

# **@criticalmanufacturing/ cli Documentation**

---

# Table of contents

---

1. About cmf-cli	4
1.1 Use cmf-cli to . . .	4
1.2 Getting started	4
2. Installation	5
2.1 Checking your version of cmf-cli and Node.js	5
3. Plugins	6
4. Telemetry	7
4.1 Telemetry implementation	7
4.2 Pipelines	7
5. Commands	8
5.1 assemble	8
5.2 build	9
5.3 build help	10
5.4 build help generateBasedOnTemplates	11
5.5 build help generateMenuItems	12
5.6 bump	13
5.7 bump iot	14
5.8 bump iot configuration	15
5.9 bump iot customization	16
5.10 init	17
5.11 ls	19
5.12 new	20
5.13 new business	21
5.14 new data	22
5.15 new database	23
5.16 new feature	24
5.17 new help	25
5.18 new html	26
5.19 new iot	27
5.20 new test	28
5.21 pack	29
5.22 restore	30
6. Scaffolding	31
6.1 Scaffolding	31
6.2 Scaffolding and Pipelines for non-CMF Users	33

6.3	Feature package scaffolding	36
6.4	Infrastructure config file	37
6.5	Migration Paths	38
6.6	Pipeline Import	39
6.7	Pipelines description	46
6.8	Exported Objects package	49
6.9	Post-scaffolding package tailoring	50
6.10	Traditional scaffolding	52

# 1. About cmf-cli

---

cmf-cli is a Command Line Interface used for Critical Manufacturing developments.

## 1.1 Use cmf-cli to . . .

---

- Scaffold a new repository
- Generate new package structures
- Adapt packages of code for Critical Manufacturing MES
- Manage multiple versions of packages and package dependencies
- Create packages that can be used by any developer or customer
- View the package tree
- Restore packages for local development
- Assemble a release bundle

and a lot more!

## 1.2 Getting started

---

To get started with cmf-cli, you need to use the command line interface (CLI) to [install cmf-cli](#). We look forward to seeing what you create!

## 2. Installation

---

To be able to install cfm-cli, you must install Node.js and the npm command line interface using either a Node version manager or a Node installer. **We strongly recommend using a Node version manager like [nvm](#) to install Node.js and npm.** We do not recommend using a Node installer, since the Node installation process installs npm in a directory with local permissions and can cause permissions errors when you run npm packages globally.

```
npm install --global @criticalmanufacturing/cli
```

### 2.1 Checking your version of cmf-cli and Node.js

---

To see if you already have Node.js and npm installed and check the installed version, run the following commands:

```
node -v  
cmf -v
```

## 3. Plugins

---

The Critical Manufacturing cli is designed with a plugin system for extensibility. In the future, it will be possible to search for plugins straight from cli. Check issue [#11](#) for progress.

In the meanwhile, some plugins are already in development. Here follows a non-exhaustive plugin list:

- [Portal SDK](#) - command line tools to interact with the Critical Manufacturing Customer Portal

## 4. Telemetry

---

### 4.1 Telemetry implementation

---

Basic telemetry currently only tracks the CLI startup and logs:

- CLI name and version
- latest version available in NPM
- if the CLI version is stable or testing
- if the CLI is outdated

#### **no identifiable information is collected in basic telemetry**

However, any user can optionally enable extended telemetry, which might help with troubleshooting. **Extended telemetry includes identifiable information** and as such should be used with care. This includes the basic telemetry, plus:

- for the version (startup) log, it also includes
- current working directory
- hostname
- IP
- username

It also tracks and logs the package tree if, for any command, the tree must be computed. This includes all of the above information plus the package name, for each package in the tree.

