



Implement embedded continuous integration with Docker and Jenkins

Version 1.0

Non-Confidential

Copyright © 2021 Arm Limited (or its affiliates).
All rights reserved.

Issue 02

102085_0100_02_en



Implement embedded continuous integration with Docker and Jenkins

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	24 February 2021	Non-Confidential	Initial release

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview.....	6
2. Introduction to Docker.....	7
3. Install Docker.....	8
4. Create and verify a Docker image.....	9
5. Introduction to Jenkins.....	14
6. Create pipeline.....	19
7. Set up simulation in Docker.....	20
8. Set up simulation in host.....	22
9. Run pipeline.....	24
10. Related information.....	26
11. Next steps.....	27

1. Overview

This guide describes how to use Jenkins and Docker in a continuous integration development flow. The audience for this guide is embedded software developers. In the guide, we also address the topic of testing platforms, by highlighting the capabilities of virtual hardware models using Arm Fast Model technology. Using a continuous integration methodology helps to minimize problems during software development, for example large merge conflicts, duplicated effort, and non-reproducible bugs.

Continuous integration practices with Jenkins, Docker containers, and Arm Fast Models provide a consistent and automated foundation for your embedded software development work.

We will work through the following topics in the guide:

- Running a “hello world” application on a virtual Arm Cortex-M4 system in a custom Docker container.
- Implementing a test to verify the completion of the app.
- Configuring Jenkins to automate the test flow that we developed.

Before you begin

We assume that you have a basic knowledge of embedded software development on Arm. Docker, Jenkins, and Arm Fast Models will be explained in the guide.

A high-level understanding of Python is helpful but is not required.

You need to have the following items installed to work through the examples in this guide:

- An evaluation license for Arm Fast Models. You can obtain a free 30-day license by emailing license.support@arm.com and specifying that you want to use a Cortex-M4 CPU to follow this guide.
- A [zip file](#) with the code that you will need to replicate the example in this guide.

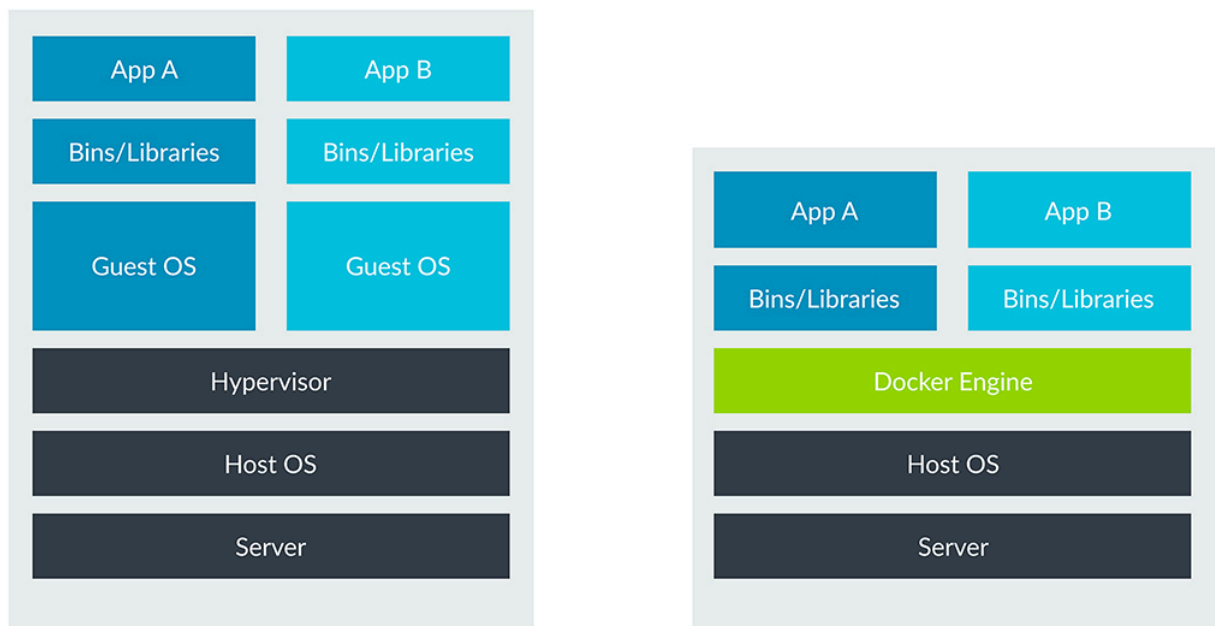
2. Introduction to Docker

Docker is a mechanism that isolates the dependencies for applications, or tests, by packing them into containers. Each container runs an image that is designed to run a software app or apps, including all necessary libraries and dependencies. You can create images from scratch, or download them free and edit them to fit your needs. Docker containers are portable, and run on Mac, Linux, and Windows. Docker allows members of a team to ensure that they are working in the same environment.

Docker operates like a Virtual Machine (VM). However, instead of each container having its own OS, resources are shared among Docker containers, as you can see in the following image. This means that applications can be packaged with only what they need to run. Containers are more lightweight, portable, and reusable than VMs:

Figure 2-1: Continuous Intergration Guide VM vs Docker block diagram

Virtual Machines and Docker comparison



3. Install Docker

You can install Docker in several forms. Which form you use depends on your end goals and your host OS. For the example in this guide, Docker Community Edition is the best option. Windows and Mac OS users need to install the Community Edition of Docker Desktop. Linux users will download the Docker engine.

To install Docker, follow these steps for your operating system:

Windows



Docker for Windows requires Microsoft Hyper-V to run. The Docker installer will handle that enablement. However, when Microsoft Hyper-V is enabled, VirtualBox will no longer work. If you need to use VMs, install Docker inside your VM. Virtualization, which is different to Hyper-V, must be enabled. You can check this by going to the Performance tab on the Windows Task Manager.

