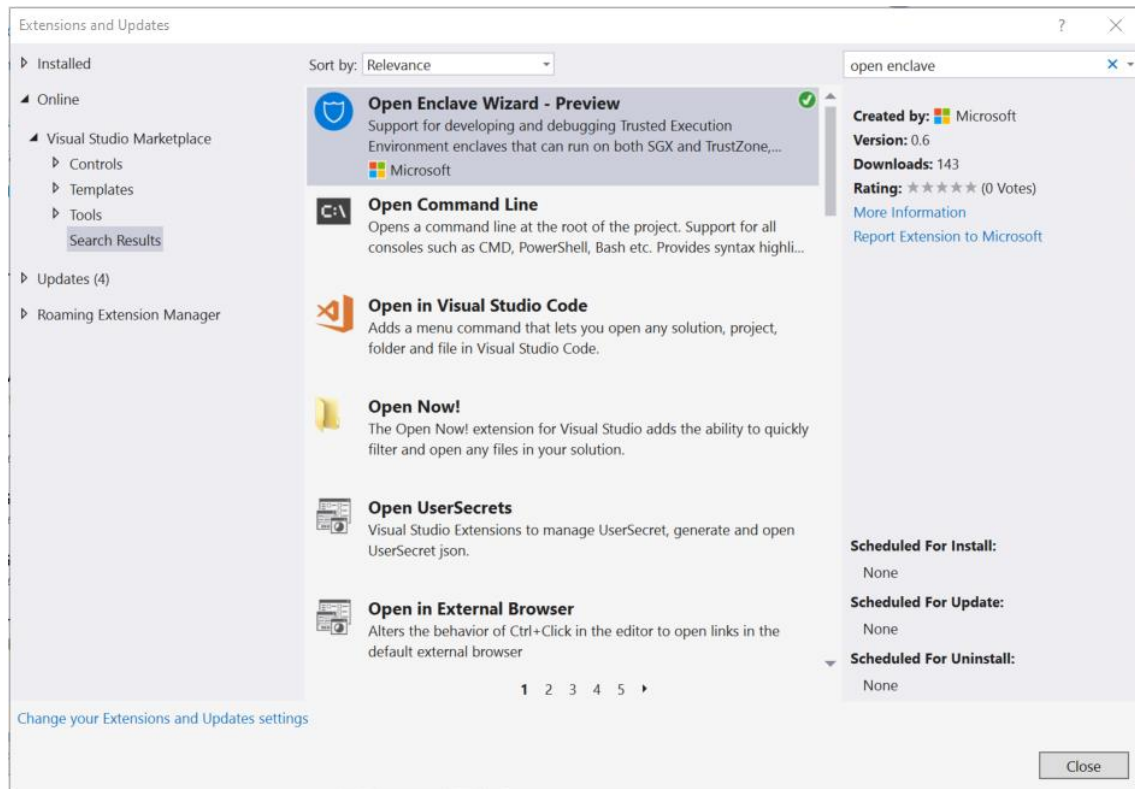
<sup>48</sup> Visual Studio Community: <https://visualstudio.microsoft.com/vs/community>

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

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

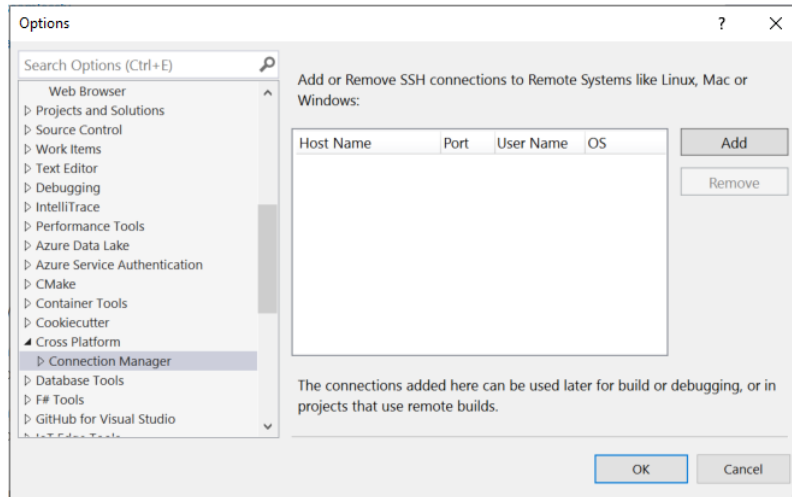
**Note** In Visual Studio 2019, select **Extensions -> Manage Extensions -> Online**.

6. Search for "*Open Enclave Wizard – Preview*", install the extension.

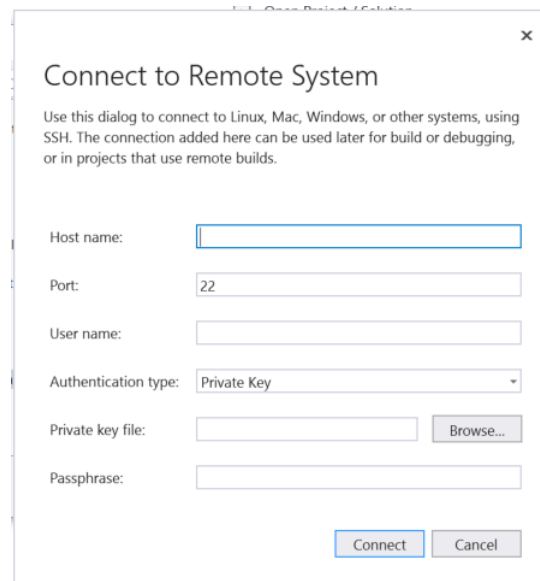


**You MUST restart Visual Studio after installing the extension to complete the installation.**

7. 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**.



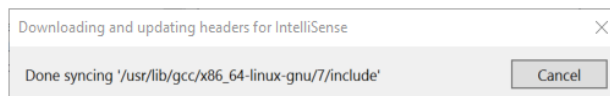
Click on **Add**. A Connect to Remote System dialog opens up.



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

**Important note** The PuTTY private key .ppk file must be first converted to the OpenSSH format. You can use the

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)<sup>49</sup>.

<sup>49</sup> 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>

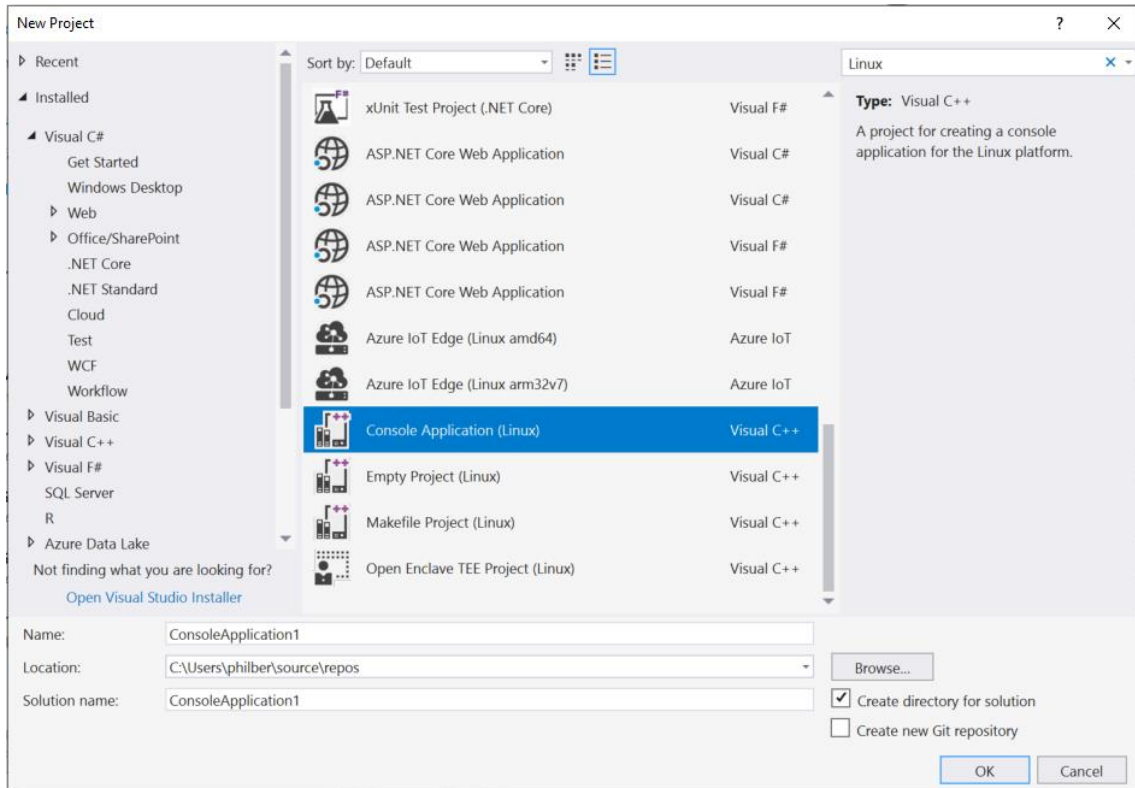
**Congrats! Your Windows 10 development machine is now fully configured.**

