



Elektrobit

SDK for EB corbos AdaptiveCore

2.3.0

Getting started guide

Release

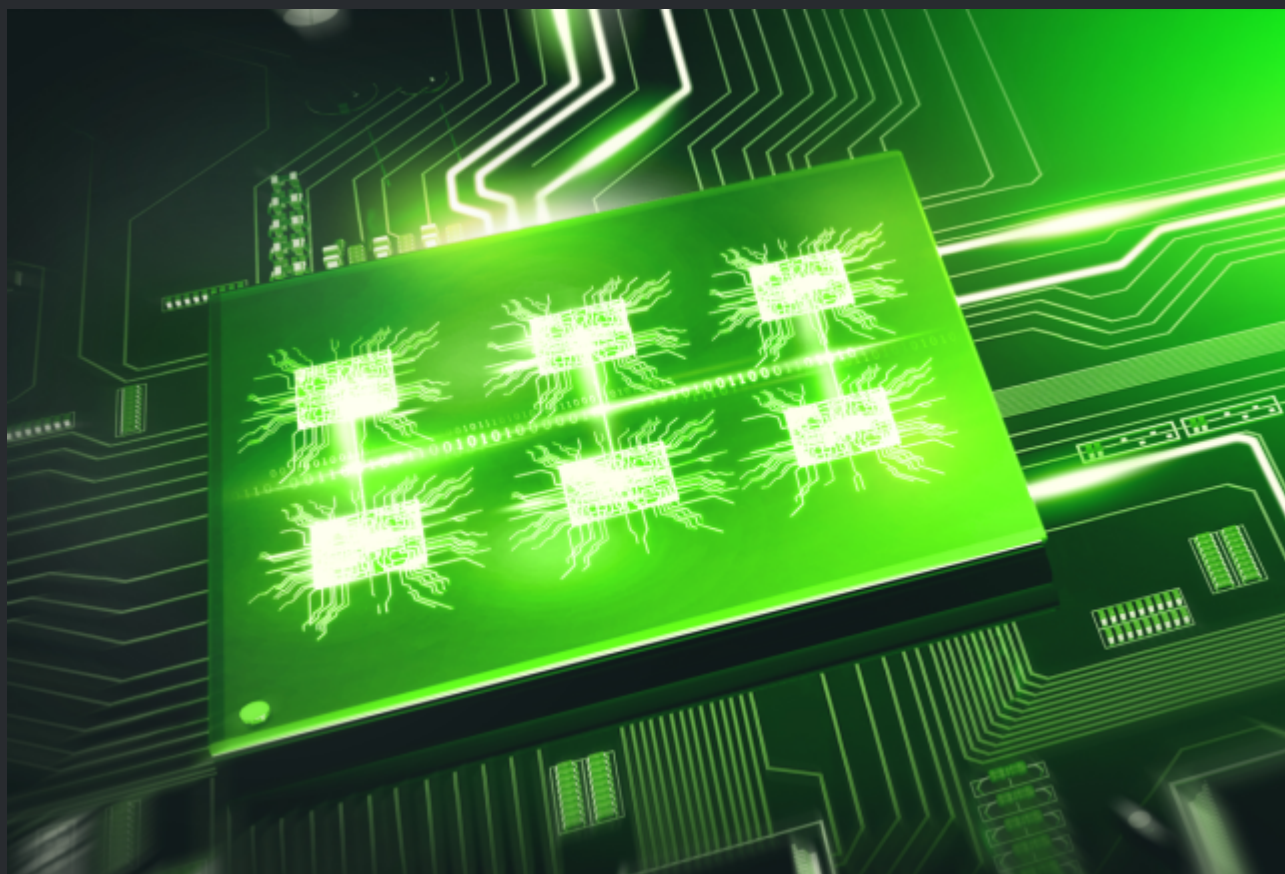


Table of Contents

Legal notice	3
Typographic and style conventions	4
About	6
Prerequisites	7
Setting up the environment	8
Configure the environment connection	8
Configuring the remote environment	8
Configuring the API key for the generators	8
Sample application for the DLT module	9
Overview	9
Structure	9
Building and running	10
Sample application for the COM module	11
Overview	11
Structure	12
Building and running	12
Sample application for the DM module	13
Overview	13
Structure	14
Building and running	14
Sample application for AI integration	16
Overview	16
Structure	16
Building and running	16
FAQ	17
Annex 1 - Mathworks Simulink modelling	18
Modelling the interface	18
Modelling the connection	19
Using the generated ARXML files	20
Annex 2 - JSON deployment manifests without ARXML	21
Annex 3 - Using the generators client	22
Annex 4 - Deploying to targets	23
Prepare the targets	23
Install the corbos AdaptiveCore runtime	23
Deploy the application to the target	24
Annex 5 - Using sil-kit for an overlay network	28
Prerequisites	29
Creating the tap interfaces on the EC2 instances	29
Setting up the overlay network	29
Setting up Wireshark to monitor the overlay network	31
Running the sample COM application	33

Legal notice

Elektrobit Automotive GmbH
Am Wolfsmantel 46
D-91058 Erlangen
GERMANY

Phone: +49 9131 7701-0
Fax: +49 9131 7701-6333
<http://www.elektrobit.com>


Internal Information

ALL RIGHTS RESERVED. No part of this publication may be copied in any form, by photocopy, microfilm, retrieval system, or by any other means now known or hereafter invented without the prior written permission of Elektrobit Automotive GmbH.


Copyright 2024, Elektrobit Automotive GmbH

Typographic and style conventions


The signal word *WARNING* indicates information that is vital for the success of the configuration.

WARNING 	<i>Source and kind of the problem</i>
	What can happen to the software?
	What are the consequences of the problem?
	How does the user avoid the problem?

The signal word *NOTE* indicates important information on a subject.

NOTE 	<i>Important information</i>
	Gives important information on a subject

The signal word *TIP* provides helpful hints, tips and shortcuts.

TIP 	<i>Helpful hints</i>
	Gives helpful hints

Throughout the documentation you find words and phrases that are displayed in **bold**, *italic*, or `monospaced` font.

To find out what these conventions mean, see the following table.

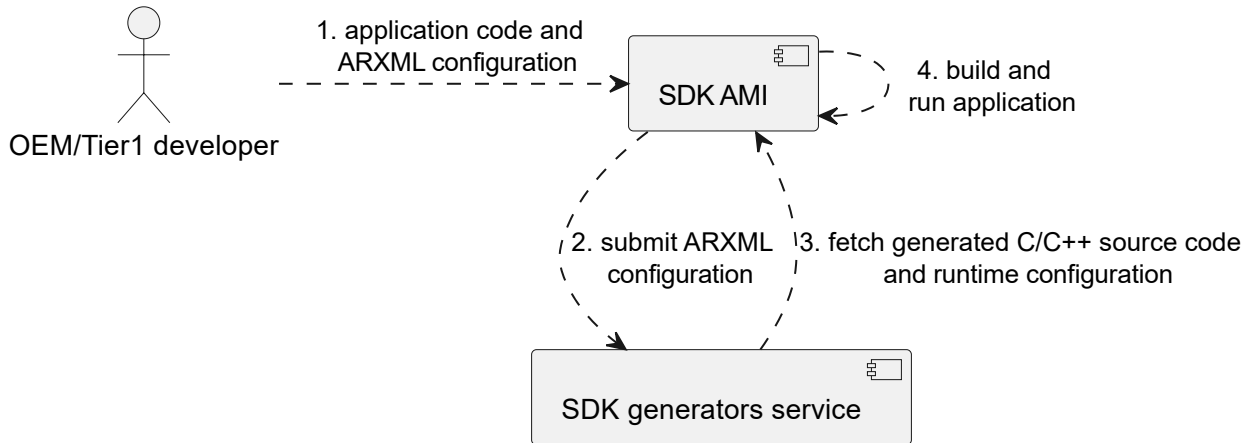
All default text is written in Arial Regular font.

Font	Description	Example
Open Sans italics	Emphasizes new or important terms	The <i>basic building blocks</i> of a configuration are module configurations.
Open Sans bold	GUI elements and keyboard keys	1. In the Project drop-down list box, select Project_A. 2. Press the Enter key.
Monospaced font (Source Code Pro)	User input, code, and file directories	The module calls the BswM_Dcm_RequestSessionMode() function. For the project name, enter Project_Test.
Square brackets []	Denotes optional parameters; for command syntax with optional parameters	insertBefore [<opt>]
Curly brackets { }	Denotes mandatory parameters; for command syntax with mandatory parameters	insertBefore {<file>}
Ellipsis ...	Indicates further parameters; for command syntax with multiple parameters	insertBefore [<opt>...]
A vertical bar	Indicates all available parameters; for command syntax in which you select one of the available parameters	allowinvalidmarkup {on off}

About

The *SDK for EB corbos AdaptiveCore* represents a cloud-based solution aimed to simplify the development process of Adaptive Autosar applications.