Enabling telemetry can be done via environment variables:

- `cmf_cli_enable_telemetry` - enable basic telemetry. If this is off (the default), no telemetry will be collected, even if extended telemetry is on. To enable, set to `true` or `1`, do not set or set to `false` or `0` to disable.
- `cmf_cli_enable_extended_telemetry` - enable extended telemetry. Note the above warnings regarding the impact of keeping this on. To enable, set to `true` or `1`, do not set or set to `false` or `0` to disable.
- `cmf_cli_telemetry_enable_console_exporter` - also print the telemetry information to the console. This is for auditing or troubleshooting as it makes the console output extremely verbose
- `cmf_cli_telemetry_host` - specify an alternate telemetry endpoint (if you're hosting your own)

### 4.2 Pipelines

---

**NOTE: The CI-Package pipeline is shipped with telemetry ON.** The provided pipelines are built for Critical Manufacturing use and provided as an example for the user to build their own. As at CM we keep telemetry on for troubleshooting, the generated pipelines have telemetry on. To disable, search for the `cmf_cli_enable_telemetry` environment variable in the pipeline YAML and disable it.

## 5. Commands

---

### 5.1 assemble

---

#### 5.1.1 Usage

```
cmf assemble [options] [<workingDir>]
```

#### Arguments

Name	Description
<workingDir>	Working Directory [default: .]

#### Options

Name	Description
-o, --outputDir <outputDir>	Output directory for assembled package [default: Assemble]
--cirepo <cirepo>	Repository where Continuous Integration packages are located (url or folder)
-r, --repo, --repos <repos>	Repository or repositories where published dependencies are located (url or folder)
--includeTestPackages	Include test packages on assemble
-, -h, --help	Show help and usage information



## 5.2 build

---

### 5.2.1 Usage

---

```
cmf build [options] [<packagePath>] [command]
```

#### Arguments

Name	Description
<packagePath>	Package Path [default: .]

#### Options

Name	Description
-?, -h, --help	Show help and usage information

#### Commands

Name	Description
help	

## 5.3 build help

---

### 5.3.1 Usage

---

```
cmf build [<packagePath>] help [options] [command]
```

**Arguments**

Name	Description
<packagePath>	Package Path [default: .]

**Options**

Name	Description
-?, -h, --help	Show help and usage information

**Commands**

Name	Description
generateBasedOnTemplates	
generateMenuItems	

## 5.4 build help generateBasedOnTemplates

---

### 5.4.1 Usage

---

```
cmf build [<packagePath>] help generateBasedOnTemplates [options]
```

#### Options

Name	Description
-?, -h, --help	Show help and usage information

---

## 5.5 build help generateMenuItems

---

### 5.5.1 Usage

---

```
cmf build [<packagePath>] help generateMenuItems [options]
```

**Options**

Name	Description
-?, -h, --help	Show help and usage information

---

## 5.6 bump

---

### 5.6.1 Usage

---

```
cmf bump [options] [<packagePath>] [command]
```

#### Arguments

Name	Description
<packagePath>	Package path [default: .]

#### Options

Name	Description
-v, --version <version>	Will bump all versions to the version specified
-b, --buildNr <buildNr>	Will add this version next to the version (v-b)
-r, --root <root>	Will bump only versions under a specific root folder (i.e. 1.0.0)
-, -h, --help	Show help and usage information

#### Commands

Name	Description
iot	

## 5.7 bump iot

---

### 5.7.1 Usage

---

```
cmf bump [<packagePath>] iot [options] [command]
```

#### Arguments

Name	Description
<packagePath>	Package path [default: .]

#### Options

Name	Description
-?, -h, --help	Show help and usage information

#### Commands

Name	Description
configuration <path>	[default: .]
customization <packagePath>	[default: .]

## 5.8 bump iot configuration

### 5.8.1 Usage

```
cmf bump [<packagePath>] iot configuration [options] [<path>]
```

#### Arguments

Name	Description
<path>	Working Directory [default: .]