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*".

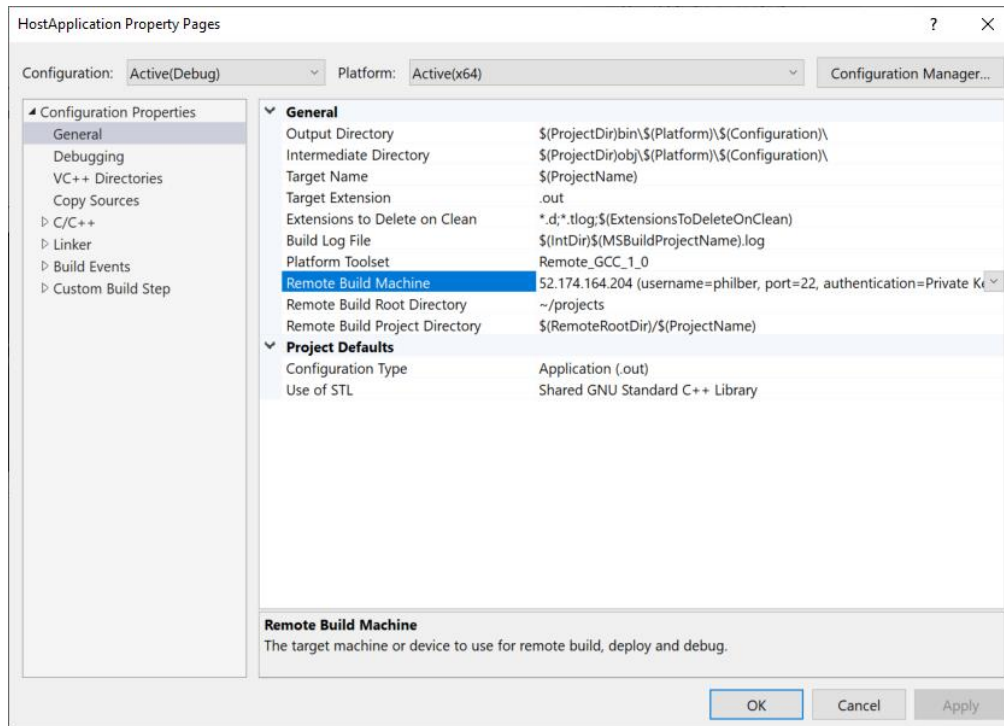


**Important note** This is NOT the "Console App (.NET Core)".

**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**.)

- a. Enter a name, for example "HostApp" in our illustration, a location, and solution name in the appropriate fields like any other Visual Studio project.
  - b. Click on **OK**. This will create a "Hello World" console application.
3. 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.



- 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)<sup>50</sup> and [LINUX DEBUGGING WALKTHROUGH](https://docs.microsoft.com/en-us/cpp/linux/deploy-run-and-debug-your-linux-project?view=vs-2017)<sup>51</sup>.

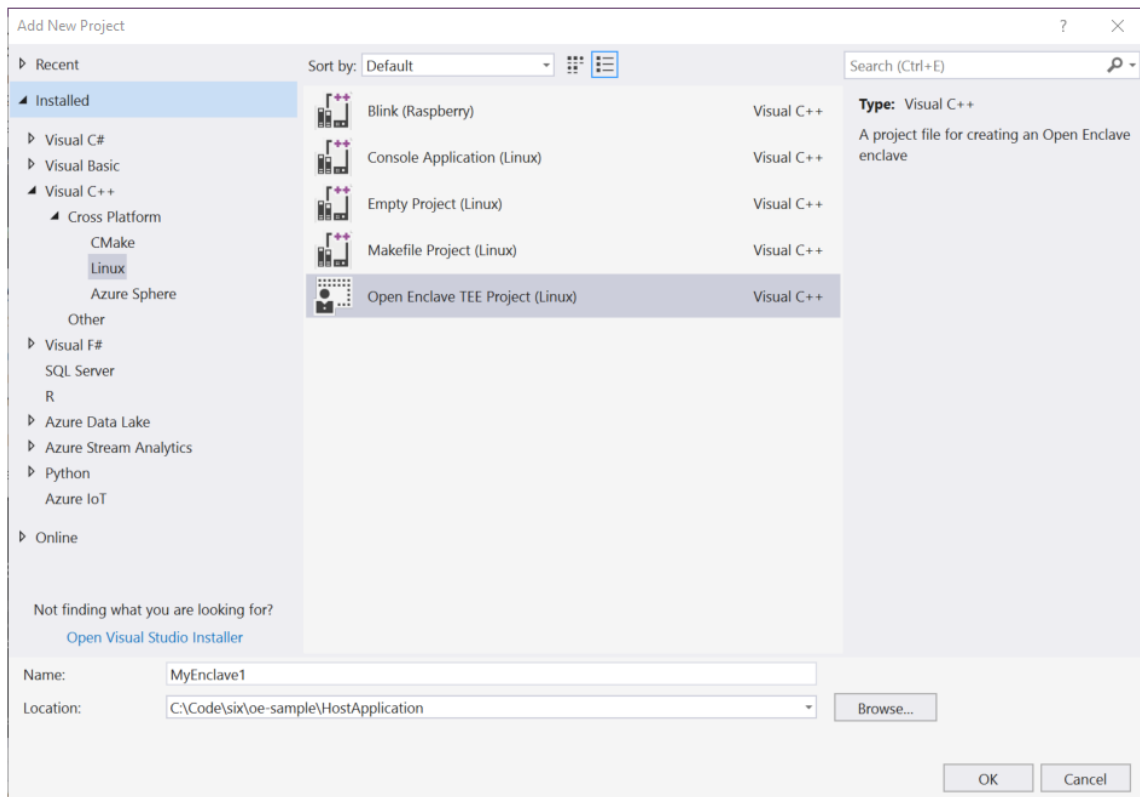
You now have your first application project, but this is only the host application. You now need to create the enclave component.

- 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.