An overview of how the development process looks is depicted below:



The *SDK for EB corbos AdaptiveCore* is delivered with three sample applications that use the `ara::com`, `ara::dm` and `ara::dlt` functional clusters.

WARNING



The sample source code should not be used for production purposes.

CAUTION



The first build is slower because it involves generating code and runtime configurations. Subsequent builds of the sample applications are faster.

Prerequisites

- a running EC2 machine based on the SDK for EB corbos AdaptiveCore AMI (available in AWS marketplace in arm64 flavor)
- a user account on the [Elektrobit Cloud](#) portal
- VSCode locally installed on your PC configured with the Remote Development extension

Setting up the environment



If you already know how to configure the Remote Development extension in VSCode you may skip this section.

Configure the environment connection

On your local PC create a `config` file under your SSH home (e.g. `C:\users\<your_user>\.ssh` for Windows, or `/home/<your_user>/.ssh`) and put the following contents:

```
Host <your_machine_name>
  HostName <ip address of the EC2 machine>
  User ubuntu
  IdentityFile <path to your ssh key for the EC2 machine>
```

Start VSCode and connect to the remote machine by pressing `Ctrl+Shift+P` and writing `Remote-SSH:Connect to Host`. You should be able to select `<your_machine_name>` from the configuration file above.

Configuring the remote environment

Once you are connected go to `File` and select `Open Workspace from File` and then navigate to the `/home/ubuntu/samples` folder and select the `.samples.code-workspace` file.

At this point you should install the following VSCode extensions in the **remote environment**:

- CMake and CMake Tools
- C/C++



For a better UX experience (in VScode) configure CMake Tools to use "Unix Makefiles" as default generator. You can do this under `File/Preferences/Settings` and search for `cmake.generator`.

Configuring the API key for the generators

Each sample app contains a `config.json` file in its root directory:

```
{
  "folder" : "/home/ubuntu/samples/hello-com/model",
  "plugets": "AraDltModelGenerator,AraComBindingGenerator,AraComManifestGenerator",
  "pname" : "hello-com",
  "token" : "<your_token_here>",
  "url": "https://adg-generator-prod.elektrobit.cloud/generator",
  "asr": "20-11"
}
```

Please register and create a user account on [Elektrobit Cloud](#). The required API token can be generated in your account settings and can be used in the `config.json` file.

Sample application for the DLT module

This application demonstrates the usage of the Log and Trace functional cluster of corbos Adaptive Autosar platform. The DLT module implements the Adaptive Autosar `ara::Log` interface.

Overview

The application uses the DLT module client libraries : `libaralog` and `libaradltproxy` to communicate with a DLT daemon (running in the corbos Adaptive Autosar runtime) and logs several messages to a Genivi DLT compatible backend.

WARNING



The below description is just a short and incomplete summary. For a rough overview please read the Explanation of Adaptive Autosar Platform (doc no 706) and Specification of Log and Trace (doc no 853) of the Adaptive Autosar standard.

The model part configures the default Log and Trace setup of the application which requires at minimum:

- an `Executable` element
- a `Process Design` element that references the executable
- a corresponding `Process` element linked to the executable
- a `Machine` element with a corresponding `MachineDesign` element
- a `LogAndTraceInstantiation` with at least one `DltLogChannel`, the latter containing the actual logging configuration like default log level, contextId, log trace modes and further descriptions
- a `DltLogChannelToProcessMapping` that links the `Process` with the `DltLogChannel`

Structure

There are three important parts:

- The `src` folder contains the actual source code of the application
- The `model` folder contains the ARXML configuration of the DLT libraries used by the application
- The `CMakeLists.txt` defines the required C++ libraries of corbos Adaptive Autosar platform and a custom function

NOTE



The custom task executes the generator client tool against the model directory and generates the actual configuration file used by the DLT libraries linked in the application.

CAUTION



Since the configuration generation is a slower process, the custom function is executed when the build directory is clean or when the ARXML model file changes.

Building and running

Generate the build & libraries configuration using CMake (preferably through an IDE like VScode) and then compile the sources.

- Start the corbos Adaptive Autosar platform runtime using `sudo adg-prepare && sudo adg-start`
- In the `hello-dlt` directory execute `sudo adg_sandbox -e -c adg_hello-dlt_process_am.json`
- To change the input argument of the application update the `adg_hello-dlt_process_am.json` file at line 18 in the `args` section
- <Optional> Start `dlt-viewer` (requires a GUI) and filter the logs based on application id `HLD`

Sample application for the COM module

This application demonstrates the usage of the Communication Management functional cluster of corbos Adaptive Autosar platform. The COM module implements the Adaptive Autosar `ara::com` interface.

Overview

The application uses the COM module client libraries (in addition to the DLT ones): `libara_com` and `libara_core` to communicate with a COM daemon (running in the corbos Platform runtime).

It contains two executables : a `service` that registers itself with a method to the COM daemon and a `client` that requests this method from the COM daemon and then calls it.

WARNING



The below description is just a short and incomplete summary. For a rough overview please read the Explanation of Adaptive Autosar Platform (doc no 706) and Specification of Communication Management (doc no 717) of the Adaptive Autosar standard.

The model part contains the default configuration for Log and Trace (see hello-dlt example) for the two executables, an interface definition between the two executables and its corresponding *optional* SomeIP configuration.

The interface definition requires at minimum:

- a `ServiceInterface` element which represents the C++ class name of the interface
- one or more `Symbol Props` element(s) that represent the namespace under which the sources are generated
- one or more corresponding `ClientServerOperation` elements along with `ArgumentDataPrototype` that describe the actual method(s) offered by the interface
- one or more `DataType` (s) that are referenced in the method signature

TIP



The above interface defines just a simple synchronous RPC scenario. `ara::com` offers the possibility to configure much more ways to communicate between two applications.

The interface SomeIP configuration requires at minimum for each process (client and server):

- a `SomeIpSdServerServiceInstanceConfig` or `SomeIpSdClientServiceInstanceConfig` element
- a `ProvidedSomeIpServiceInstance` or `RequiredSomeIpServiceInstance` named according to the `InstanceIdentifier` used from the client and service C++ code
- a corresponding `EthernetCluster` that contains the network settings of the hosts of the client or the service
- a corresponding `EthernetCommunicationConnector` that links the `Machine` to the network settings
- a `SomeIpServiceDiscovery` that specifies the service discovery settings and links to the network settings
- a corresponding `SomeIpServiceInstanceToMachineMapping` that maps the network settings against the `ProvidedSomeIpServiceInstance` or `RequiredSomeIpServiceInstance`

Structure

There are five important parts:

- The `client` folder contains the actual source code of the client executable
- The `service` folder contains the actual source code of the service executable
- The `model` folder contains the ARXML configuration of the DLT libraries used by the two executables , the ARXML interface definition between the service and the client and the ARXML SomeIP configuration
- The `run_generators.sh` shell script that handles all the generator interaction
- The `CMakeLists.txt` defines the required C++ libraries of corbos AdaptiveCore and a custom task for DLT and COM.

NOTE



The custom task for DLT & COM executes a shell script against the model directory and generates a client implementation(proxy) and a stub(skeleton) to be implemented by the service for the defined interface. Additionally it also creates actual configuration files used by the DLT libraries linked in the each executable (two configurations - one for each executable is generated).

CAUTION



Since the configuration generation is a lengthy process, the custom task is executed when the build directory is clean or when any of the ARXML model files change.

Building and running

Generate the build & libraries configuration using CMake (preferably through an IDE like VScode) and then compile the sources.

IPC execution

IPC execution starts the two processes on the same machine and allows communication. It works out-of-the -box without any specific corbos Adaptive Autosar Platform settings.

- Start the Corbos Platform runtime using `sudo adg-prepare && adg-start`
- In the `hello-com` directory execute `sudo adg_sandbox -e -c adg_com_svc_process_am.json`
- In the `hello-com` directory execute `sudo adg_sandbox -e -c adg_com_client_process_am.json`
- Press `enter` in the standard out of the `client` to trigger the function call
- Start the `dlc-viewer` and filter the logs based on application id `HLCC` and `HLCS`

SomeIP execution

For details on how to set up the SomeIP communication please refer to the [Annex 5 - Using sil-kit for an overlay network](#) section.