1. Navigate to the Docker [installation page](#) and select the Stable build, which will download the Docker for Windows Installer executable.
2. Run the Docker installer and follow the on-screen instructions.
3. Start Docker by searching for Docker for Windows in the Windows search bar. When Docker opens for the first time, you will see a hello message and a link to documentation. On subsequent resets, Docker will start by itself.

Mac OS

1. Navigate to the Docker [installation page](#) and select the Stable build, which will download the Docker for Mac Installer executable.
2. Run the Docker installer and follow the on-screen instructions.
3. Start Docker by searching for Docker for Mac in the Mac Finder. When Docker opens for the first time, you will see a hello message and a link to documentation. On subsequent resets, Docker will start by itself.

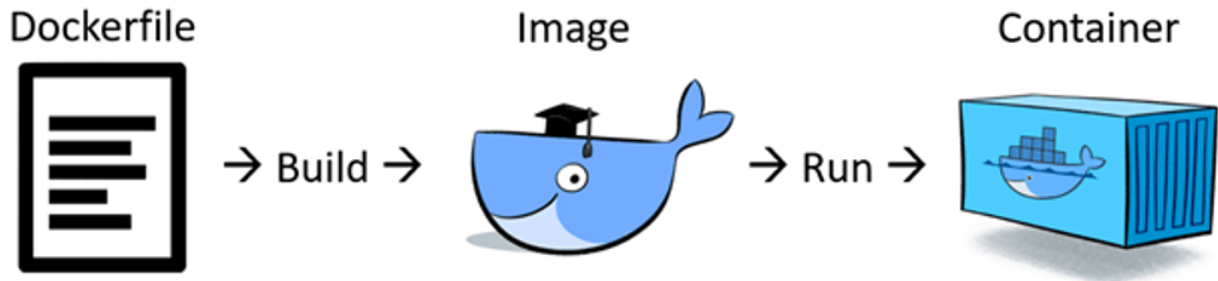
Linux

1. Navigate to the Docker [installation page](#).
2. Follow the install instructions specific to your OS type.
3. On subsequent resets, Docker will start by itself.

4. Create and verify a Docker image

To run an application in a container, you must create an image that contains the required software dependencies. An image is created by writing and then building a Dockerfile. The following image shows the steps in this process:

Figure 4-1: Dockerfile flow



In this guide, we will provide the Dockerfile that is used to create the image used in this example, and we will walk through each step. Be sure to place the downloaded Fast Model package .zip file next to this Dockerfile for it to build properly.

Let's take a look at the Dockerfile that we will use in our example.

Every Dockerfile must start with a `FROM` command, specifying what the image is built from. In our example, the `FROM` command is a custom image, with Arm tools pre-installed on a minimalist Ubuntu 16.04 image, that is pulled from Docker Hub. Additional packages are installed to support Arm Fast Models and Python scripting. You can see this in the following code:

```

FROM ubuntu:16.04

# Needed test packages so newer Fast Model versions work (11.10 and up)
RUN apt-get update -y
RUN apt-get install -y software-properties-common
RUN add-apt-repository -y ppa:ubuntu-toolchain-r/test
RUN apt-get update -y
RUN apt-get install -y gcc-7 g++-7
RUN update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-7 80 --slave /usr/
bin/g++ g++ /usr/bin/g++-7

# Install packages
RUN apt-get update && apt-get install -y apt-utils
RUN apt-get install -y \
    #FMs
    lsb-core \
    libxext6 \
    libsm6 \
    libxcursor1 \
    libxft2 \
    libxrandr2 \
    libxt6 \
    libxinerama1 \
    #Python
    python2.7 \
    python-pip
  
```

To ensure image security, we create a new user with limited privileges. The group name and username are `jenkins`, as shown here:

```
# Create new user with GID and UID of jenkins          # RUN useradd --create-home
--shell /bin/bash jenkins
RUN  mkdir --parents /home/jenkins &&\
      groupadd --system jenkins &&\
      useradd --system --home /home/jenkins --shell /sbin/nologin --gid jenkins
      jenkins
ENV   JENKINS_HOME=/home/jenkins
```

Next, we install Fast Models. This involves adding the tarball to the Docker image (the `ADD` command automatically untars compressed files) and running the install script `setup.sh` with relevant parameters. For the Dockerfile to work with your downloaded Fast Models tarball, you must update the path names to match the version and release numbers that are specific to your `.tgz` downloaded version, as you can see in the following code:

```
# Install FMs
ADD FastModels_11-4-043_Linux64.tgz $JENKINS_HOME/
RUN cd $JENKINS_HOME/FastModels_11-4-043_Linux64/ && ./setup.sh --i-accept-the-
license-agreement --basepath "$JENKINS_HOME/Arm/" &&\
  rm -r $JENKINS_HOME/FastModels_11-4-043_Linux64/
```

Next, we reference the license file, to ensure that the example model can build properly. Replace `localhost` with the network path to your license, if the license server is installed on a remote machine. Make sure that there is no space between the `=` and your file location. Use these commands:

```
# Set License file path
ENV   ARMLMD_LICENSE_FILE=7010@localhost
```

Copy over the example code, and the Fast Model virtual platform is built on the Docker image. The required Fast Model scripts are set in your `bashrc` file. You can invoke the scripts automatically when you run the image manually from inside the Docker container. You must also change the Fast Models version in these variables to match your version. Use these commands:

```
# Setup example FM system
COPY ./m4_system/ $JENKINS_HOME/m4_system/
COPY ./ITMTrace/ $JENKINS_HOME/plugins/
COPY ./run_m4.py $JENKINS_HOME
RUN  . $JENKINS_HOME/Arm/FastModelsTools_11.4/source_all.sh &&\
      cd $JENKINS_HOME/m4_system/model/ &&\
      ./linux_build.sh

# Set FM startup sourcing for manual code work
RUN  echo "\n#FM Startup Code\n" >> $JENKINS_HOME/.bashrc &&\
      echo "source $JENKINS_HOME/Arm/FastModelsTools_11.4/source_all.sh\n"
      >> $JENKINS_HOME/.bashrc
```