<sup>50</sup> 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>

<sup>51</sup> LINUX DEBUGGING WALKTHROUGH: <https://docs.microsoft.com/en-us/cpp/linux/deploy-run-and-debug-your-linux-project?view=vs-2017>

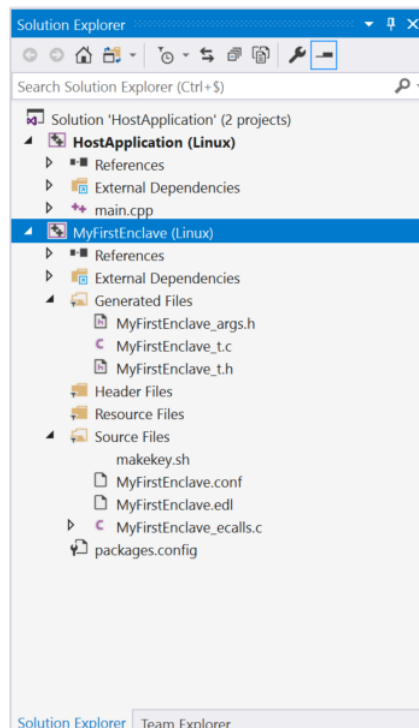




(If it is not immediately visible, look under **Installed > Visual C++ > Cross Platform > Linux**.)

6. 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.

7. 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...
9. You now need to import the enclave into your host application project. To do so, right click on your host application project in the **Solution 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.

8. 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 <stdio>

extern "C" {
    void sample_enclave_call(void);
};

int main()
{
    printf("hello from LinuxApp!\n");
    sample_enclave_call();
    return 0;
}
```

8. 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](#)<sup>52</sup>.

As noticed in the module's introduction, **this guide will assume that you use, as a developer, a Windows 10 development machine with the Windows Subsystem for Linux (WSL) feature installed on it.**

---

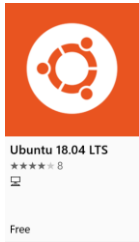
<sup>52</sup> OPEN ENCLAVE EXTENSION FOR VISUAL STUDIO CODE:

[https://github.com/openenclave/openenclave/blob/feature.new\\_platforms/new\\_platforms/vscode-extension/README.md](https://github.com/openenclave/openenclave/blob/feature.new_platforms/new_platforms/vscode-extension/README.md)

This environment allows you to notably:

- Make it easy for new contributors to get started and keep everyone on a consistent environment.
- Sandbox your development environment to avoid impacting your local machine configuration.
- Use tools or runtimes not available on your local OS or manage multiple versions of them.
- Develop your Linux-deployed applications using the WSL feature.

To get started, if not already present on your local Windows 10 machine, install WSL along with your preferred Linux distribution, for example Ubuntu18.04 from the Windows Store.



You will then 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.

Visual Studio Code will be installed **on the Windows side** (not in WSL).

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

1. The [Open Enclave extension for Visual Studio Code](#)<sup>53</sup>, supporting the Open Enclave SDK, including development, debugging, emulators, and deployment.

You will use WSL and the [Visual Studio Code Remote Development Extension Pack](#)<sup>54</sup> hereafter to create standalone project on a Windows desktop. On Windows, by default, you can only create an Azure IoT Edge Module project – On Linux, you have the options to create both projects by default -. In the suggested configuration, you will need to install in the WSL subsystem of your Windows 10 machine all the [requirements](#)<sup>55</sup> listed.

2. The Visual Studio Code Remote Development Extension Pack that allows you to open any folder in a container, in the aforementioned WSL subsystem, or on a remote machine like your DC-series VM in Azure. This pack includes three extensions:
  - a. [Visual Studio Code Remote - WSL](#)<sup>56</sup>. It allows you to get a Linux-powered development experience from the comfort of Windows 10 by opening any folder in WSL.
  - b. [Visual Studio Code Remote - SSH](#)<sup>57</sup>. 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.

---

<sup>53</sup> Open Enclave extension for Visual Studio Code: [https://marketplace.visualstudio.com/items?itemName=ms-iot.msio-vsc-Open Enclave](https://marketplace.visualstudio.com/items?itemName=ms-iot.msio-vsc-Open%20Enclave)

<sup>54</sup> Visual Studio Code Remote Development Extension Pack: <https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vsc-remote-extensionpack>

<sup>55</sup> Open Enclave extension for Visual Studio Code: <https://marketplace.visualstudio.com/items?itemName=ms-iot.msio-vsc-Openenclave#Requirements>

<sup>56</sup> Visual Studio Code Remote – WSL: <https://aka.ms/vscode-remote/download/wsl>

<sup>57</sup> Visual Studio Code Remote – SSH: <https://aka.ms/vscode-remote/download/ssh>

- c. [Visual Studio Code Remote - Containers](#)<sup>58</sup>. It allows you to work with a sandboxed toolchain or container-based application by opening any folder inside (or mounted into) a container.

For this activity, you will use the first extension.

Perform the following steps:

1. Install Visual Studio Code or [Visual Studio Code – Insiders](#)<sup>59</sup>.

**Important note** When prompted to **Select Additional Tasks** during installation, be sure to check the **Add to PATH** option so you can easily open a folder in WSL using the code command.

2. Install [Git for Windows](#)<sup>60</sup> (2.10 or later) and make sure that long paths are enabled:

```
git config --global core.longpaths true
```

3. 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 [DEVELOPING IN WSL](#)<sup>61</sup>, [REMOTE DEVELOPMENT USING SSH](#)<sup>62</sup>, and [DEVELOPING INSIDE A CONTAINER](#)<sup>63</sup>. For troubleshooting tips and tricks for each of these extensions, see article [REMOTE DEVELOPMENT TIPS AND TRICKS](#)<sup>64</sup>.

4. Optionally install an [OpenSSH compatible SSH client](#)<sup>65</sup> if one is not already present on your local Windows 10 machine.
5. Open a WSL terminal window and launch Visual Studio Code from here:
  - a. Open a WSL terminal window using the Start Menu item or by typing “wsl” from the command prompt.
  - b. Navigate to a folder you'd like to open in Visual Studio Code (including, but not limited to, Windows filesystem mounts like the folder `/mnt/c` for the C drive on your Windows 10 local machine.
  - c. Type in the terminal the following command:

```
code .
```

When doing this for the first time, you should see Visual Studio Code fetching components needed to run in WSL. This should only take short while and is only needed once.

---

<sup>58</sup> Visual Studio Code Remote – Containers: <https://aka.ms/vscode-remote/download/containers>

<sup>59</sup> Visual Studio Code – Insiders: <https://code.visualstudio.com/insiders>

<sup>60</sup> Git for Windows: <https://git-for-windows.github.io/>

<sup>61</sup> DEVELOPING IN WSL: <https://code.visualstudio.com/docs/remote/wsl>

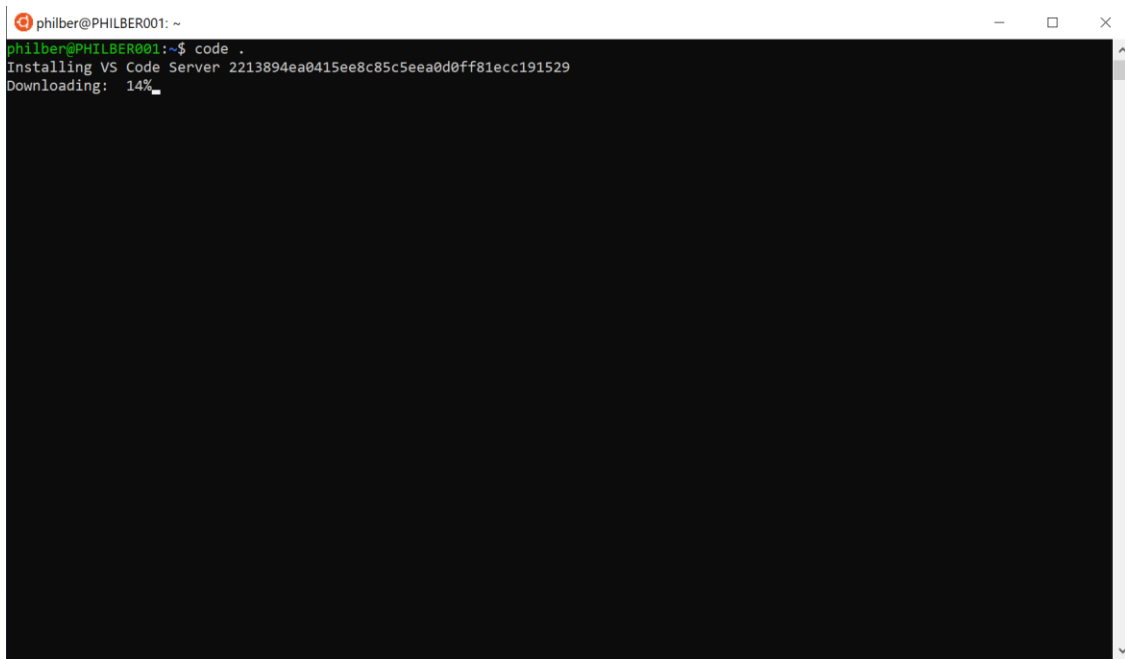
<sup>62</sup> REMOTE DEVELOPMENT USING SSH: <https://code.visualstudio.com/docs/remote/ssh>

<sup>63</sup> DEVELOPING INSIDE A CONTAINER: <https://code.visualstudio.com/docs/remote/containers>

<sup>64</sup> REMOTE DEVELOPMENT TIPS AND TRICKS: [https://code.visualstudio.com/docs/remote/troubleshooting#\\_installing-a-supported-ssh-client](https://code.visualstudio.com/docs/remote/troubleshooting#_installing-a-supported-ssh-client)

<sup>65</sup> INSTALLATION OF OPENSSH FOR WINDOWS SERVER 2019 AND WINDOWS 10: [https://docs.microsoft.com/en-us/windows-server/administration/openssh/openssh\\_install\\_firstuse](https://docs.microsoft.com/en-us/windows-server/administration/openssh/openssh_install_firstuse)

**Note** If this command does not work, you may not have added VS Code to your path when it was installed.



```
philber@PHILBER001: ~  
philber@PHILBER001:~$ code .  
Installing VS Code Server 2213894ea0415ee8c85c5eea0d0ff81ecc191529  
Downloading: 14%
```

- d. After a moment, a new Visual Studio Code window will open up, and you'll see a notification that Visual Studio Code is starting in WSL.

① Starting VS Code in WSL (Ubuntu):

Visual Studio Code will now continue to configure itself in WSL and keep you up to date as it makes progress. Once finished, you now see a WSL indicator in the bottom left corner, and you'll be able to use Visual Studio Code as you would normally!

>< WSL: Ubuntu

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

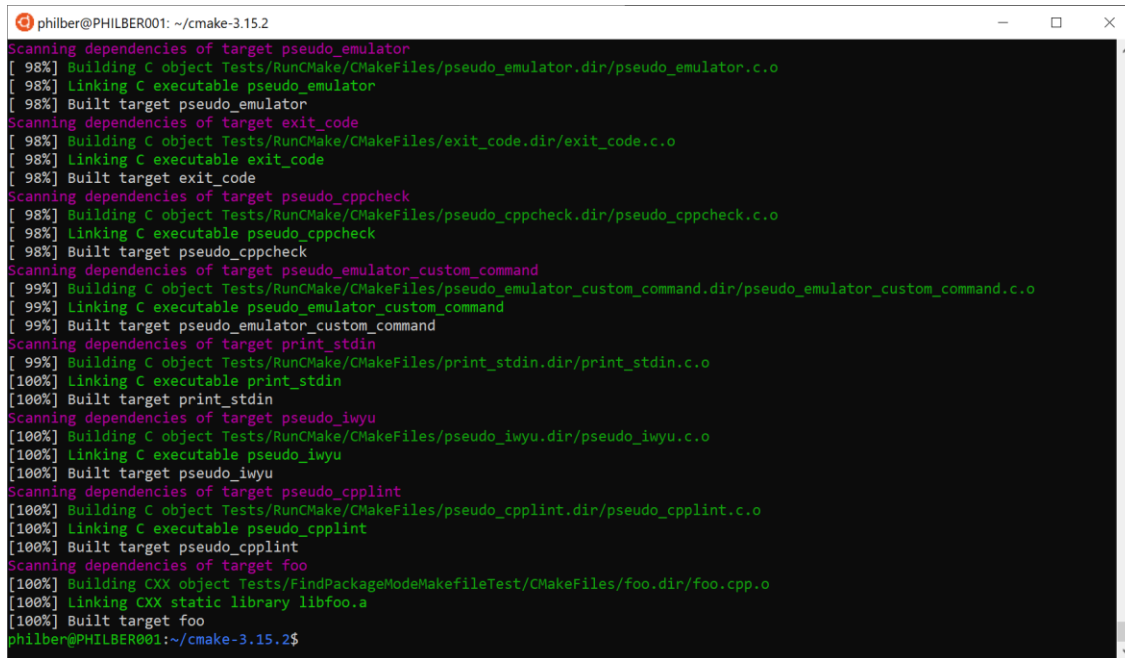
6. Ensure that the [requirements](#)<sup>66</sup> are met for the Open Enclave extension for Visual Studio Code:
- In Visual Studio Code, make sure that the [Visual Studio Code Native Debug extension](#)<sup>67</sup> is installed. Press CTRL-p and run `ext install webfreak.debug` in Visual Studio Code and install GDB/LLDB.
  - Install [CMake 3.12 or above](#)<sup>68</sup>, currently 3.15. You can do it either from the previous WSL terminal window or from Visual Studio Code by pressing F1 and the selecting Remote-WSL: New Window.
    - Download the archive file `cmake-3.15.2.tar.gz` and compile it:

<sup>66</sup> Open Enclave extension for Visual Studio Code: <https://marketplace.visualstudio.com/items?itemName=ms-iot.msio-vc-openenclave#Requirements>

<sup>67</sup> Native Debug extension: <https://marketplace.visualstudio.com/items?itemName=webfreak.debug>

<sup>68</sup> CMake download: <https://cmake.org/download/>