Sample application for the DM module

This application demonstrates the usage of the Diagnostic Management functional cluster of corbos Adaptive Autosar platform. The DM module implements the Adaptive Autosar `ara::diag` interface.

Overview

The application uses the DM module client library (in addition to the DLT ones): `libdiagapi` to offer a Data Identifier using the `ReadDataByIdentifier` service through the DM daemon (running in the corbos Adaptive Autosar runtime) and logs several messages to a Genivi DLT compatible backend.

The application registers itself to the DM daemon as the provider of the value of the Data Identifier `1000`. The value returned is constant `10`.

WARNING



The below description is just a short and incomplete summary. For a rough overview please read the Explanation of Adaptive Autosar Platform (doc no 706) and Specification of Diagnostics (doc no 723) of the Adaptive Autosar standard. Additionally you might find useful to take a look at the following standards: UDS (ISO 14229-1:2013) and DoIP (ISO 13400-2:2012).

The model part contains the default configuration for Log and Trace (see hello-dlt example) for the executable, an interface definition for the Data Identifier `1000` and the corresponding Diagnostic Management(including DoIP) configuration.

The interface definition requires at minimum:

- a `DiagnosticDataIndentifierInterface` element which represents the C++ class name of the interface
- a `DiagnosticDataIndentifier` element which defines the structure of the offered Data Identifier
- a `DiagnosticServiceDataIdentifierPortMapping` which connects the service interface to the actual process
- one or more `Symbol Props` element(s) that represent the namespace under which the sources are generated
- one corresponding `Read | ClientServerOperation` element along with ``ArgumentDataPrototype`s` that describe the actual method offered by the interface
- one `DataType` that is referenced in the method signature and depicts the type returned by the Data Identifier
- one `AdaptiveApplicationSwComponentType` that contains one `PPortPrototype`, one `PortGroup` and one `AdaptiveSwcInternalBehavior`
- the `PPortPrototype` points to the `DiagnosticDataIndentifierInterface`; the `PortGroup` points to the `DiagnosticDataIndentifier` and the `AdaptiveSwcInternalBehavior` points to the `PortGroup`

CAUTION



The above interface defines a simple `ReadDataByIdentifier` scenario. This is similar to the COM definition example.

NOTE



The deployment configuration for the application along with the DoIP configuration would require an extensive description that does not fit the purpose of this small Readme.

The following points describe the *truly important* aspects of the DM and DoIP configuration:

- the `DiagnosticServiceTable` depicts which Data Identifiers are offered and by what application
- the `SoftwareCluster` depicts the diagnostic address of the DM server (this is needed by the client)
- the `EthernetCluster` describes the DoIP server unicast network binding properties and the type of diagnostic entity
- the `PlatformModuleEthernetEndpointConfiguration` describes the DoIP server multicast network binding properties
- the `DiagnosticConnection` and `DoIpTpConfig` describe the diagnostic address routing logic (between the clients and the software cluster)
- the custom `DiagnosticManagement` module instantiation depicts in general the non-application specific configuration of the DoIP server

Structure

There are four important parts:

- The `src` folder contains the actual source code of the application
- The `model` folder contains the ARXML configuration necessary to generate the stubs for the DM interface, the DM configuration and the DLT logging configuration
- The `run_generators.sh` script that handles all the generator interaction
- The `CMakeLists.txt` defines the required C++ libraries of corbos AdaptiveCore and a custom task for DLT and DM

NOTE



The custom task for DLT & DM executes a shell script against the model directory and generates a stub(skeleton) of the DID to be implemented by the diagnostic application. Additionally it also creates actual configuration files used by the DLT and DM libraries linked in the executable.

CAUTION



Since the configuration generation is a lengthy process, the custom task is executed when the build directory is clean or when any of the ARXML model files change.

Building and running

Generate the build & libraries configuration using CMake (preferably through an IDE like VScode) and then compile the sources.

- Copy the `build/ara_DM_mm.json` file to `/etc/adaptive/ara_DM/`
 - Start the corbos Adaptive Autosar Platform runtime using `sudo adg-prepare && sudo adg-start`
 - In the `hello-dm` directory execute `sudo adg_sandbox -e -c adg_dm_process_am.` (Press CTRL + C to exit)
 - Start the `dlt-viewer` and filter the logs based on application id `HLDM`
-

Open Wireshark and sniff the Dolp traffic on localhost (using the ssh port-forwarding facilities)

- Execute the `test_doip.sh` script and see the exchanged messages

Sample application for AI integration

The application demonstrates AI integration with functional clusters of corbos Adaptive Autosar platform.

Overview

The application uses the ONNXRuntime library to run inference on a given neural network model and logs the post-processed outputs as messages to a Genivi DLT compatible backend. The neural network model is a HMI predictor, its scope being predicting whether the A/C or the heating shall be turned on inside the car cabin based on data from sensors describing the cabin temperature, the outside temperature and the outside humidity. The architecture of this feedforward neural network consists in a shared hidden layer followed by two parallel softmax output heads, solving a multi-head classification problem.

The flow of the application is as follows:

- Sensor data (cabin temperature, outside temperature, outside humidity) is generated by applying sinusoidal functions in order for the values to simulate a 24 hour cycle
- Values are then split into time sequences which are fed to the neural network
- The outputs are confidence scores which are compared to thresholds for both cold and hot condiditons. If the value for one output is bigger than its corresponding threshold, then the respective command is suggested (either A/C or heating on)

Structure

There are four important parts:

- The `src` folder contains the actual source code of the application
- The `model` folder contains the ARXML configuration of the DLT libraries used by the application
- The `CMakeLists.txt` defines the required C++ libraries of corbos Adaptive Autosar platform as well as the ONNXRuntime library required for inference and a custom function
- The `onnx-model` folder contains the neural network model serialized in `onnx` format; it can be placed at a path chosen without restrictions by the user, but mentioned path shall be given as a command line argument when running the application

Building and running

Generate the build & libraries configuration using CMake (preferably through an IDE like VScode) and then compile the sources.

- Start the corbos Adaptive Autosar platform runtime using `sudo adg-prepare && sudo adg-start`
- In the `hello-ai` directory execute `sudo adg_sandbox -e -c adg_hello-ai_process_am.json`
- To change the input argument of the application update the `adg_hello-ai_process_am.json` file at line 18 in the `args` section
- <Optional> Start `dlt-viewer` (requires a GUI) and filter the logs based on application id `HLD`

FAQ

How is the support for different AUTOSAR versions aligned?

Elektrobit offers various versions of this AMI covering different releases of the *corbos AdaptiveCore*. Currently the supported Autosar versions are 19-03 and 20-11.

Can I debug the sample applications ?

Yes. You need to install the C/C++ extension for VSCode in the remote environment.

How do I connect with the DLT viewer?

You will need to forward port 3490 from the remote environment to your local machine either using a tool like Putty or directly from VSCode.

How can I sniff traffic with Wireshark?

You can use SSH remote capture extension of Wireshark to capture traffic in the remote environment. (see [Setting up Wireshark to monitor the overlay network](#) for example)

If I want to change the model files, what options do I have?

The current workflow assumes you will provide your own model files corresponding to your application needs. These can be exported from third party tools (see [Annex 1 - Mathworks Simulink modelling](#) for example) or from Elektrobit corbos Studio. Furthermore, some configurations can be created or edited directly in Visual Studio Code (see [Annex 2 - JSON deployment manifests without ARXML](#)).

How can I get my source-controlled (e.g. Git) applications into the remote environment?

Depending on your AWS account setup you can allow the EC2 instance to access various source control systems. The EC2 comes with git pre-installed so you can simply clone your repos inside.

What generators are supported?

Currently this environment allows the integration with `ara::dm`, `ara::com`, `ara::em`, `ara::pm`, `ara::iam`, `ara::ucm` and `ara::dlc` modules and only supports their corresponding generators.

How do I start a project from scratch?

The AMI provides some quickstart templates that can be used as a basis for new projects. The choice of build environment technology (make, Cmake, conan etc.) is entirely up to the developer.

Annex 1 - Mathworks Simulink modelling

This annex depicts the modelling process of the sample application for the COM module in Mathworks Simulink. It is by no means an exhaustive depiction and just covers the essentials.

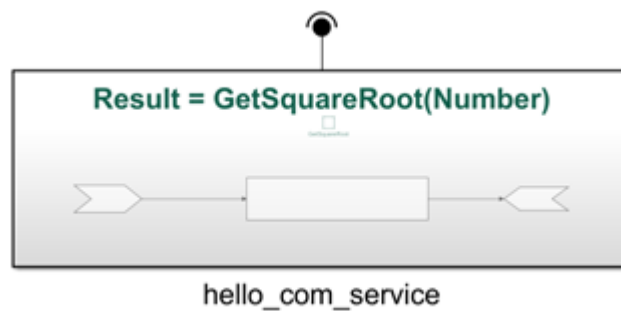
The `HelloAdgService` and the `HelloAdgClient` components were modeled in Mathworks Simulink as block Models.