Finally, switch the user to `jenkins`, with rights to files and folders in that home directory. Use these commands:

```
# Switch to jenkins user with proper rights to files in $JENKINS_HOME
RUN chown -R jenkins:jenkins $JENKINS_HOME
```

```
USER jenkins
WORKDIR /home/Jenkins
```

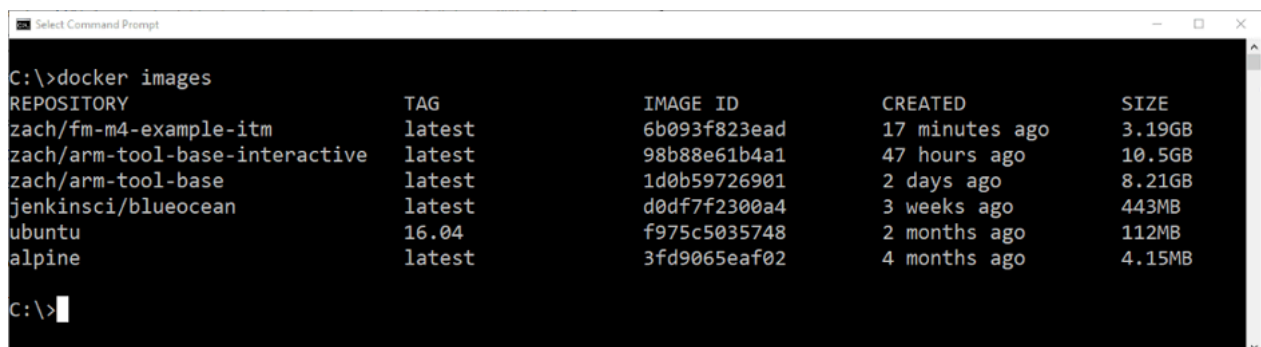
In common Dockerfile syntax, the `\` character denotes a multi-line command, and the `&&` indicates that there is another command to run after the current one finishes.

Run the following command in your terminal or command prompt to build a Docker image that is based on our example Dockerfile. In our example, we tag the Docker image as the user `zach`. You can use your Docker ID if you have one, or you can use a name of your choice:

```
docker build -t zach/fm-m4-example-itm:latest -f Dockerfile .
```

The `-t` specifies the tag for the image, `-f` points to the Dockerfile, and the period `.` at the end specifies the build context. The build context is what files look at when building the image. Because it is a period `.`, the build context is the current directory. Sub-directories can be named here if needed, but are not needed in this case. The `copy` command must point to a file or directory within the noted build context. After building, check that the image was created with the `docker images` command, as you can see here:

Figure 4-2: Docker images



REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
zach/fm-m4-example-itm	latest	6b093f823ead	17 minutes ago	3.19GB
zach/arm-tool-base-interactive	latest	98b88e61b4a1	47 hours ago	10.5GB
zach/arm-tool-base	latest	1d0b59726901	2 days ago	8.21GB
jenkinsci/blueocean	latest	d0df7f2300a4	3 weeks ago	443MB
ubuntu	16.04	f975c5035748	2 months ago	112MB
alpine	latest	3fd9065eaf02	4 months ago	4.15MB

To execute the docker container, run the following command:

```
docker run --rm -ti --cap-drop=all --memory=2G --cpus=1 zach/fm-m4-example-itm:latest
```

The `docker run` command options specify, in order:

`--rm`

Enables a proper clean-up after you exit container

`-ti`

Opens an interactive shell

`--cap-drop=all`

Restricts permissions of the container even further for security

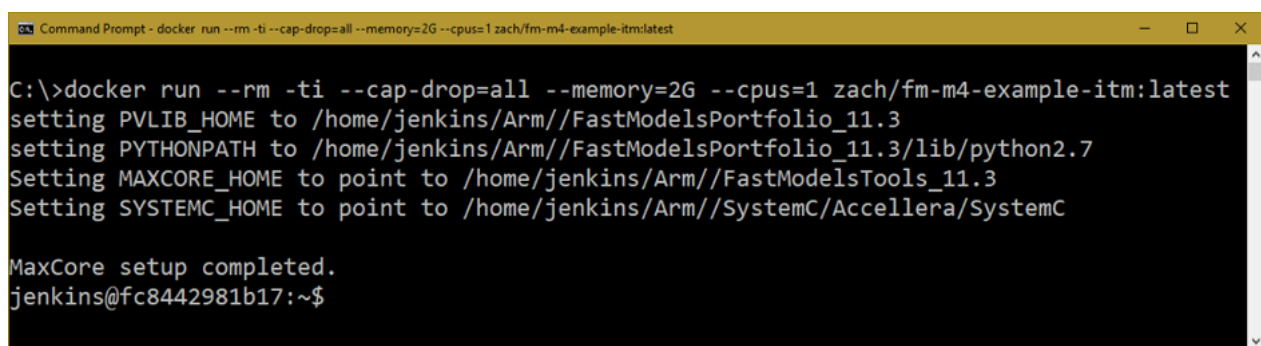
`--memory=2G` and `--cpus=1`

Restricts the maximum resource usage that the container can use at one time for security reasons

`zach/fm-m4-example:latest`

Points to the correct image to run, that is the image that we just built. When running, the Docker image will start in the home directory of the user jenkins. The command prompt should look like the following screenshot:

Figure 4-3: Jenkins home



```

Command Prompt - docker run --rm -ti --cap-drop=all --memory=2G --cpus=1 zach/fm-m4-example-itm:latest

C:\>docker run --rm -ti --cap-drop=all --memory=2G --cpus=1 zach/fm-m4-example-itm:latest
setting PVLIB_HOME to /home/jenkins/Arm//FastModelsPortfolio_11.3
setting PYTHONPATH to /home/jenkins/Arm//FastModelsPortfolio_11.3/lib/python2.7
Setting MAXCORE_HOME to point to /home/jenkins/Arm//FastModelsTools_11.3
Setting SYSTEMC_HOME to point to /home/jenkins/Arm//SystemC/Accellera/SystemC

MaxCore setup completed.
jenkins@fc8442981b17:~$

```

Run an example program on a Cortex-M4 system, noted as `startup_Cortex-M4.axf` under the `m4_system/app_helloWorld` directory. Your example includes a simple Python script to automate the process, using the Fast Model scripting language PyCADL as seen in the following code block:

```

# Import libraries
import sys,os
# Set python path to Fast Models, as a check to see if FM installed properly
try:
    sys.path.append(os.path.join(os.environ['PVLIB_HOME'], 'lib', 'python27'))
except KeyError as e:
    print "Error! Make sure you source all from the fast models directory."
    sys.exit(1) # Exit with error
import fm.debug

def ITM_redirect(file_name):
    targets = model.get_target_info()
    for target_info in targets:
        if target_info.target_name.find("ITMtrace") >= 0:
            target = model.get_target(target_info.instance_name)
            target.parameters["trace-file"] = file_name

jenkins_home = os.environ['JENKINS_HOME']

plugin_path = str(jenkins_home)+"/plugins/ITMtrace.so"
model_path = str(jenkins_home)+"/m4_system/model/cadi_system/cadi_system_Linux64-Release-GCC-5.4.so"
app_path = str(jenkins_home)+"/m4_system/app_helloWorld/startup_Cortex-M4.axf"
out_path = str(jenkins_home)+"/output.txt"

# Set Environmental variable
os.environ["FM_TRACE_PLUGINS"] = plugin_path

```

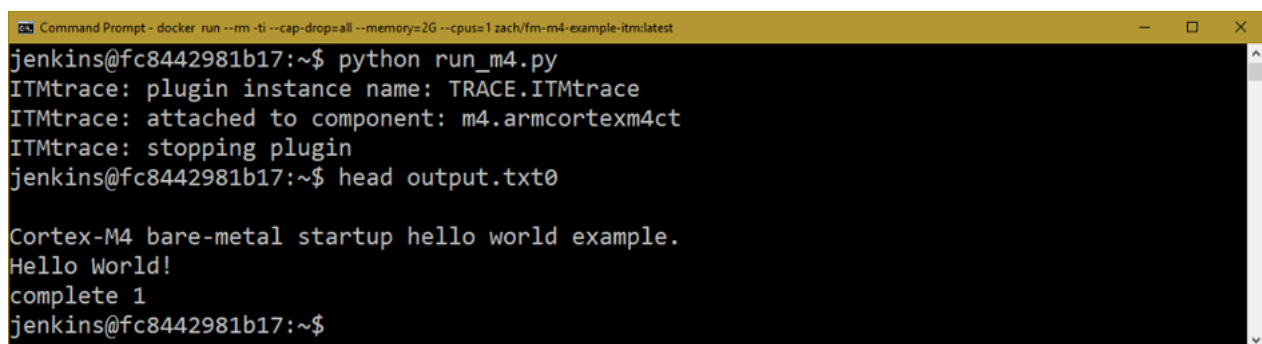
```
# Load model
model = fm.debug.LibraryModel(model_path)
# Get cpu
cpu = model.get_cpus()[0]
# Load app onto cpu
cpu.load_application(app_path)

# Send ITM to stdout
ITM_redirect(out_path)

# Run the model, exit after timeout.
try:
    model.run(timeout=1)
except:
    sys.exit(0)
```

When you run this Python script in the created container, you generate a new file called `output.txt0` with some welcome messages. The `0` at the end of the filename indicates that ITM channel 0 was the used channel. This screenshot shows the commands and their respective outputs:

Figure 4-4: hello world output



```
jenkins@fc8442981b17:~$ python run_m4.py
ITMtrace: plugin instance name: TRACE.ITMtrace
ITMtrace: attached to component: m4.armcortexm4ct
ITMtrace: stopping plugin
jenkins@fc8442981b17:~$ head output.txt0

Cortex-M4 bare-metal startup hello world example.
Hello World!
complete 1
jenkins@fc8442981b17:~$
```

The file `output.txt0` is created after running the test, which contains a welcome message and the traditional Hello World!. In this case, the creation of file `output.txt0` indicates that the Hello world application ran successfully. This indication is verified by the command `head output.txt0` returning the contents of the generated file. A more complicated application might generate multiple files based on some given input, or any other test application that is used to verify code integrity.

Moving to automation

Having one Docker image to share between a team creates a reliable and consistent development environment to operate in. To get the most benefits out of the Docker platform, automating these test runs will be introduced in [Introduction to Jenkins](#).

Automation adds additional benefits, like saving time and version control management. In the following sections of the guide, we will explain Jenkins and set up a working example.

5. Introduction to Jenkins

In this section of the guide, we look at Jenkins. Jenkins is an important tool when enabling continuous integration workflows. Jenkins is an open source continuous integration server that automates and integrates the build, test and merge of your development flow. You can configure Jenkins to fit the needs of your project. We will run Jenkins in a Docker container.