#### Options

Name	Description
-v, --version <version>	Will bump all versions to the version specified
-b, --buildNrVersion <buildNrVersion>	Will add this version next to the version (v-b)
-md, --masterData	Will bump IoT MasterData version (only applies to .json) [default: False]
-iot	Will bump IoT Automation Workflows [default: True]
-pckNames, --packageNames <packageNames>	Packages to be bumped
-r, --root <root>	Specify root to specify version where we want to apply the bump
-g, --group <group>	Group of workflows to change, typically they are grouped by Automation Manager
-wkflName, --workflowName <workflowName>	Specific workflow to be bumped
-isToTag	Instead of replacing the version will add -\$version [default: False]
-mdCustomization	Instead of replacing the version will add -\$version [default: False]
-, -h, --help	Show help and usage information

## 5.9 bump iot customization

---

### 5.9.1 Usage

```
cmf bump [<packagePath>] iot customization [options] [<packagePath>]
```

#### Arguments

Name	Description
<packagePath>	Package Path [default: .]

#### Options

Name	Description
-v, --version <version>	Will bump all versions to the version specified
-b, --buildNrVersion <buildNrVersion>	Will add this version next to the version (v-b)
-pckNames, --packageNames <packageNames>	Packages to be bumped
-isToTag	Instead of replacing the version will add -\$version [default: False]
-, -h, --help	Show help and usage information



## 5.10 init

---

### 5.10.1 Usage

---

```
cmf init [options] <projectName> [<rootPackageName> [<workingDir>]]
```

#### Arguments

Name	Description
<projectName>	
<rootPackageName>	[default: Cmf.Custom.Package]
<workingDir>	Working Directory [default: .]

## Options

Name	Description
<code>--version &lt;version&gt;</code>	Package Version [default: 1.0.0]
<code>-c, --config &lt;config&gt; (REQUIRED)</code>	Configuration file exported from Setup [default: ]
<code>--repositoryUrl &lt;repositoryUrl&gt; (REQUIRED)</code>	Git repository URL
<code>--deploymentDir &lt;deploymentDir&gt; (REQUIRED)</code>	Deployments directory
<code>--MESVersion &lt;MESVersion&gt; (REQUIRED)</code>	Target MES version
<code>--DevTasksVersion &lt;DevTasksVersion&gt; (REQUIRED)</code>	Critical Manufacturing dev-tasks version
<code>--HTMLStarterVersion &lt;HTMLStarterVersion&gt; (REQUIRED)</code>	HTML Starter version
<code>--yoGeneratorVersion &lt;yoGeneratorVersion&gt; (REQUIRED)</code>	@criticalmanufacturing/html Yeoman generator version
<code>--nugetVersion &lt;nugetVersion&gt; (REQUIRED)</code>	NuGet versions to target. This is usually the MES version
<code>--testScenariosNugetVersion &lt;testScenariosNugetVersion&gt; (REQUIRED)</code>	Test Scenarios Nuget Version
<code>--infra, --infrastructure &lt;infrastructure&gt;</code>	Infrastructure JSON file [default: ]
<code>--nugetRegistry &lt;nugetRegistry&gt;</code>	NuGet registry that contains the MES packages
<code>--npmRegistry &lt;npmRegistry&gt;</code>	NPM registry that contains the MES packages
<code>--azureDevOpsCollectionUrl &lt;azureDevOpsCollectionUrl&gt;</code>	The Azure DevOps collection address
<code>--agentPool &lt;agentPool&gt;</code>	Azure DevOps agent pool
<code>--agentType &lt;Cloud Hosted&gt;</code>	Type of Azure DevOps agents: Cloud or Hosted
<code>--ISOLocation &lt;ISOLocation&gt;</code>	MES ISO file [default: ]
<code>--nugetRegistryUsername &lt;nugetRegistryUsername&gt;</code>	NuGet registry username
<code>--nugetRegistryPassword &lt;nugetRegistryPassword&gt;</code>	NuGet registry password
<code>--cmfCliRepository &lt;cmfCliRepository&gt;</code>	NPM registry that contains the CLI
<code>--cmfPipelineRepository &lt;cmfPipelineRepository&gt;</code>	NPM registry that contains the CLI Pipeline Plugin
<code>--releaseCustomerEnvironment &lt;releaseCustomerEnvironment&gt;</code>	Customer Environment Name defined in DevOpsCenter
<code>--releaseSite &lt;releaseSite&gt;</code>	Site defined in DevOpsCenter
<code>--releaseDeploymentPackage &lt;releaseDeploymentPackage&gt;</code>	DeploymentPackage defined in DevOpsCenter
<code>--releaseLicense &lt;releaseLicense&gt;</code>	License defined in DevOpsCenter
<code>--releaseDeploymentTarget &lt;releaseDeploymentTarget&gt;</code>	DeploymentTarget defined in DevOpsCenter
<code>-, -h, --help</code>	Show help and usage information