```
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
```



```
philber@PHILBER001: ~/cmake-3.15.2
Scanning dependencies of target pseudo_emulator
[ 98%] Building C object Tests/RunCMake/CMakeFiles/pseudo_emulator.dir/pseudo_emulator.c.o
[ 98%] Linking C executable pseudo_emulator
[ 98%] Built target pseudo_emulator
Scanning dependencies of target exit_code
[ 98%] Building C object Tests/RunCMake/CMakeFiles/exit_code.dir/exit_code.c.o
[ 98%] Linking C executable exit_code
[ 98%] Built target exit_code
Scanning dependencies of target pseudo_cppcheck
[ 98%] Building C object Tests/RunCMake/CMakeFiles/pseudo_cppcheck.dir/pseudo_cppcheck.c.o
[ 98%] Linking C executable pseudo_cppcheck
[ 98%] Built target pseudo_cppcheck
Scanning dependencies of target pseudo_emulator_custom_command
[ 99%] Building C object Tests/RunCMake/CMakeFiles/pseudo_emulator_custom_command.dir/pseudo_emulator_custom_command.c.o
[ 99%] Linking C executable pseudo_emulator_custom_command
[ 99%] Built target pseudo_emulator_custom_command
Scanning dependencies of target print_stdin
[ 99%] Building C object Tests/RunCMake/CMakeFiles/print_stdin.dir/print_stdin.c.o
[100%] Linking C executable print_stdin
[100%] Built target print_stdin
Scanning dependencies of target pseudo_iwyu
[100%] Building C object Tests/RunCMake/CMakeFiles/pseudo_iwyu.dir/pseudo_iwyu.c.o
[100%] Linking C executable pseudo_iwyu
[100%] Built target pseudo_iwyu
Scanning dependencies of target pseudo_cpplint
[100%] Building C object Tests/RunCMake/CMakeFiles/pseudo_cpplint.dir/pseudo_cpplint.c.o
[100%] Linking C executable pseudo_cpplint
[100%] Built target pseudo_cpplint
Scanning dependencies of target foo
[100%] Building CXX object Tests/FindPackageModeMakefileTest/CMakeFiles/foo.dir/foo.cpp.o
[100%] Linking CXX static library libfoo.a
[100%] Built target foo
philber@PHILBER001:~/cmake-3.15.2$
```

- ii. 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:

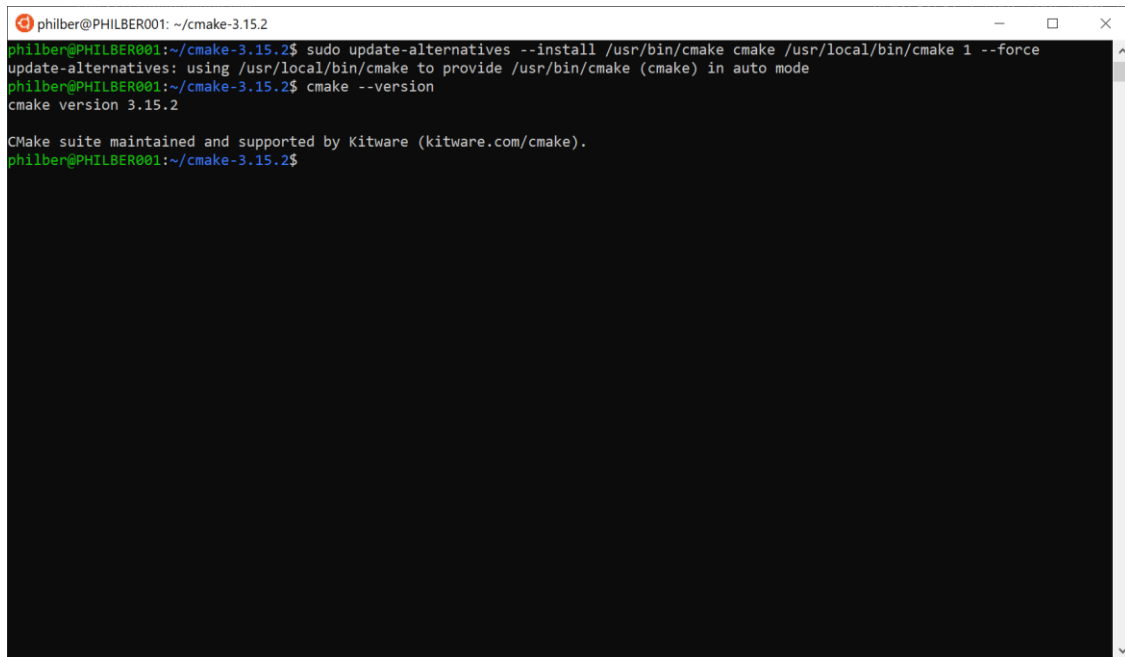
```
sudo make install
```

- iii. 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
```

A terminal window titled 'philber@PHILBER001: ~/cmake-3.15.2'. The prompt is 'philber@PHILBER001:~/cmake-3.15.2\$'. The first command is 'sudo update-alternatives --install /usr/bin/cmake cmake /usr/local/bin/cmake 1 --force'. The output is 'update-alternatives: using /usr/local/bin/cmake to provide /usr/bin/cmake (cmake) in auto mode'. The second command is 'cmake --version'. The output is 'cmake version 3.15.2'. Below this, a message reads 'CMake suite maintained and supported by Kitware (kitware.com/cmake)'. The prompt returns to 'philber@PHILBER001:~/cmake-3.15.2\$'.

c. 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 key 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:

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

```
gpg --keyserver hkps://keyserver.ubuntu.com:80 --recv EB3E94ADBE1229CF
```

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

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



```
philber@PHILBER001: ~  
Get:28 http://archive.ubuntu.com/ubuntu bionic-updates/universe amd64 Packages [983 kB]  
Get:29 http://archive.ubuntu.com/ubuntu bionic-updates/universe Translation-en [299 kB]  
Get:30 http://archive.ubuntu.com/ubuntu bionic-updates/multiverse amd64 Packages [11.9 kB]  
Get:31 http://archive.ubuntu.com/ubuntu bionic-updates/multiverse Translation-en [5764 B]  
Get:32 http://archive.ubuntu.com/ubuntu bionic-backports/main amd64 Packages [2512 B]  
Get:33 http://archive.ubuntu.com/ubuntu bionic-backports/main Translation-en [1644 B]  
Get:34 http://archive.ubuntu.com/ubuntu bionic-backports/universe amd64 Packages [3732 B]  
Get:35 http://archive.ubuntu.com/ubuntu bionic-backports/universe Translation-en [1696 B]  
Reading package lists... Done  
W: GPG error: https://packages.microsoft.com/repos/azure-cli bionic InRelease: The following signatures couldn't be verified because the public key 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 is therefore disabled by default.  
N: See apt-secure(8) manpage for repository creation and user configuration details.  
philber@PHILBER001:~$ gpg --keyserver hkps://keyserver.ubuntu.com:80 --recv EB3E94ADBE1229CF  
gpg: directory '/home/philber/.gnupg' created  
gpg: keybox '/home/philber/.gnupg/pubring.kbx' created  
gpg: key EB3E94ADBE1229CF: 5 signatures not checked due to missing keys  
gpg: /home/philber/.gnupg/trustdb.gpg: trustdb created  
gpg: key EB3E94ADBE1229CF: public key "Microsoft (Release signing) <gpgsecurity@microsoft.com>" imported  
gpg: no ultimately trusted keys found  
gpg: Total number processed: 1  
gpg: imported: 1  
philber@PHILBER001:~$ gpg --export --armor EB3E94ADBE1229CF | sudo apt-key add -  
OK  
philber@PHILBER001:~$
```

- iii. Rerun the above command to install the required build components.
- d. Ensure that all QEMU dependencies are installed. [QEMU](https://www.qemu.org/)<sup>69</sup> 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 WSL, on Ubuntu 18.04, run:

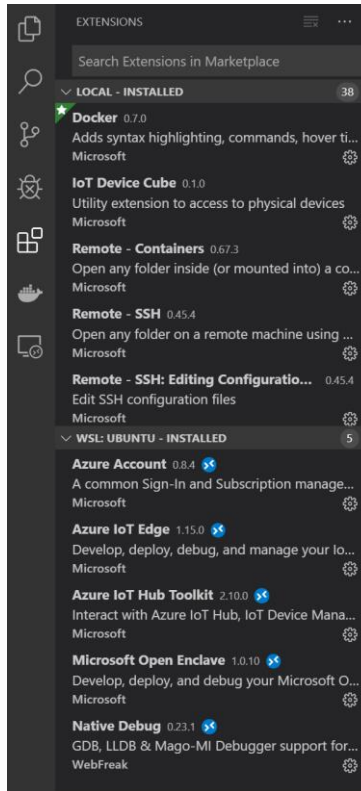
```
sudo apt update && sudo apt install -y libpixman-1-0 zlib1g libc6 libfdt1 libglib2.0-0 libpcrc3  
libstdc++6
```

- 7. 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) in WSL. While extensions that affect the Visual Studio Code UI, like themes and snippets, are installed locally, most extensions will reside inside WSL. This will be the case for the Open Enclave extension for Visual Studio Code.