The model block references the specified model. It displays input and output ports that correspond to the top-level input and output ports of the referenced model.

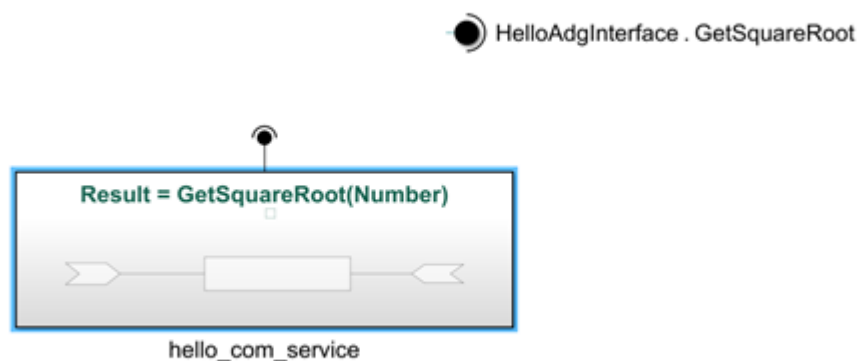
Modelling the interface

The `GetSquareRoot()` method provided by the `HelloAdgService` can be modeled as a Mathworks Simulink Function block.

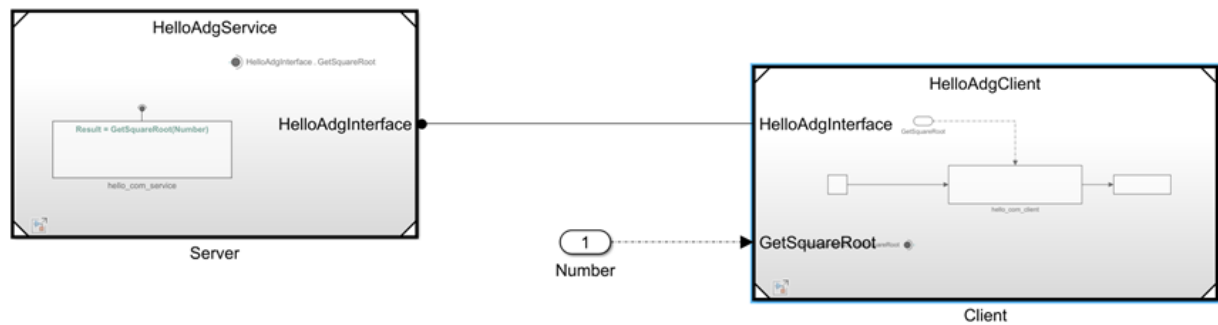
This is a subsystem block pre-configured as a starting point for graphically defining a function with Mathworks Simulink. It provides a text interface to function callers. A Function block can be called from a Function Caller block.



The Function block `HelloAdgInterface` allows the Mathworks Simulink function `GetSquareRoot()` in the referenced model `HelloAdgService` to be called by a Function Caller in another referenced model.

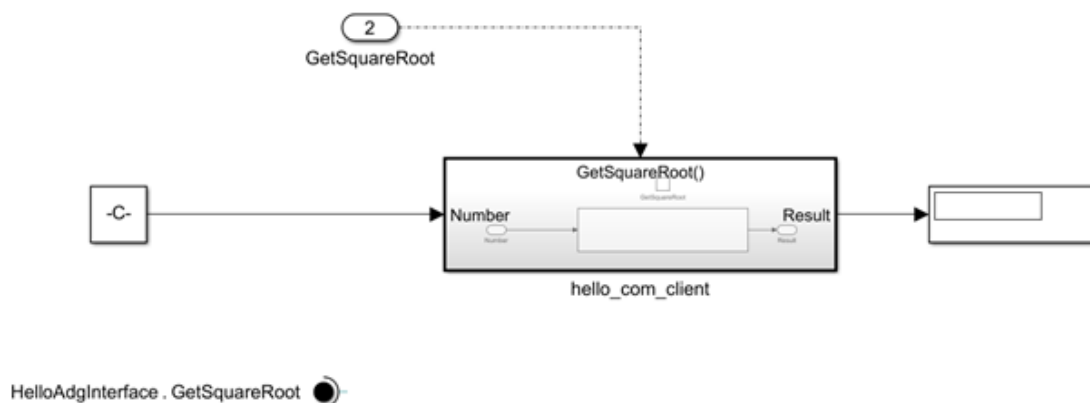


The Function Element block, when placed at the root level of a model referenced by a Model block, creates an exporting function port in the Model block.

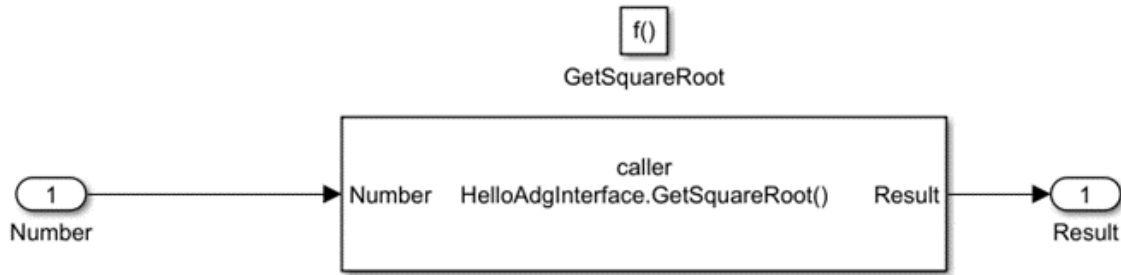


Modelling the connection

The Function Element Call block allows a Function Caller block in a referenced model to call a Mathworks Simulink function in another referenced model i.e. **HelloAdgClient**. The name of the Function Element Call block must be same as the Function Element block in **HelloAdgService**.



When the exporting function port is connected to an invoking function port of another Model block i.e. **HelloAdgService**, a function caller in **HelloAdgClient** can issue **GetSquareRoot()** and receive return values namely **Result** through the respective function ports of the Model blocks **HelloAdgClient**.



Using the generated ARXML files

IMPORTANT

The code-generation facilities of Mathworks Simulink will export multiple ARXML files along with various pieces of source code. For the purpose of this sample application only the generated `HelloAdgService_interface.arxml`, `HelloAdgService_datatype.arxml` are used.

Once the ARXML files have been exported through the Mathworks Simulink Coder extension SDK generators service can be used:

- copy the two files (`HelloAdgService_interface.arxml`, `HelloAdgService_datatype.arxml`) into a `model` folder under a preferred path e.g. `/home/ubuntu/samples/com-simulink`
- copy one of the `config.json` files from the sample applications next to the `model` folder and adjust it:

```
{
  "folder" : "/home/ubuntu/samples/com-simulink/model",
  "plugets": "AraComBindingGenerator",
  "pname" : "com-simulink",
  "token" : "<your_token_here>",
  "url": "https://adg-generator-prod.elektrobit.cloud",
  "downloadPath" : "/home/ubuntu/samples/com-simulink",
  "asr": "20-11"
}
```

- execute in the `/home/ubuntu/samples/com-simulink` folder `api_studio_client_cli-linux -c /home/ubuntu/samples/com-simulink/config.json`
 - extract the contents of the archive to obtain the source stubs for the `ara::com` C++ language bindings (e.g. under `src/` and `inc/`)
 - these source files can then be used in your own build environment and your own custom application
-

WARNING



Currently Mathworks Simulink ARXML generation does not allow complete generation of runtime configurations (for example SOMEIP deployment manifests). For prototyping purposes you can re-use the ones generated in the `samples/hello-com` application and adapt the human readable JSON files to your needs.

Annex 2 - JSON deployment manifests without ARXML

This annex explains the process of creating/updating your JSON deployment manifests for DLT and DM bypassing ARXML configuration. In some cases this is useful for debugging and testing various configuration options for your application faster during integration.

NOTE



ARXML is still needed for language binding purposes. E.g. in case of DM and COM you still need to model your interface and run through the generators.

The `schemas` directory contains two JSON schemas:

- `dlt-schema.json` is used to validate the JSON deployment manifest for the DLT configuration of your application
- `dm-schema.json` is used to validate the JSON deployment manifest for DM configuration of your application

The `.samples.code-workspace` file already references these two schemas for validating the corresponding JSON manifest files. You can then edit or create these JSON manifest files (with the corresponding naming conventions) directly from the Visual Studio Code.