# 5.11 ls

## 5.11.1 Usage

```
cmf ls [options] [<workingDir>]
```

### Arguments

Name	Description
<workingDir>	Working Directory [default: .]

### Options

Name	Description
-r, --repo, --repos <repos>	Repositories where dependencies are located (folder)
-?, -h, --help	Show help and usage information

## 5.12 new

### 5.12.1 Usage

```
cmf new [options] [command]
```

#### Options

Name	Description
<code>--reset</code>	Reset template engine. Use this if after an upgrade the templates are not working correctly.
<code>-, -h, --help</code>	Show help and usage information

#### Commands

Name	Description
<code>business &lt;workingDir&gt;</code>	[default: ]
<code>database &lt;workingDir&gt;</code>	[default: ]
<code>data &lt;workingDir&gt;</code>	[default: ]
<code>feature &lt;packageName&gt; &lt;workingDir&gt;</code>	[default: ]
<code>help &lt;workingDir&gt;</code>	[default: ]
<code>html &lt;workingDir&gt;</code>	[default: ]
<code>iot &lt;workingDir&gt;</code>	[default: ]
<code>securityPortal &lt;workingDir&gt;</code>	[default: ]
<code>test</code>	

## 5.13 new business

---

### 5.13.1 Usage

---

```
cmf new business [options] [<workingDir>]
```

**Arguments**

Name	Description
<workingDir>	Working Directory [default: ]

**Options**

Name	Description
--version <version>	Package Version [default: 1.0.0]
-, -h, --help	Show help and usage information

## 5.14 new data

---

### 5.14.1 Usage

---

```
cmf new data [options] [<workingDir>]
```

#### Arguments

Name	Description
<workingDir>	Working Directory [default: ]

#### Options

Name	Description
--version <version>	Package Version [default: 1.0.0]
--businessPackage <businessPackage>	Business package where the Process Rules project should be built [default: ]
?, -h, --help	Show help and usage information

## 5.15 new database

---

### 5.15.1 Usage

---

```
cmf new database [options] [<workingDir>]
```

#### Arguments

Name	Description
<workingDir>	Working Directory [default: ]

#### Options

Name	Description
--version <version>	Package Version [default: 1.0.0]
-, -h, --help	Show help and usage information

## 5.16 new feature

---

### 5.16.1 Usage

---

```
cmf new feature [options] <packageName> [<workingDir>]
```

#### Arguments

Name	Description
<packageName>	The Feature package name
<workingDir>	Working Directory [default: ]

#### Options

Name	Description
--version <version>	Package Version [default: 1.0.0]
-, -h, --help	Show help and usage information



## 5.17 new help

---

### 5.17.1 Usage

---

```
cmf new help [options] [<workingDir>] [command]
```

#### Arguments

Name	Description
<workingDir>	Working Directory [default: ]

#### Options

Name	Description
--version <version>	Package Version [default: 1.0.0]
--docPkg, --documentationPackage <documentationPackage> (REQUIRED)	Path to the MES documentation package
-, -h, --help	Show help and usage information