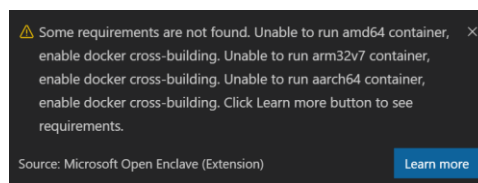
---

<sup>69</sup> QEMU: <https://www.qemu.org/>



8. 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

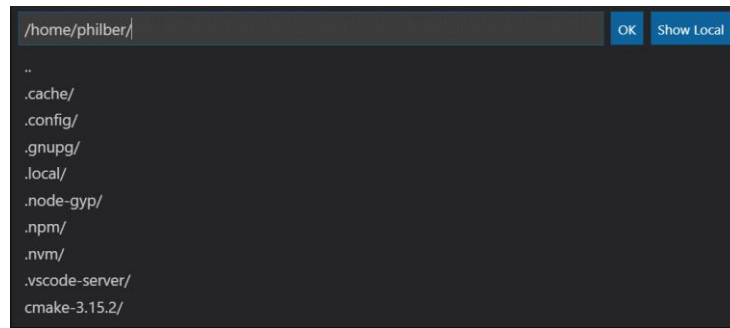
Perform the following steps:

1. Launch Visual Studio Code.
2. As already stated, and illustrated above, the Visual Studio Code Remote - WSL extension lets you use the WSL 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.

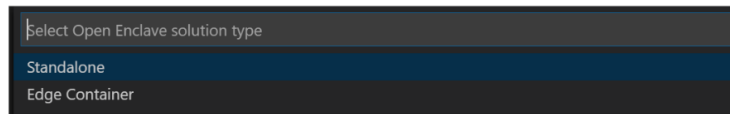
Press F1 and select Remote-WSL: New Window for the default Ubuntu 18.04 distro (or Remote-WSL: New Window using Distro if you have installed multiple distros in WSL using Windows Store).

**Note** For more information, see article [Developing in WSL](#)<sup>70</sup>.

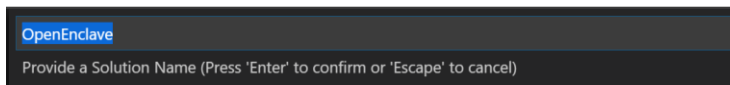
3. Use the Microsoft Open Enclave: New Open Enclave Solution command - commands can be found using F1 or CTRL-Shift-P - to create your first new standalone application.



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

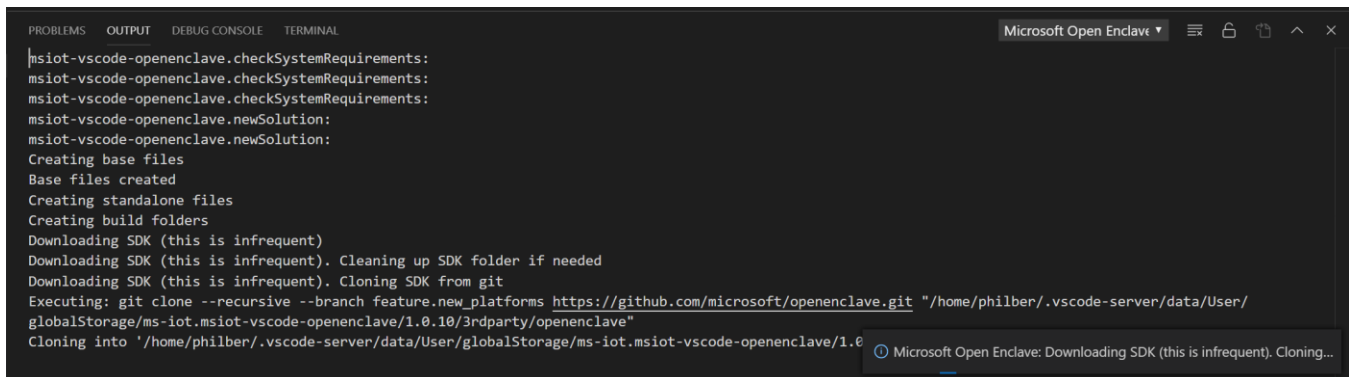


5. When invited to choose the option, create a **Standalone** project.



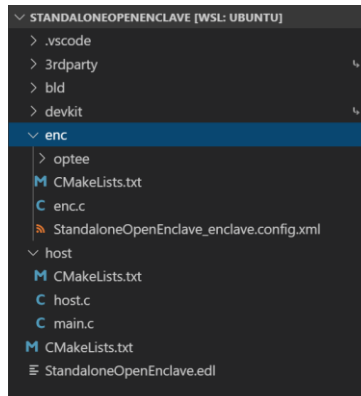
6. Provide a name for the host application/enclave, and press ENTER to confirm.

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.

<sup>70</sup> DEVELOPING IN WSL: <https://code.visualstudio.com/docs/remote/wsl>



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);
    };

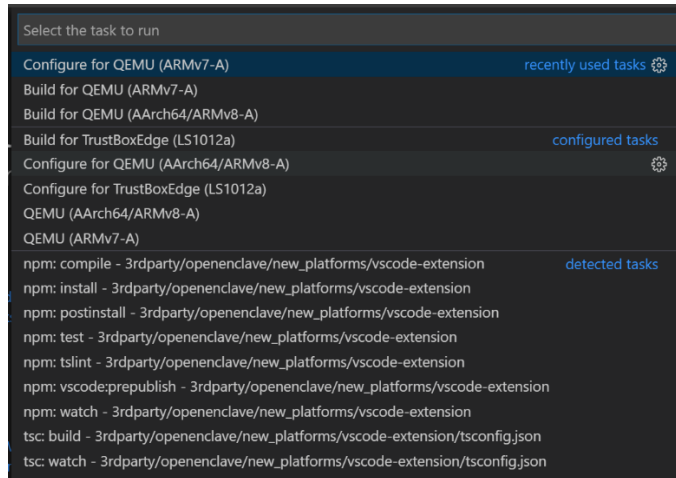
    untrusted {
        /* define OCALLs here. */
        int ocall_log([in, string] char *msg);
    };
};
```

### 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. Using 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**, **AArch64/ARMv8-A**, and **TrustBoxEdge(LS1012a)**.



3. Select **Configure for QEMU ARMv7-A**. The configure task will invoke cmake to create the required build files. This is only required to be run once.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: Task - Configure for QEMU (ARMv7-A)

> Executing task: cmake -DQTEE=TZ -DTA_DEV_KIT_DIR=/mnt/c/code/six/sa-sample/StandaloneOpenEnclave/devkit/devkits/vexpress-qemu_virt/export-ta_arm32 -DQTEE_TA_TOOLCHAIN_PREFIX=arm-linux-gnueabi -DQTEE_TOOLCHAIN_PREFIX=arm-linux-gnueabi -DQTEE_TOOLCHAIN_FILE=/mnt/c/code/six/sa-sample/StandaloneOpenEnclave/3rdparty/openenclave/new_platforms/cmake/linux-arm-v6.cmake -DQTEE_BUILD_TYPE=Debug -DQTEE_C_FLAGS=-no-pie /mnt/c/code/six/sa-sample/StandaloneOpenEnclave <

CMake Warning at CMakeLists.txt:6 (project):
  VERSION keyword not followed by a value or was followed by a value that
  expanded to nothing.

-- The C compiler identification is GNU 7.4.0
-- The CXX compiler identification is GNU 7.4.0
-- Check for working C compiler: /usr/bin/arm-linux-gnueabi-gcc
-- Check for working C compiler: /usr/bin/arm-linux-gnueabi-gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/arm-linux-gnueabi-g++

```

4. Select **Build for QEMU ARMv7-A**. The build task will do the actual compiling and linking.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
2: Task - Build for QEMU (ARMv7-A)