Annex 3 - Using the generators client

This annex explains how to directly use the code and configuration generator client. This allows the developer to use any build system as per their preference.

The `api_client_cli-linux` is bundled in the AMI and offers two ways of interaction:

- passing multiple cli arguments as input (pass `--help` flag to see the options)
- passing a configuration file as input (pass the `-c /absolute/path/to/config` option)

WARNING



Due to some limitations in the client, currently only absolute paths are supported when passing the configuration file.

The sample applications in the AMI use the second approach. For example:

```
ubuntu@:~$ api_client_cli-linux -c /home/ubuntu/samples/hello-com/config.json
```

This will generate the code and configuration files for the `hello-com` sample application. In this case you'll find the generated files in an archive named `hello-com_<some_timestamp>.zip` in the same folder where the client was executed.

The configuration file is structured as follows:

```
{
  "folder" : "/home/ubuntu/samples/hello-com/model", ❶
  "plugets": "AraDltModelGenerator,AraComBindingGenerator,
             AraComManifestGenerator", ❷
  "pname"  : "hello-com", ❸
  "token"   : "<your_token_here>", ❹
  "url"     : "https://adg-generator-prod.elektrobit.cloud", ❺
  "downloadPath" : "/home/ubuntu/samples/hello-com", ❻
  "asr"     : "20-11" ❼
}
```

1. the path to the model folder that contains the ARXML files
2. the generator(s) to use
3. name of your project
4. the authentication token
5. the url of code and configuration generator service
6. the path where the generated files will be downloaded
7. the Autosar version used

Annex 4 - Deploying to targets

This annex explains how to deploy an ADG SDK application to one of these three targets:

- virtual ECU as AWS EC2 arm64 instance - called Graviton
- virtual ECU as Arm Virtual Hardware (Raspberry Pi 4) - called AVH in the following
- physical ECU as Raspberry Pi 4 - called RasPi in the following

Prepare the targets

Graviton - Launch an EC2 arm64 instance with Ubuntu Server 22.04 (by default running applications on the SDK instance is equivalent to this)

AVH - First register to [Arm Virtual Hardware](#) and then create a Raspberry Pi 4 device with Ubuntu Server on RPi (22.04.1) as firmware.

RasPi - Install Ubuntu Server 22.04 as described here: [Install Ubuntu on a Raspberry Pi](#)

Install the corbos AdaptiveCore runtime

First, connect via SSH to the target(s). Next, install the corbos AdaptiveCore runtime environment from an installer package with following commands in your target shell:

```
ubuntu@:~$ cd /tmp
ubuntu@:~$ curl -L -H "Authorization: Bearer <your_token_here>" "https://adg-generator-
prod.elektrobit.cloud/runtime?adgv=2.18&platform=ubuntu&target=raspi" -o adg-2.18-
runtime.tar.gz
ubuntu@:~$ tar xzvf adg-2.18-runtime.tar.gz
ubuntu@:~$ cd adg
ubuntu@:~$ sudo chmod +x ./install-adg.sh
ubuntu@:~$ sudo ./install-adg.sh
```

IMPORTANT

Replace `<your_token_here>` with your authentication token as described in [Configuring the API key for the generators](#).

At the end you should be able start the runtime with the following commands:

```
ubuntu@:~$ sudo adg-prepare
ubuntu@:~$ sudo adg-start
```

You should see the following output:

```
Starting EB Corbos AdaptiveCore...
Starting adg_emd in detached mode
Checking adg_emd is running..
Wait 1...
adg_emd PID: 41309
```

Deploy the application to the target

You can now copy any of the sample applications to the target.

First, create a folder structure to store the application binary on the target. Next, copy your application binary to the target.

Option 1: Copy via command line

Use the `scp` command to copy your application binary via the command line. For example for `hello-com`, use this from a command line on your target:

```
ubuntu@:~$ ssh root@<target_ip> "mkdir -p /root/hello-com"
ubuntu@:~$ scp -r /home/ubuntu/samples/hello-com/build/hello_com_client
root@<target_ip>:/root/hello-com/
ubuntu@:~$ scp -r /home/ubuntu/samples/hello-com/build/hello_com_server
root@<target_ip>:/root/hello-com/
ubuntu@:~$ scp /home/ubuntu/samples/hello-com/build/libsvcifc.so root@<target_ip>:/usr/lib/
```



For applications that come with a shared library make sure to copy this to `/usr/lib` (e.g. `hello-dm` and `hello-com`).

NOTE



The `hello-com` and `hello-dm` sample applications contain also configuration files needed by some daemons part of the corbos AdaptiveCore platform. Make sure to copy them on the target in their required locations. For details see the Building and Running section of the respective sample application.

Option 2: Copy via Visual Studio Code extension

As an alternative to the command line, you can also use a custom Visual Studio Code extension for copying the application binary to the target.

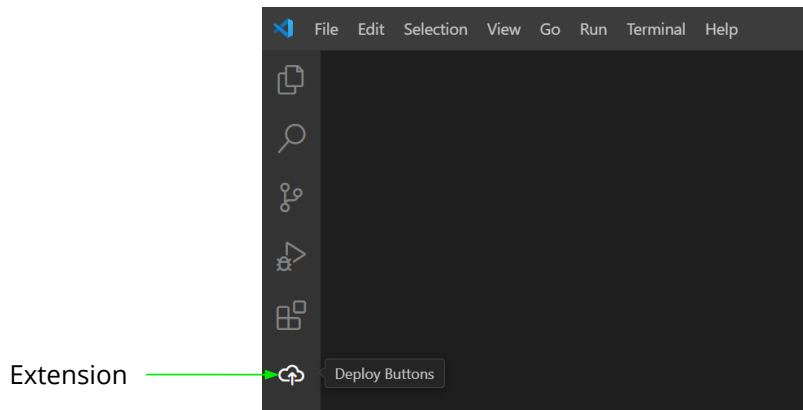
Install extension

You first need to install the extension on your AWS instance. Log in to your AWS instance via SSH in Visual Studio Code. Once you are connected, open a Visual Studio code terminal. You can then install the extension with this command:

```
code --install-extension /home/ubuntu/elektrobit/extensions/deploy-buttons-0.1.0.vsix
```

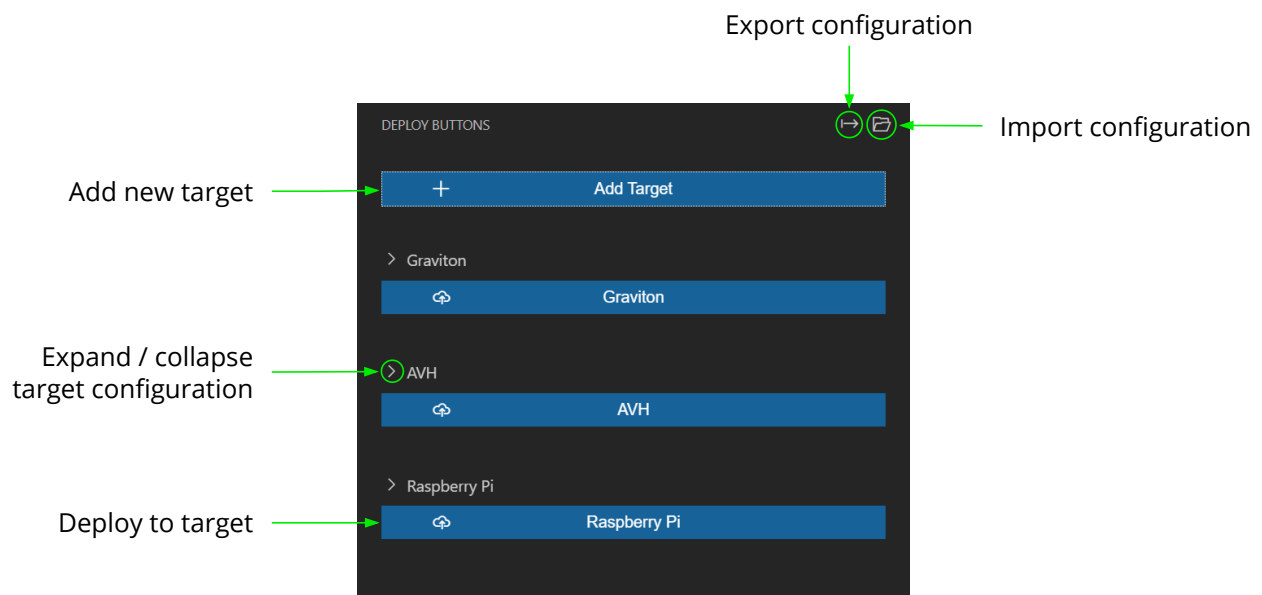

Open extension

In Visual Studio Code, open the "Deploy Buttons" extension with the cloud upload icon in the sidebar:

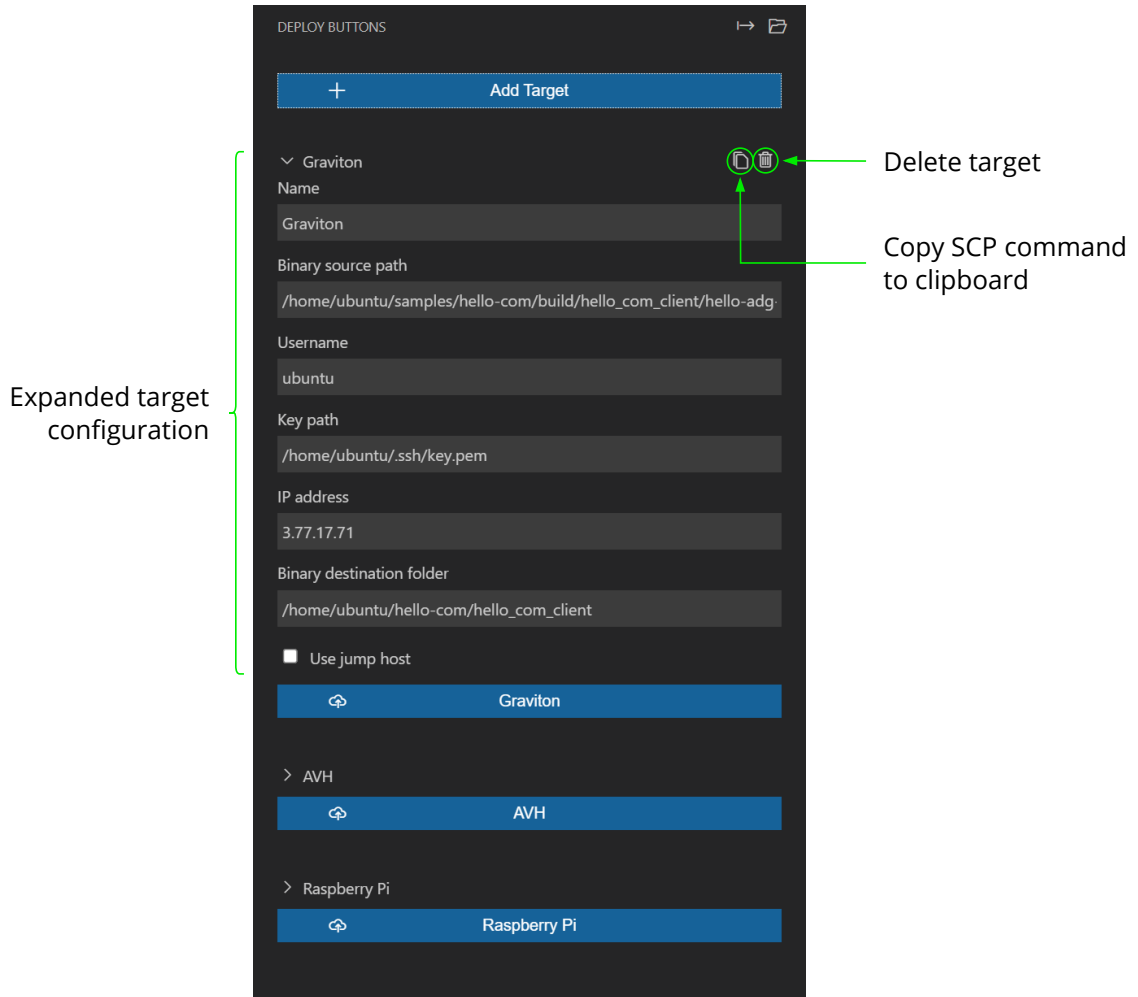


Use extension

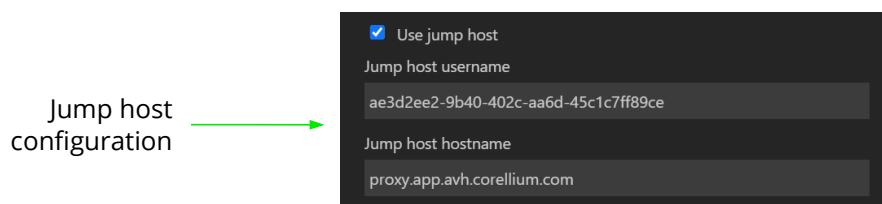
The extension allows you to dynamically add and configure targets. You can then deploy to one of the targets with the respective deploy button. You can also export and import the entire configuration. Here is an overview over the basic functions:



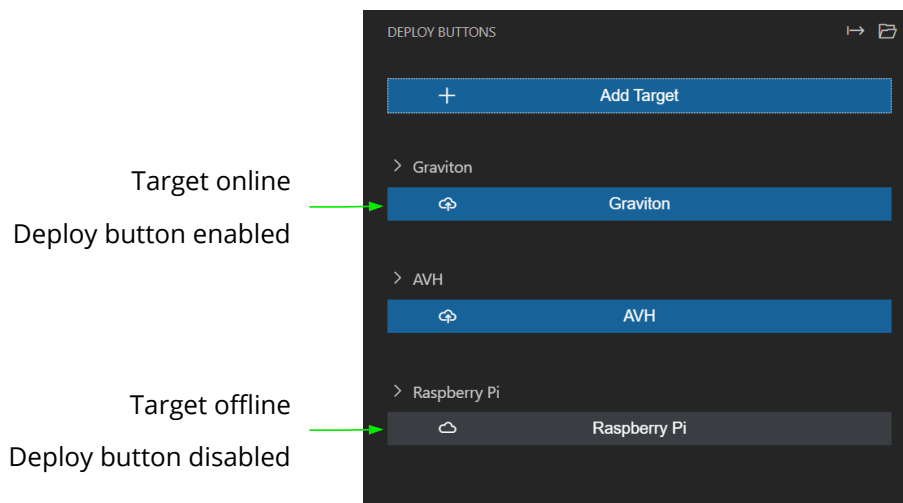
The deploy buttons copy a file to a network target by running an `scp` command in the background. You can configure the parameters of the `scp` command separately for each target. Use the chevron icon to expand and collapse the target configuration. The expanded view also includes buttons for copying the `scp` command to the clipboard and for deleting a target. Here is an example of an expanded target configuration:



The optional configuration of a jump host appears only when you enable the option `Use jump host`. For **AVH**, you need to enable the option `Use jump host` and configure the username and hostname of the jump host according to the details in the `Connect` tab of your AVH device.



The extension periodically checks if the targets are online. When a target is offline, its deploy button gets disabled. Targets that use a jump host are being treated as always online.