We install Jenkins in an Ubuntu 16.04 VM using the package manager titled Advanced Package Tool (apt). We will then examine:

- How Jenkins usage applies to any Jenkins install implementation
- How to use Jenkins Blue Ocean, a graphical tool that simplifies the continuous integration and delivery process

Install Jenkins on Docker

To install Jenkins with Docker for any OS, follow these steps:

1. Start Jenkins in Docker by running the following command in a terminal/command prompt:

```
docker run \  
-u root \  
--rm \  
-d \  
-p 8080:8080 \  
-p 50000:50000 \  
-v jenkins-data:/var/jenkins_home \  
-v /var/run/docker.sock:/var/run/docker.sock \  
jenkinsci/blueocean
```

For Windows, replace the \ characters with ^ to enable multi-line command line inputs. Here is a brief explanation of each section of the command:

`docker run`

Run a specified image, that is noted later in the command, in a new container

`-u`

Run the container as the user `root`

`--rm`

For cleanliness, the container is automatically removed when it is shut down.

`-d`

Run the container in the background, in detached mode. If not specified, the Docker log for the container will output in the terminal window. Keeping the `-a` option helps keep this command tidy in a shell script, and keeps the launching terminal from being attached to the Jenkins instance.

```
-p 8080:8080
```

Map, or publish, the container port to a host port. The host port number is first, and the container port is second. Port 8080 is used to access Jenkins through a web browser. Port 50000 allows you to use other JNLP-based Jenkins agents on other machines. This functionality is not required for this example, but is good to know about if you are working with a master-slave system.

```
-v jenkins-data:/var/jenkins_home
```

Map host volumes to the container. This means that the container can use, store, and create data on the host. The first `-v` command is used to save Jenkins configuration data to the host machine, so that restarting Jenkins does not mean restarting all your work. The second `-v /var/run/docker.sock:/var/run/docker.sock` allows Jenkins to communicate with the Docker daemon on the host. This is required to run Docker containers through Jenkins, because Jenkins itself is a Docker container.

```
jenkinsci/blueocean
```

Run the Docker container with this name, which is the [blue ocean image that is maintained by Jenkinsci](#). If this image is not already downloaded on your host machine, the `docker run` command will automatically download the image.

2. Follow the Post-installation setup wizard:
 - a. Navigate to this address in a web browser: `localhost:8080`
 - b. Follow basic setup instructions. For first-time users, a special passcode is required which is stored in a local file. The Jenkins setup wizard will guide you through how to find and enter this passcode.
 - c. Install recommended plug-ins.
 - d. Create the first admin user with your selected username and password.

Check that the Open Blue Ocean option is on the left side of the browser when on `localhost:8080`, because it should be pre-installed with the Docker image. If not, see step 3 of the steps in [Install Jenkins on Linux](#) for instructions on how to get the Blue Ocean plug-in.

Install Jenkins on Linux

If you prefer to install Jenkins directly on your Linux machine and not in a Docker container, follow these steps. On Linux, you can install Jenkins through the Advanced Package Tool (`apt`). This is useful if you are not able to host Jenkins within a Docker container. As an extra requirement, however, you must have Java installed for Jenkins to install and work on Linux. Java is not required when installing through Docker.

Type `java -version` into your command-line. If Java is not installed, install it with the following terminal command:

```
sudo apt install openjdk-8-jre
```

To install Jenkins, follow these steps:

1. Run these commands from a terminal:

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -  
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/  
sources.list.d/jenkins.list'  
sudo apt-get update  
sudo apt-get install jenkins
```

Jenkins should automatically start and is accessible through a web browser at the address `localhost:8080`. To complete the installation, additional steps are required:

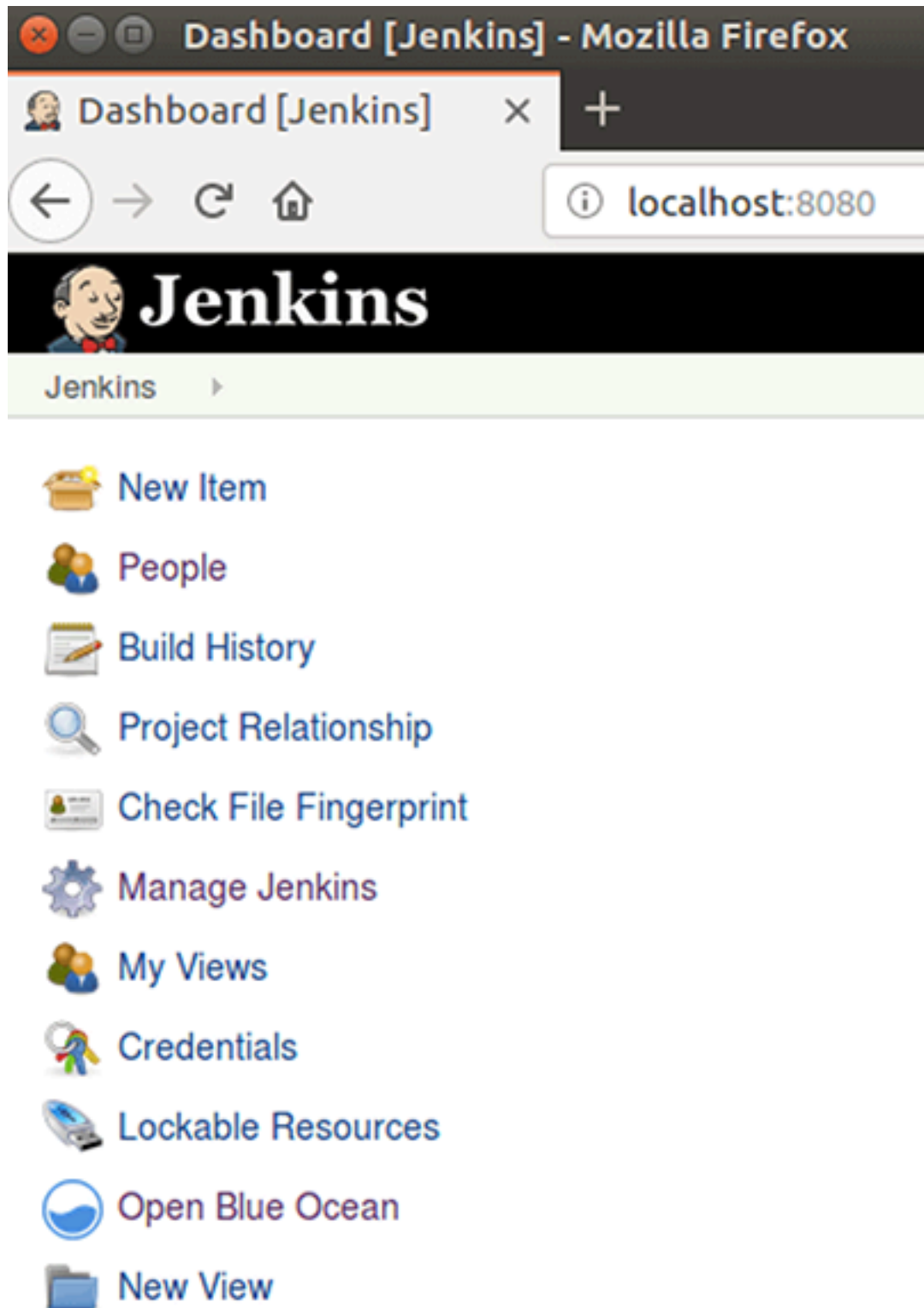
2. In a web browser, navigate to `localhost:8080`. Follow the basic setup instructions.



For first-time users only, a special pass code, which is stored in a local file, is required. The Jenkins setup wizard will guide you through how to find and enter this pass code.

3. Install the recommended plug-ins.
4. Create the first admin user with the username and password of your choice.
5. Install the BlueOcean plug-in.
6. Navigate to Manage Jenkins > Plugin Manager using the left navigation bars or go to the url `localhost:8080/pluginManager`.
7. Navigate to the Available tab and type in Blue Ocean. Select the top option, and Jenkins will install dependencies automatically.
8. Click Install without restart. After installation, refresh the page and navigate back to `localhost:8080`. You will see a new option. Open Blue Ocean in the left-hand menu, as you can see in the following screenshot:

Figure 5-1: Jenkins dashboard



Now that Jenkins is installed, we can create the first pipeline. Let's look at this in [Create pipeline](#).

6. Create pipeline

In this guide, a pipeline refers to two things:

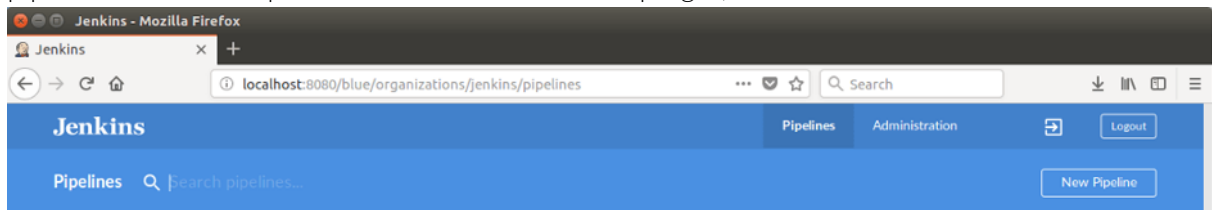
- A Continuous Integration (CI) pipeline
- A Jenkins pipeline

A CI pipeline is the sequence of steps in a CI flow. An example CI pipeline consists of running some unit tests, and verifying that they pass every time they merge code to version control. Your CI pipeline is to build the software, test the software, then merge the software in version control. A Jenkins pipeline is a suite of different plug-ins that enable Jenkins to script CI pipelines. In this guide, you can assume that the term pipeline refers to a CI pipeline, unless we refer to these Jenkins plugins.

A `Jenkinsfile` defines each step in the pipeline, and can be managed and tracked in source control, just like any other code. This enables you to use CI best practices with the CI pipeline configuration file itself. Blue Ocean provides a GUI interface to edit the pipeline code and the `Jenkinsfile`.

Follow these steps to create a pipeline:

1. Click on Open Blue Ocean on the left of the Jenkins GUI.
2. Navigate to the New Pipeline button, if you do not see a prompt to create a new pipeline. The New Pipeline button will be in the top right, as shown in this screenshot:



3. Select your source control repository. We recommend SSH, which works well with both Jenkins and git. When you enter your repository URL, Jenkins will generate a public SSH key, which you can include in your own git server.
4. Select Create Pipeline to open a new screen, which is the Jenkins Blue Ocean Pipeline editor.

Depending on whether you are using Docker to run the Fast Models, or running locally on Linux, you will configure slightly different Jenkins pipelines. Both steps and outputs are detailed in [Set up simulation in Docker](#).

7. Set up simulation in Docker

In this section of the guide, we will describe how to run a Fast Model simulation in Docker. You can use this method with any Jenkins install implementation, through `apt` on Linux or through Docker on any OS. Follow these steps:

1. Click on the plus sign icon to create the first test stage, which will add two steps in the pipeline: running the Python script and checking its output. Give the file a name, for example, Test.
2. Click on the add step > Shell Script buttons to add the following two steps:

```
set +e &&  
. //root//ARM//FastModelsTools_11.3//source_all.sh &&  
set -e &&  
python //root//FMs//run_m4.py &&
```

and