[ 39%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/pkwrite.c.o
[ 42%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/platform.c.o
[ 42%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/ripemd160.c.o
[ 42%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/rsa.c.o
[ 42%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/sha256.c.o
[ 42%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/rsa_internal.c.o
[ 45%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/sha512.c.o
[ 45%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/sha1.c.o
[ 45%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/threading.c.o
[ 45%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/version.c.o
[ 45%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/xtea.c.o
[ 45%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/version_features.c.o
[ 48%] Building C object out/mbdts/library/CMakeFiles/mbdcrypto.dir/timing.c.o
[ 48%] Linking C static library ../lib/libmbdcrypto.a
[ 48%] Built target mbdcrypto
In file included from /usr/arm-linux-gnueabi/include/c++/7/vector:64:0,
                 from /home/philber/oe-sample/OpenEnclave/3rdparty/openenclave/new_platforms/3rdparty/googletest/googletest/include/gtest/gtest.h:57,
                 from /home/philber/oe-sample/OpenEnclave/3rdparty/openenclave/3rdparty/googletest/googletest/src/gtest-all.cc:38:

```

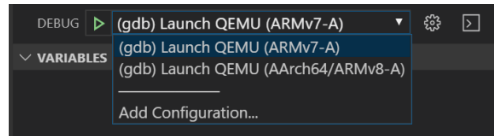
## Debugging the standalone C/C++ application

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:
  - a. Navigate to the Visual Studio Debug view – Use CTRL-Shift-D -.



- b. Select (gdb) Launch QEMU (ARMv7-A) from the debug configuration dropdown.
3. 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.
4. Open the terminal view.
5. Log into QEMU using root (no password is required).

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

**Now that you've known the basics on how to get an up and running TEE-based application, 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

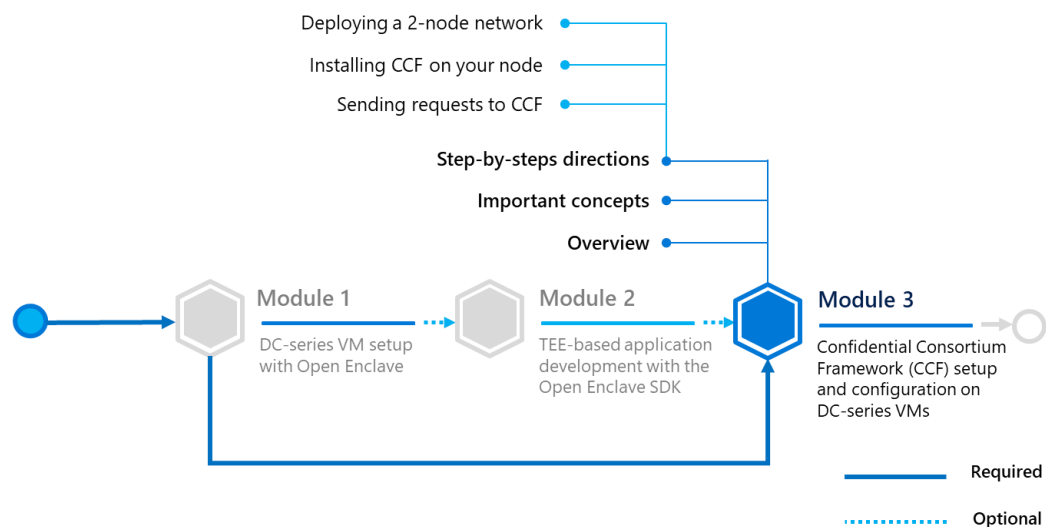
In this last module, you will learn about the [Confidential Consortium Framework](https://aka.ms/ccf)<sup>71</sup> (CCF). This project, led by Microsoft Research, is a Microsoft's open source framework based on Open Enclave (OE) for building a new category of secure, highly available, and performant applications that focus on multi-party compute and data.

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.

<sup>71</sup> Confidential Consortium Framework: <https://aka.ms/ccf>

# 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/governed 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 lose 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)<sup>72</sup> (PoW) or [Proof of stake](https://en.wikipedia.org/wiki/Proof_of_stake)<sup>73</sup> (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.

---

<sup>72</sup> Proof of work: [https://en.wikipedia.org/wiki/Proof\\_of\\_work](https://en.wikipedia.org/wiki/Proof_of_work)

<sup>73</sup> Proof of stake: [https://en.wikipedia.org/wiki/Proof\\_of\\_stake](https://en.wikipedia.org/wiki/Proof_of_stake)



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

## Confidential Computing Framework (CCF)

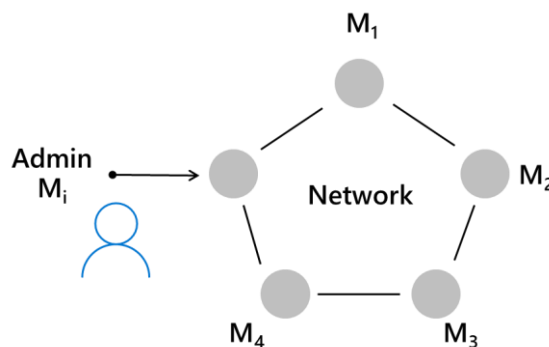
In this context, Microsoft tried another approach: what if the algorithm were just a way to establish a consensus between nodes, and the security part would be managed by something else, such as Trusted Execution Environments (TEEs) or enclaves.

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, the Confidential Consortium Framework (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)<sup>74</sup> 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:

---

<sup>74</sup> Paxos (computer science): [https://en.wikipedia.org/wiki/Paxos\\_%28computer\\_science%29](https://en.wikipedia.org/wiki/Paxos_%28computer_science%29)

- **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](#)<sup>75</sup>, or the website [at this address](#).<sup>76</sup>

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.

---

<sup>75</sup> Confidential Consortium Framework: <https://github.com/microsoft/CCF>

<sup>76</sup> Welcome to CCF's documentation: <https://microsoft.github.io/CCF/>

Virtual Machine Settings

Size

\* Virtual machine size ⓘ

Standard\_DC2s

Storage

\* OS disk type ⓘ

Standard SSD

Network

\* Virtual network ⓘ

VirtualNetwork

Subnets ⓘ

Configure subnets

\* Select public inbound ports ⓘ

None SSH (Linux)/RDP(Windows)

Monitoring

Boot diagnostics

Disabled Enabled

\* Diagnostic storage account ⓘ

(new) openenclavevm276edace1...

OK

Choose virtual network

These are the virtual networks in the selected subscription and location 'West Europe'.

Create new

VirtualNetwork RG-OPEN-ENCLAVE

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**.

Virtual Machine Settings

Size

\* Virtual machine size ⓘ

Standard\_DC2s

Storage

\* OS disk type ⓘ

Premium SSD

Network

\* Virtual network ⓘ

VirtualNetwork

Subnets ⓘ

Configure subnets

\* Select public inbound ports ⓘ

None SSH (Linux)/RDP(Windows)

Monitoring

Boot diagnostics

Disabled Enabled

\* Diagnostic storage account ⓘ

(new) openenclavevm28a5e3a6c...

OK

Subnets

\* Subnet

Subnet-1

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.

Connected devices

| DEVICE               | TYPE              | IP ADDRESS | SUBNET   |
|----------------------|-------------------|------------|----------|
| open-enclave-vm-nic  | Network interface | 10.3.0.4   | Subnet-1 |
| open-enclave-vm2-nic | Network interface | 10.3.0.5   | Subnet-1 |

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 ~
```

4. 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
```

5. 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.
  - a. Simply type the following command:

```
cd CCF/getting_started/setup_vm
```

- b. and then, execute the script:

```
./setup.sh
```

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

6. The next step consists in building CCF from its source files.

- a. Go back into the root directory of CCF by typing:

```
cd ../../
```

- b. Create a *build* directory:

```
mkdir build
```

- c. Then, go inside it:

```
cd build
```

- d. 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)

- e. 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)<sup>77</sup> 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).

---

<sup>77</sup> GETTING STARTED: [https://microsoft.github.io/CCF/getting\\_started.html](https://microsoft.github.io/CCF/getting_started.html)

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](#)<sup>78</sup> 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](#)<sup>79</sup>), 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**.

---

<sup>78</sup> STARTING UP THE NETWORK: [https://microsoft.github.io/CCF/start\\_network.html](https://microsoft.github.io/CCF/start_network.html)

<sup>79</sup> Guide samples on the GitHub repo: <https://aka.ms/CCDevGuideSamples>

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.