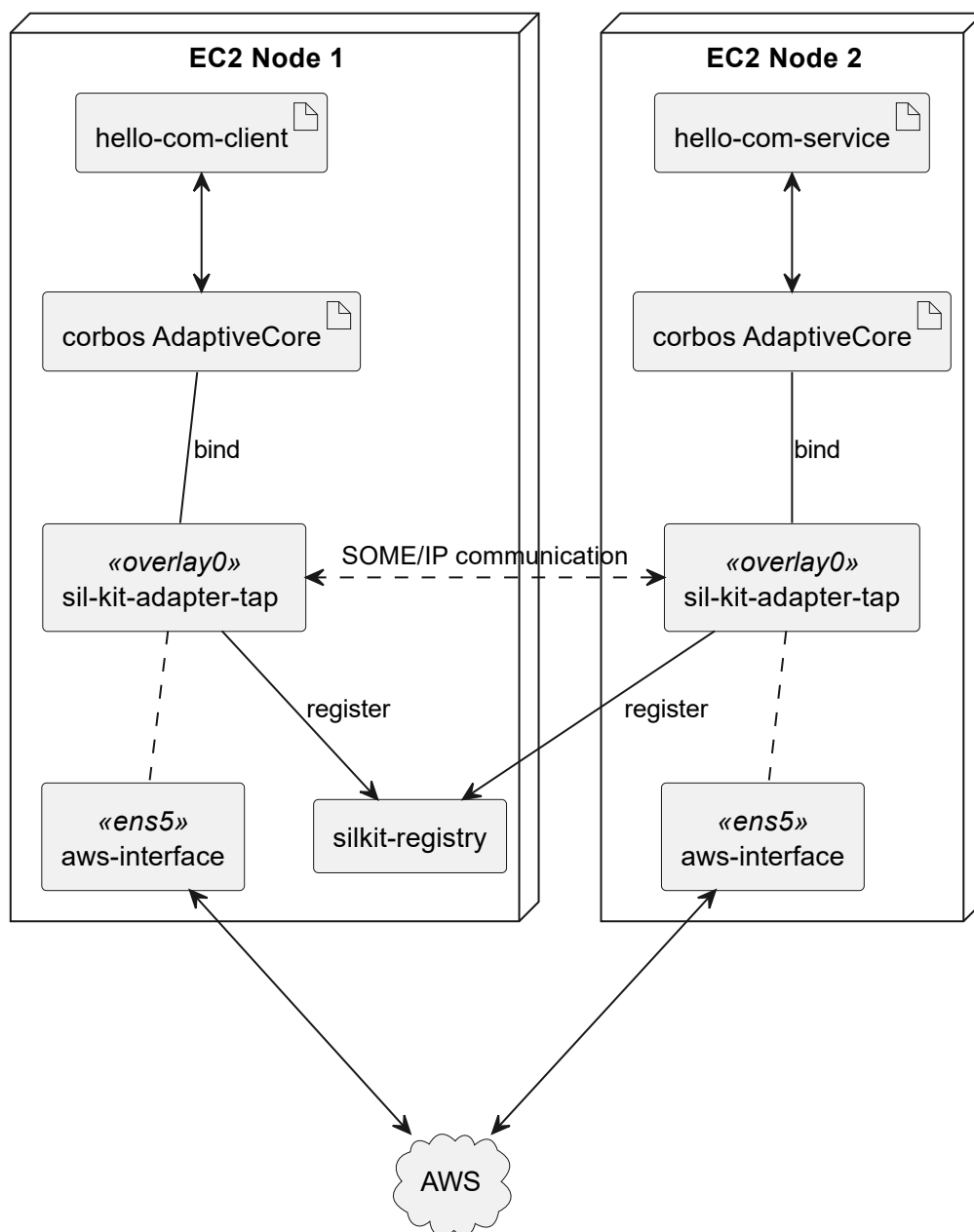


Annex 5 - Using sil-kit for an overlay network

This annex explains how to set-up an overlay network using [sil-kit](#). This tool comes pre-bundled in the *SDK for EB corbos AdaptiveCore* AMI. The overlay network is a virtual network that runs on top of the physical network.

Such a network is useful for testing automotive applications distributed across multiple ECUs. Additionally, developers can use locally proprietary test tools to interact with virtual ECUs in this overlay network (e.g. inject traffic, connect local emulators or even real targets).

The following example setup demonstrates how to create a virtual network with two ECUs, ECU1 and ECU2, each running on a separate EC2 instance based on the *SDK for corbos AdaptiveCore* AMI. The sample application for COM module is used to demonstrate SOME/IP communication between the two ECUs using the overlay network.



Prerequisites

- two running EC2 instances based on the SDK for EB corbos AdaptiveCore AMI (available in AWS marketplace in arm64 flavor)
- the ip addresses of the two EC2 instances are known
- the security group of the two EC2 instances allows all traffic between them (this is required for sil-kit).

Creating the tap interfaces on the EC2 instances

On each of the two instances execute the following commands to create the tap interfaces:

```
ubuntu@:~$ sudo ip tuntap add mode tap overlay0
ubuntu@:~$ sudo ip link set overlay0 up
```

Next, for the first instance assign the corresponding ip where hello_com_client will run :

```
ubuntu@:~$ sudo ip addr add 172.18.0.2 dev overlay0
```

For the second instance assign the corresponding ip where hello_com_server will run:

```
ubuntu@:~$ sudo ip addr add 172.18.0.3 dev overlay0
```

These will be the actual interfaces the virtual ECUs will use to communicate with each other.

Setting up the overlay network

One of the instances will act as the registry where other instances can register themselves. The other instances will act as taps that can communicate with each other through the registry.

Since this setup only uses two EC2 machines, one instance will run both the registry and the tap.



TIP

Before starting the sil-kit-registry, ensure that the tap interface is up and running on both instances. Additionally make sure to clean up any previous instances of the sil-kit-registry and sil-kit-adapter-tap.

On the first instance, run the following commands to start the sil-kit registry:

```
ubuntu@:~$ sudo killall -9 sil-kit-registry || true
ubuntu@:~$ sudo sil-kit-registry --listen-uri silkit://0.0.0.0:8501 -s > /tmp/sil-kit-registry.log 2>&1 &
```

Next, setup a dummy fifo on each instance to keep the tap running:

```
ubuntu@:~$ sudo killall -9 sil-kit-adapter-tap || true
ubuntu@:~$ sudo rm -rf /tmp/silkit_dummy_fifo0
ubuntu@:~$ sudo mkfifo /tmp/silkit_dummy_fifo0
```

On the first instance, run the following command to start the sil-kit-adapter-tap:

```
ubuntu@:~$ nohup sh -c cat /tmp/silkit_dummy_fifo0 | sudo sil-kit-adapter-tap --log Info --
registry-uri silkit://<ip_address_of_the_registry_instance>:8501 --name ECU1 --network
tap_bridge --tap-name overlay0 > /tmp/SilKitAdapterTap.log 2>&1 &
```

On the second instance, run the following command to start the sil-kit-adapter-tap:

```
ubuntu@:~$ nohup sh -c cat /tmp/silkit_dummy_fifo0 | sudo sil-kit-adapter-tap --log Info --
registry-uri silkit://<ip_address_of_the_registry_instance>:8501 --name ECU2 --network
tap_bridge --tap-name overlay0 > /tmp/SilKitAdapterTap.log 2>&1 &
```

NOTE



Replace `<ip_address_of_the_registry_instance>` with the actual ip address of the first instance(where the registry runs).

TIP



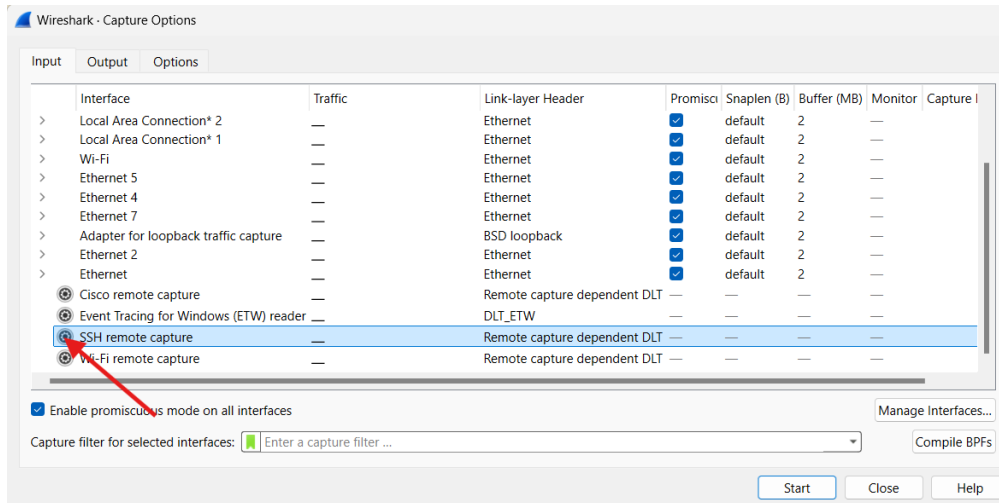
You can check the logs of the sil-kit-registry and sil-kit-adapter-tap by looking at the respective log files in `/tmp`.