## 5.18 new html

---

### 5.18.1 Usage

---

```
cmf new html [options] [<workingDir>]
```

#### Arguments

Name	Description
<workingDir>	Working Directory [default: ]

#### Options

Name	Description
--version <version>	Package Version [default: 1.0.0]
--htmlPackage, --htmlPkg <htmlPackage> (REQUIRED)	Path to the MES Presentation HTML package
-, -h, --help	Show help and usage information

## 5.19 new iot

---

### 5.19.1 Usage

---

```
cmf new iot [options] [<workingDir>] [command]
```

#### Arguments

Name	Description
<workingDir>	Working Directory [default: ]

#### Options

Name	Description
--version <version>	Package Version [default: 1.0.0]
-, -h, --help	Show help and usage information

#### Commands

Name	Description
configuration <path>	[default: .]
customization <packagePath>	[default: .]

## 5.20 new test

---

### 5.20.1 Usage

---

```
cmf new test [options]
```

**Options**

Name	Description
--version <version>	Package Version [default: 1.0.0]
-, -h, --help	Show help and usage information

## 5.21 pack

### Description

cmf pack is a package creator for the CM MES developments. It puts files and folders in place so that CM Deployment Framework is able to install them.

It is extremely configurable to support a variety of use cases. Most commonly, we use it to pack the developments of CM MES customizations.

Run `cmf pack -h` to get a list of available arguments and options.

### Important

cmf pack comes with preconfigured [Steps](#) per [PackageType](#) to run during the installation. This pre defined steps are assuming a restrict structure during the installation, this can be disabled using the flag `isToSetDefaultSteps:false` in your `cmfpackage.json`.

### How it works

When the cmf pack is executed it will search in the working directory, for a `cmfpackage.json` file, that then is serialized to the [CmfPackage](#) this will guarantee that the `cmfpackage.json` has all the valid and needed fields. Then it will get which is the [PackageType](#), and based on that will generate the package.

### 5.21.1 Usage

```
cmf pack [options] [<workingDir>]
```

### Arguments

Name	Description
<code>&lt;workingDir&gt;</code>	Working Directory [default: .]

### Options

Name	Description
<code>-o, --outputDir &lt;outputDir&gt;</code>	Output directory for created package [default: Package]
<code>-f, --force</code>	Overwrite all packages even if they already exists
<code>-, -h, --help</code>	Show help and usage information

## 5.22 restore

---

### Description

`cmf restore` allows fetching development dependencies from Deployment Framework (DF) packages, as an alternative to the stricter NuGet and NPM packages.

### How it works

Running this command, any dependencies defined in `cmfpackage.json` will be obtained from the configured repositories (either specified via command option or registered in the `repositories.json` file) and are then unpacked to the `Dependencies` folder inside the package. Then each solution can add references/link packages from the `Dependencies` folder.

### 5.22.1 Usage

---

```
cmf restore [options] <packagePath>
```

### Arguments

Name	Description
<code>&lt;packagePath&gt;</code>	Package path

### Options

Name	Description
<code>-r, --repo, --repos &lt;repos&gt;</code>	Repositories where dependencies are located (folder)
<code>-, -h, --help</code>	Show help and usage information

## 6. Scaffolding

---

### 6.1 Scaffolding

---

#### 6.1.1 Pre-requisites

Though `@criticalmanufacturing/cli` runs with the latest `node` version, to run scaffolding commands the versions required by the target MES version are **mandatory**.