```
nsix@nsix-vm-1:~/CCF/build$ ./client --pretty-print --host 10.1.0.4 --port 38888  
--ca networkcert.pem userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "get  
NetworkInfo"}' --cert user0_cert.pem --pk user0_privk.pem  
Sending RPC to 10.1.0.4:38888  
Doing user RPC:  
{  
  "commit": 4,  
  "global_commit": 4,  
  "id": 2,  
  "jsonrpc": "2.0",  
  "result": {  
    "leader_id": 0,  
    "nodes": [  
      {  
        "host": "10.1.0.4",  
        "node_id": 0,  
        "port": "38888"  
      },  
      {  
        "host": "10.1.0.5",  
        "node_id": 1,  
        "port": "38888"  
      }  
    ]  
  },  
  "term": 2  
}
```

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)<sup>80</sup>.

---

<sup>80</sup> RPC API: [https://microsoft.github.io/CCF/rpc\\_api.html](https://microsoft.github.io/CCF/rpc_api.html)

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

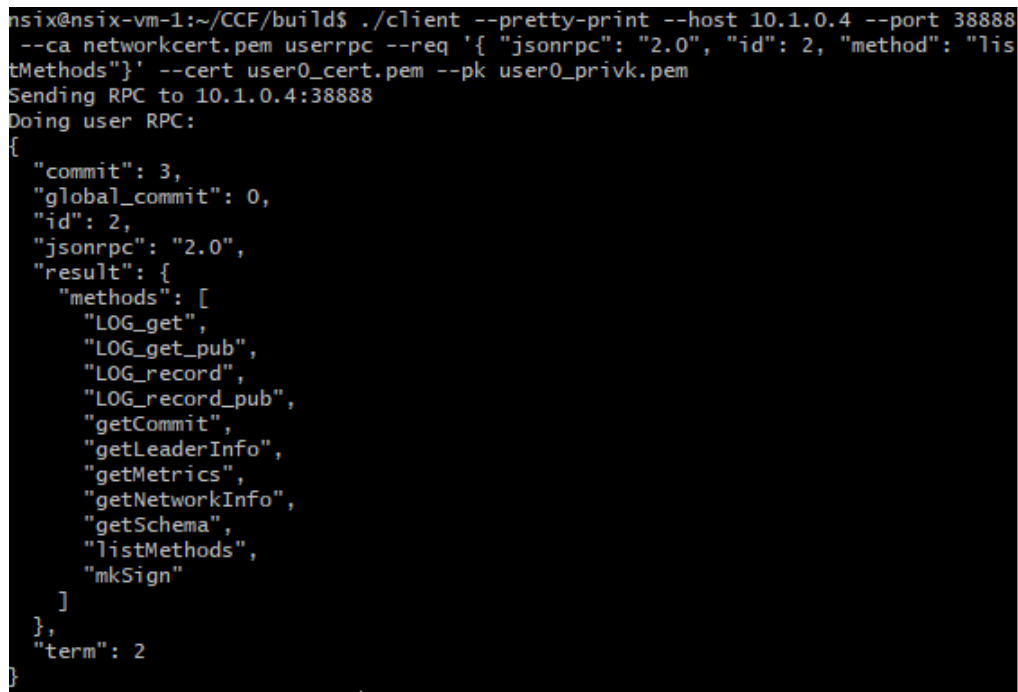
```
./client -pretty-print -host <your node IP> --port <your node TLS port> --ca networkcert.pem userrpc  
--req '{ "jsonrpc": "2.0", "id": 2, "method": "<method name>", "params": {<method parameters>}}' --  
cert user0_cert.pem --pk user0_privk.pem
```

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 come 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:



```
nsix@nsix-vm-1:~/CCF/build$ ./client --pretty-print --host 10.1.0.4 --port 38888  
--ca networkcert.pem userrpc --req '{ "jsonrpc": "2.0", "id": 2, "method": "lis  
tMethods"}' --cert user0_cert.pem --pk user0_privk.pem  
Sending RPC to 10.1.0.4:38888  
Doing user RPC:  
{  
  "commit": 3,  
  "global_commit": 0,  
  "id": 2,  
  "jsonrpc": "2.0",  
  "result": {  
    "methods": [  
      "LOG_get",  
      "LOG_get_pub",  
      "LOG_record",  
      "LOG_record_pub",  
      "getCommit",  
      "getLeaderInfo",  
      "getMetrics",  
      "getNetworkInfo",  
      "getSchema",  
      "listMethods",  
      "mkSign"  
    ]  
  },  
  "term": 2  
}
```

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))<sup>81</sup>, 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.

---

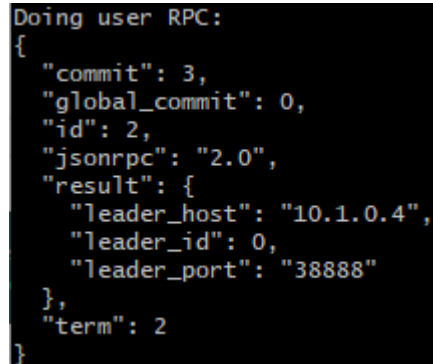
<sup>81</sup> Raft (computer science): [https://en.wikipedia.org/wiki/Raft\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Raft_(computer_science))



In your context, you 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
```



```
Doing user RPC:  
{  
  "commit": 3,  
  "global_commit": 0,  
  "id": 2,  
  "jsonrpc": "2.0",  
  "result": {  
    "leader_host": "10.1.0.4",  
    "leader_id": 0,  
    "leader_port": "38888"  
  },  
  "term": 2  
}
```

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, you need to generate him/her his/her cert and his/her private key:

```
./genesisgenerator cert -name "newmember"
```

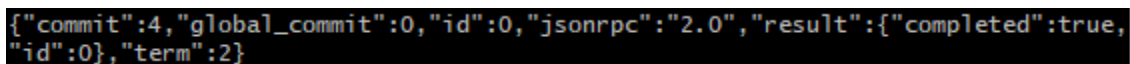
You should see those new files when doing ls inside your directory.

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:



```
{"commit":4,"global_commit":0,"id":0,"jsonrpc":"2.0","result":{"completed":true,  
"id":0},"term":2}
```

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_record` but 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.

```
nsix@nsix-vm-1:~/CCF/build$ ./client --pretty-print --host 10.1.0.4 --port 38888 --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
Sending RPC to 10.1.0.4:38888
Doing user RPC:
{
  "commit": 10,
  "global_commit": 0,
  "id": 2,
  "jsonrpc": "2.0",
  "result": true,
  "term": 2
}
nsix@nsix-vm-1:~/CCF/build$
```

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:

```
nsix@nsix-vm-1:~/CCF/build$ ./client --pretty-print --host 10.1.0.4 --port 38888  
--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  
Sending RPC to 10.1.0.4:38888  
Doing user RPC:  
{  
  "commit": 11,  
  "global_commit": 0,  
  "id": 2,  
  "jsonrpc": "2.0",  
  "result": {  
    "msg": "Hello there"  
  },  
  "term": 2  
}
```

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

## 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.