Next, update the startup script of the COM daemon to use the tap interface on both instances. Modify the startup script at `/etc/adaptive/adg-platform/adg_com_start-startup.sh` :

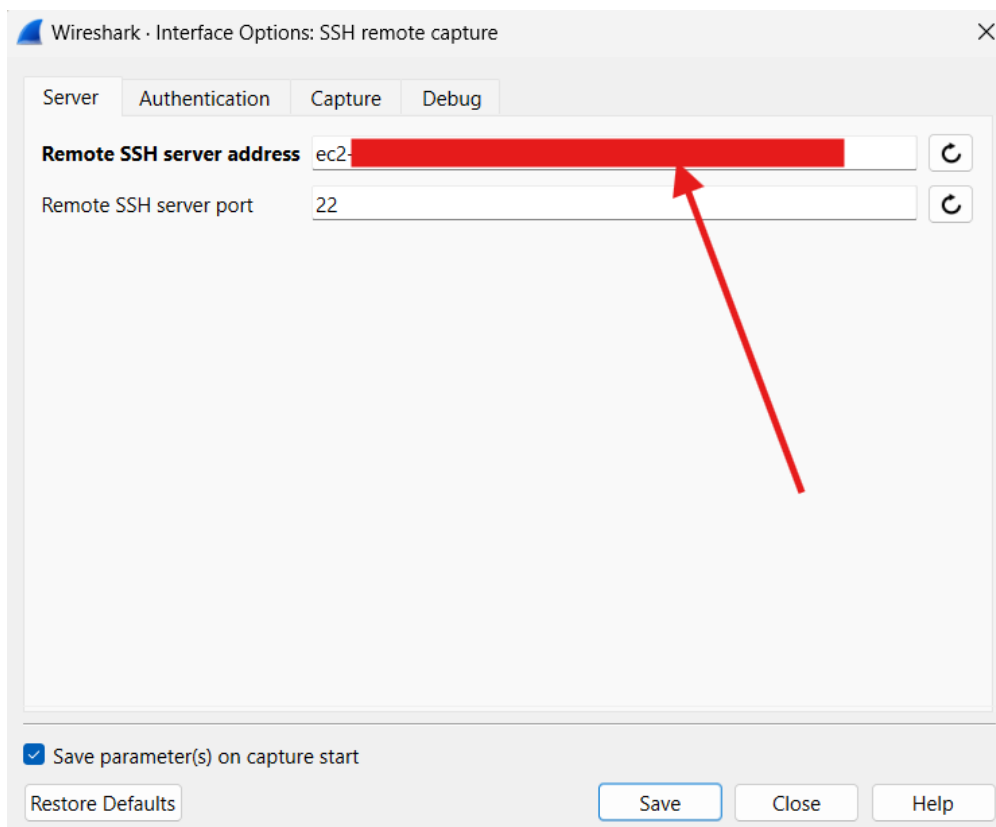
```
#!/bin/bash
interface="overlay0" <-- change this to the tap interface name
...
```

Setting up Wireshark to monitor the overlay network

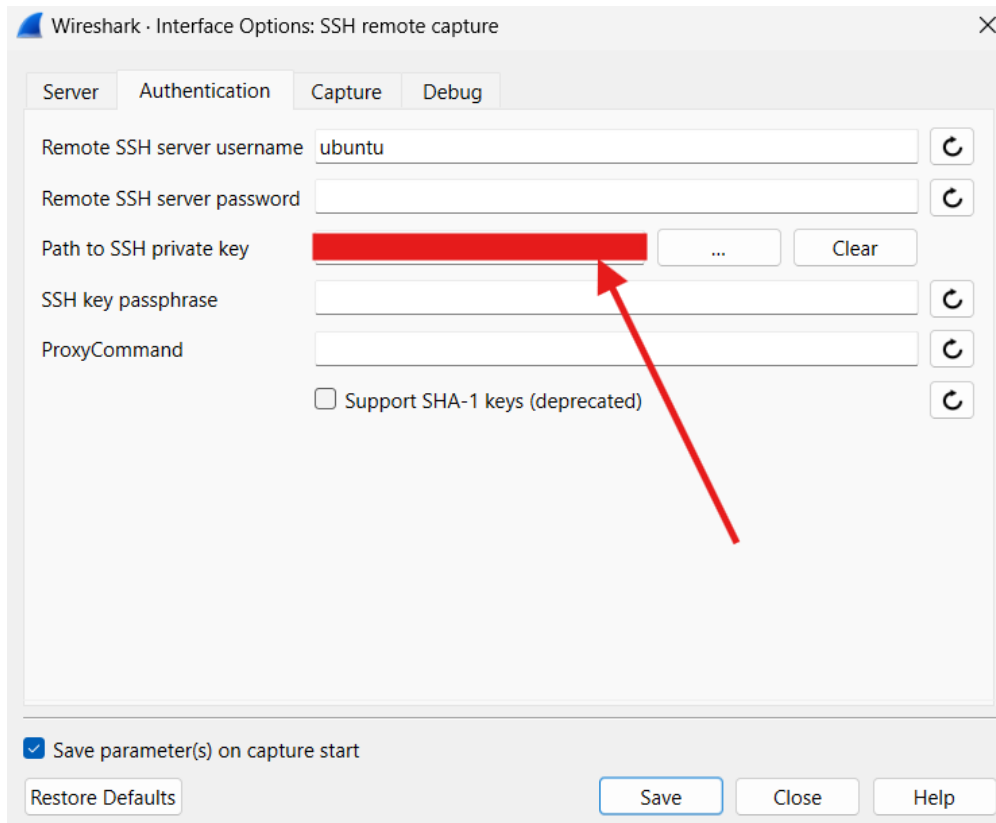
To monitor the overlay network, install Wireshark on your local machine and configure the ssh capture interface to capture packets from the tap interface on the first instance.



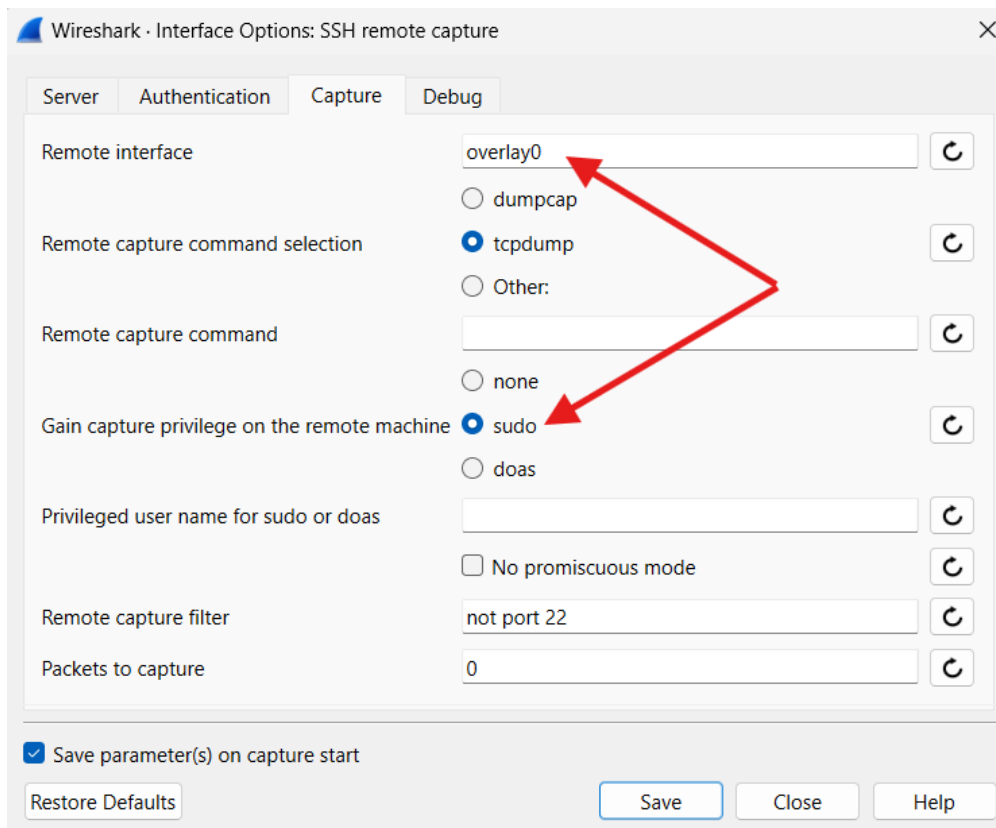
First configure the instance on which to sniff the packets:



Next configure the authentication to the instance:



Finally, configure the interface to sniff on and start the capture:



Running the sample COM application

On the second instance, copy the application manifest and run the following commands to start the COM server:

```
ubuntu@:~$ sudo mkdir /etc/adaptive/ara_Com/daemon_1
ubuntu@:~$ sudo cp /home/ubuntu/samples/hello-com/build/hello_com_service/hello-
com_CM_ServiceMachineIPv4_ServiceEthConnector_amd.json /etc/adaptive/ara_Com/daemon_1
ubuntu@:~$ sudo adg-prepare
ubuntu@:~$ sudo adg-start
ubuntu@:~$ sudo adg_sandbox -e -c adg_com_svc_process_am.json
```

On the first instance, copy the application manifest and run the following commands to start the COM client:

```
ubuntu@:~$ sudo mkdir /etc/adaptive/ara_Com/daemon_1
ubuntu@:~$ sudo cp /home/ubuntu/samples/hello-com/build/hello_com_client/hello-
com_CM_ClientMachineIPv4_ClientEthConnector_amd.json /etc/adaptive/ara_Com/daemon_1
ubuntu@:~$ sudo adg-prepare
ubuntu@:~$ sudo adg-start
ubuntu@:~$ sudo adg_sandbox -e -c adg_com_client_process_am.json
```