For MES v8, the recommended versions are:

- latest node 12 (Erbium)
- latest npm 6 (should come with node)

Apart from those, scaffolding also needs the following dependencies:

```
npm install -g windows-build-tools
npm install -g gulp@3.9.1
npm install -g yo@3.1.1
```

#### Infrastructure

Rarely changing information, possibly sensitive, like NuGet or NPM repositories and respective access credentials are considered infrastructure. More information on how to set up your own is available at [Infrastructure](#)

#### Environment Config

A valid MES installation is required to initialize your repository, either installed via Setup or via DevOps Center. For the Setup: - in the final step of the Setup, click Export to obtain a JSON file with your environment settings For DevOps Center: - Open your environment and click Export Parameters

Both these files contain sensitive information, such as user accounts and authentication tokens. They need to be provided to the `init` command with:

```
cmf init --config <config file.json> --infra...
```

#### 6.1.2 Scaffolding a repository

Let's start by cloning the empty repository.

```
git clone https://git.example/repo.git
```

Move into the repository folder

```
cd repo
```

For a classic project example, check the [traditional](#) structure documentation.

For more advanced structures, you'll probably be using [Features](#).

## 6.1.3 Continuous Integration and Delivery

---

The scaffolding templates provide a few pipelines designed for [Azure DevOps](#). They work both in [Azure DevOps Server](#) and [Azure DevOps Services](#).

**IMPORTANT:** Only the pipelines for Pull Request and for Package generation (CI-Changes and CI-Package) are designed to run outside of Critical Manufacturing infrastructure. We currently do not support running the Continuous Delivery part of the pipelines in a client infrastructure.

The YAML files are available in the Builds folder at the repository root. Next to them are some JSON files which contain the metadata for the pipelines in Azure DevOps format, which can be used by directly invoking the Azure DevOps API. These files are in git ignore and they should **not** be committed, as they can contain secrets in plain text, such as Nuget credentials.

For CMFers: you can use an internal tool to import the pipeline metadata, as well as the branch policies. Check "How To Import Builds" in the COMMON wiki at Docs/Pipelines.

The **CD-Containers** pipeline requires a secret to be created into the Azure DevOps library. Check more details in the [pipeline import document](#).

## 6.1.4 Manual Pipeline import

---

For non-CMFs, it's simple to import the pipelines. Check out [this document](#).

## 6.1.5 External Users

---

There is more available information for non-CMFers at [External Users](#).



## 6.2 Scaffolding and Pipelines for non-CMF Users

This document details a base approach on how to use `@criticalmanufacturing/cli` to scaffold a Product Customization solution if you are not in the CMF infrastructure. This guide assumes it is being executed in a Windows machine, but can be run in Linux with PowerShell Core if needed. Adapt as required.

### 6.2.1 Notes on the Local Repository

In this document, it is assumed that the user has got a copy of the necessary packages from CMF. However, it is also possible that the user simply has access to the needed repositories. If this is the case, skip the repository setup steps.

If the user is setting up their own repository, this guide is using the Sonatype Nexus v3 repo as an example. This repository provides hosting for NuGet, NPM and raw (i.e. Zip file) packages. Other repositories can be used, hosted together or not. However, it may be left to the user to determine the correct endpoints. In this document, it is assumed the local repository live at `example.local` and is an OSS Nexus v3.

For hosting Nexus, the easiest way is to use the official Docker image. The setup of the repository is not detailed in this guide but information can be found in Sonatype's help page.

The Nexus repository used in this guide needs to:

1. Have a NuGet repository. In this guide it is called `nuget-hosted`. We do not use the proxy or grouped repository types, as the public packages are directly referenced from `nuget.org`
2. Have an NPM repository. In this guide it is called `npm` and is of type `npm-grouped`. Unlike NuGet, here we do setup proxy and grouped repositories. The configuration is as follows:
  - a. Hosted npm repository `npm-hosted`: this will contain the private CMF packages
  - b. Proxy npm repository `npm-proxy`: this repository only proxies public package request to `npmjs.org`
  - c. Grouped npm repository `npm`: this repository is a group of the `npm-hosted` and `npm-proxy` repositories.
- d. In Settings > Realms, have NPM Bearer Token and NuGet API Key realms active. The API Key can be obtained from the user profile.

### Authentication

By default, NuGet and NPM repositories operate with anonymous access, requiring only credentials to publish. If this does not meet your requirements, you can opt to require authentication to all feeds.