```
head //root//FMs//output.txt0
```

Because Jenkins will run the Docker container non-interactively, you must source the Fast Models tools manually before you can run the Fast Model using Python. This is usually done on startup with `.bashrc`, which does not apply here because Jenkins does not run Docker commands in a login shell.

3. Select the Test stage that you just created and can see at the bottom right of the screen, then select the Settings button.
4. Set the Agent to docker and set the Image to `zach/fm-m4-example:latest`. This image is what was built in the previous section, [Create and verify a Docker image](#). When running the Jenkins pipeline, that Docker image will be pulled and the shell scripts will run in the container that is created.

Each time that a change is made on the Blue Ocean GUI, the underlying `Jenkinsfile` changes, which dictates the pipeline behavior at runtime. To view the `Jenkinsfile` code, use the hotkey CTRL-S. You can edit the code using this method, which is helpful when the Blue Ocean GUI does not offer the necessary syntax option.

However, adding too much code that is not represented by the Blue Ocean GUI can be confusing, because your system behavior is hidden from the Blue Ocean GUI.

Now the pipeline script is set up and should look like what you can see in the following code:

```
pipeline {  
  agent none  
  stages {  
    stage('Test') {  
      agent {  
        docker {  
          image 'zach/fm-m4-example:latest'  
        }  
      }  
    }  
  }  
  steps {
```

```
        sh '''set +e &&
. //root//ARM//FastModelsTools_11.3//source_all.sh &&
set -e &&
python //root//FMs//run_m4.py'''
        sh "head //root7/FMs//output.txt0"
    }
}
}
```

8. Set up simulation in host

If you do not use Docker to run Arm Fast Model simulations, as described in [Set up simulation in Docker](#), you can use the instructions in this section of the guide. Before you begin, ensure that the Arm Fast Models are initialized on your host Linux system.

Follow these steps to set up a Fast Model simulation using Jenkins:

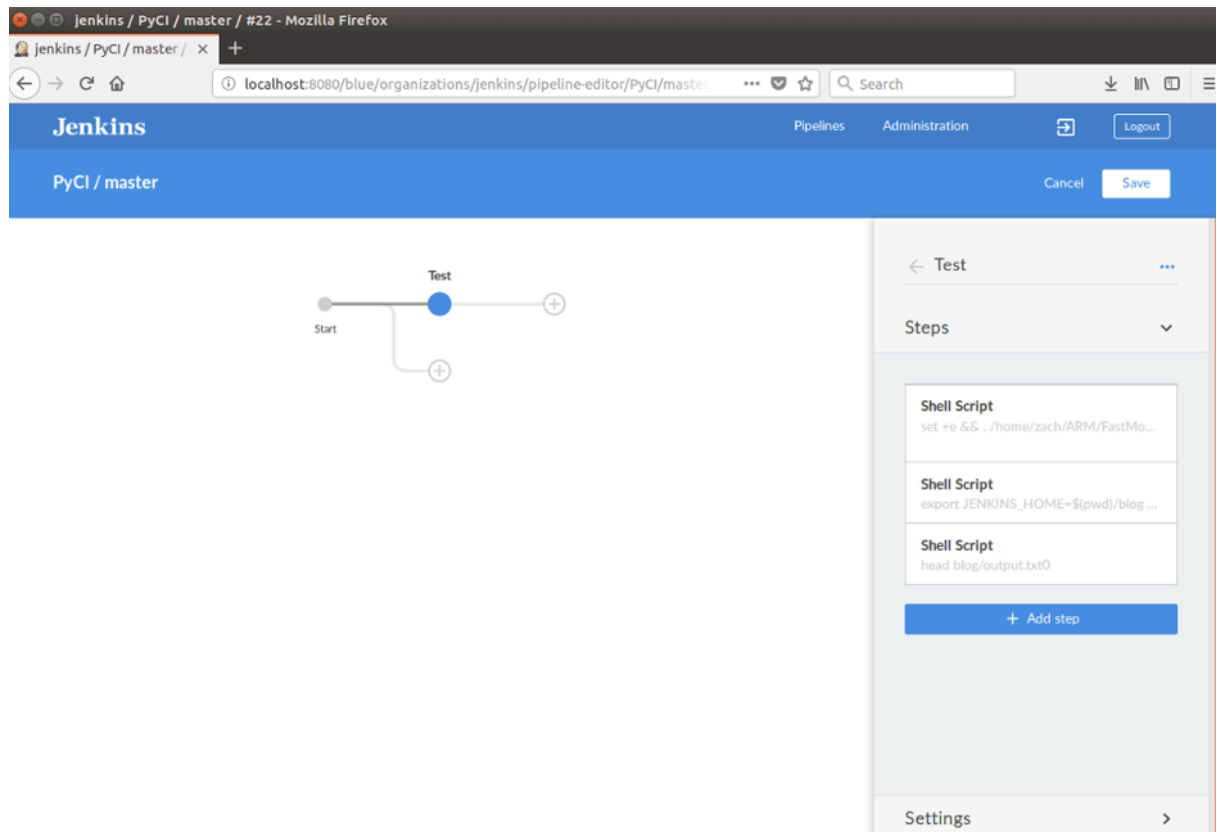
1. Download the .zip files from [Before you begin](#) and untar them in a directory in the same git repository that this Jenkins pipeline is connected to. We will call this repository `$BLOGBASEDIR`.
2. Rename the directory that is called `ITMtrace` to `plug-ins`, or you will get an error that the Python script cannot find the correct Fast Model plug-in.
3. Commit the git repository with `$BLOGBASEDIR` included. This will allow Jenkins to see these files in the git repository.

Here are the steps to run Fast Model simulations on the Linux host:

1. Build the hardware virtual platform model. The hardware virtual platform models are not pre-built, because the files are large enough to cause problems uploading to some git repositories.
2. Run the simulation.
3. Check the results.

The steps will look like this in the Blue Ocean GUI:

Figure 8-1: Blue ocean GUI



- Click on the add step > Shell Script to add the following three steps into the Blue Ocean GUI:

```
set +e
&&
. //home//zach/ARM//FastModelsTools_11.3//source_all.sh &&
set -e &&
cd $BLOGBASEDIR//m4_system//model
&& ./linux_build.sh
```

and

```
set +e &&
. //home//zach/ARM//FastModelsTools_11.3//source_all.sh &&
set -e &&
python $BLOGBASEDIR//run_m4.py
```

and

```
head $BLOGBASEDIR//output.txt0
```

- Replace the paths to the Fast Models tools with the correct ones for your system. The `output.txt0` file will be placed in `$BLOGBASEDIR` where the `run_m4.py` script is also located.

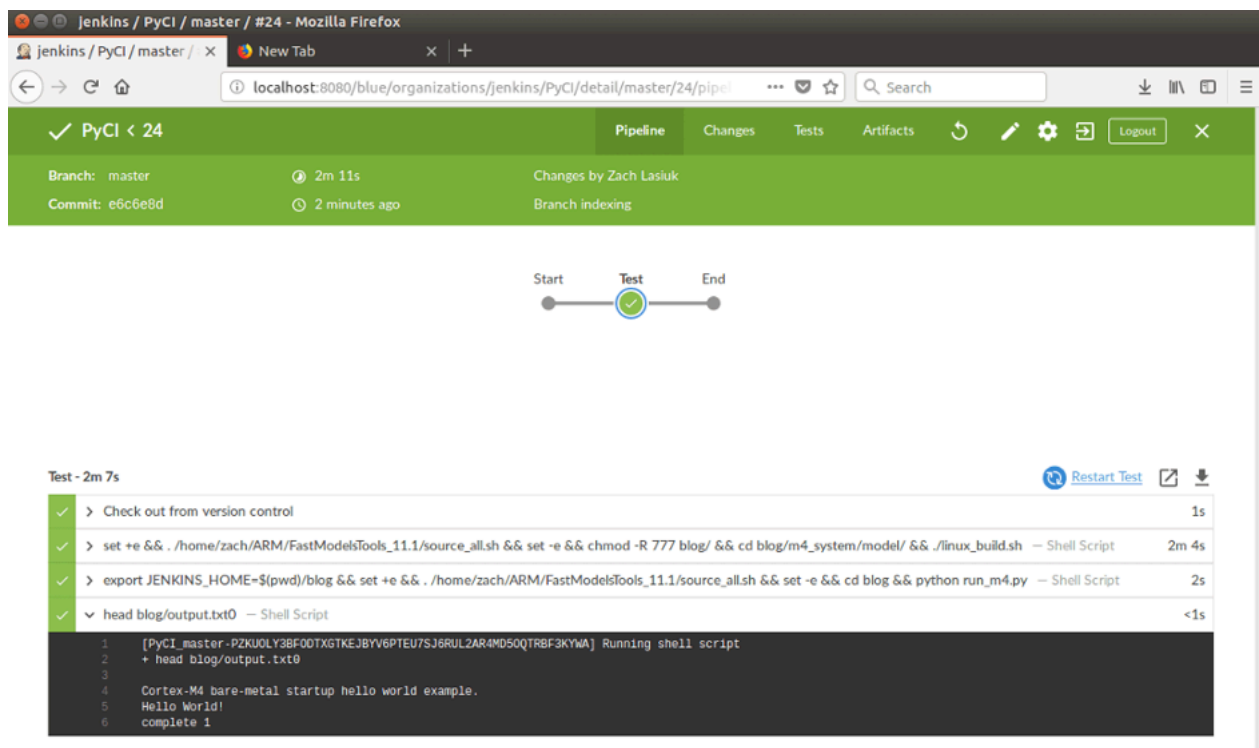
9. Run pipeline

In this section of the guide, we will execute the CI pipeline. To save and run the script, do the following:

- Click Save in the top right, then Save & run. This will send you to a new page which records current and past runs. Selecting the current run will show different stats, along with the overall results. You may need to refresh the page to see results in a timely manner.

If you are also building the virtual model on the Linux host machine, and following the steps for setting up the simulation on the Linux host instead of Docker, the test will take 1-2 minutes. This screenshot shows a run completed from running on the Linux host machine, with the `$BLOGBASEDIR` called `blog/`:

Figure 9-1: Linux host



You can see in the preceding screenshot that:

- The pipeline with the name `PyCI` ran its 24th test successfully
- The expected output from the `output.txt0` file is displayed

If `output.txt0` was not generated, the Jenkins pipeline would stop at that step and give an error. This means that, in our example, whenever the pipeline passes our code can be considered correct, by outputting a hello message.

You can extend this type of test infrastructure to, for example, file existence checks, equivalence checks, content verification, customized test report analysis, and code coverage. You can also set up the Jenkins pipeline to:

- Run before any commit to a certain git branch, or
- Include merging branches when all tests pass, ensuring that certain branches are always working at a defined state.

10. Related information

Here are some resources that are related to material in this guide:

- [Arm Cortex-M4](#) processor
- [Arm Fast Models](#)
- [Docker website](#)
 - [Docker Hub](#)
 - [Docker installation page](#)
- [Jenkins](#)
- [Blog with more information about PyCADI](#)

11. Next steps

Using Docker, Jenkins, and Arm tools for embedded software development saves time and increases quality by providing a consistent and automated CI workflow across a development team. These tools provide opportunities to streamline software development. For another example of how to create an embedded software development CI flow with Jenkins, Docker and Arm Fast Models you can look at this [GitHub example](#).