#### NUGET

1. In Nexus, go to Settings > Security > Anonymous Access and disable anonymous access.
2. When running `init`, provide an infrastructure file that contains the credentials like so:

```
"NuGetRegistryUsername": "", "NuGetRegistryPassword": ""
```

If provided, the NuGet restore calls will be authenticated. Note that currently there is no way to do the restore using NuGet ApiKeys.

#### NPM

1. login to your NPM repo with the build account (This is also done when publishing the packages but for the hosted repo. In this case the grouped repo must be used):

```
npm adduser --registry=http://example.local/repository/npm/
```

2. in the `.npmrc` file in your home folder, a new line should be added, in the format `//example.local/repository/`

```
npm/:_authToken=NpmToken.<GUID>
```

3. take this line and add it to the `.npmrc` in both the Help and HTML solutions. The files are always in the `apps/<web solution>` path
4. commit these files

NOTE: this file will be visible to anyone with source code access! If push permissions are not desired here, use an account that can only download packages!

In some cases `npm adduser` may fail. A known case is when using Personal Access Tokens (PATs) instead of passwords in Windows systems. In these cases, it's possible to directly use the credentials without this authentication step:

1. edit the `.npmrc` file in the HTML and Help package directories with your text editor
2. fill in the repository: `registry=https://example.io/repository/npm/`
3. fill in the credentials. The `_auth` string is obtained by doing a Base 64 encode of `<user>:<password>` with the respective username and password of the repository account: `//example.io/repository/npm/:_auth=<base64 string>`
4. tell NPM to always authenticate: `//example.io/repository/npm/:always_auth=true`

So if your username is `user1` and your password is `secret2` the `.npmrc` files would look like

```
registry=https://example.io/repository/npm/
//example.io/repository/npm/:_auth="dQBzAGUAcgAxADoAcwBLAGMAcgBIAHQAMgA="
//example.io/repository/npm/:always_auth=true
```

where the auth string can be generated in Powershell using

```
[Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes("user1:secret2"))
```

## 6.2.2 Process

### Host packages in local repository

If using a local repository, the dependency packages need to be hosted there. CMF will have provided a package with the NuGet packages. The NPM packages are extracted from the MES ISO file. The instructions are available either on your Help website or in CMF's general [Help repository](#).

However, currently the script provided in the Help tutorial only uploads the HTML solution dependencies, not the Help solution ones. As such, an updated copy of the script can be provided if requested.

It is recommended that you follow the Help tutorial, but a few quick steps will be detailed here

#### 1. Authenticate to your local repo:

```
npm adduser --registry=http://example.local/repository/npm-hosted/
```

a. NOTE: If using Nexus OSS, you will have to upload to the `npm-hosted` repo, not to `npm`

#### 2. Run

You only need to specify the Username if you don't have anonymous access to your repository

```
.\LoadNPMPackagesToLocalRepository.ps1 -PathToISO "<ISO path>" "http://example.local/repository/npm-hosted/" -ExtractionFolder "C:\_CMTemp\packages" [-Username <User>]
```

a. the script will automatically mount the ISO, extract the packages and upload them to the repository

3. OPTIONAL: tag the loaded packages with the dist-tags used by the solutions. If this is not desired, the scaffolded solutions will have to be changed to point to the exact versions that are uploaded into the repository. Adding the dist-tags is generally easier. The dist-tags always follow the format `release-<version>` with no dots. As an example, for MES version 8.1.0 the dist-tag commands would be:

```
npm dist-tag add @criticalmanufacturing/mes-ui-web@8.1.0-202103302 release-810 --registry=http://example.local/repository/npm-hosted/
npm dist-tag add cmf.docs.web@8.1.0-20210329.4 release-810 --registry=http://example.local/repository/npm-hosted/
```

## 6.2.3 Loading the scaffolding into the target project

### Using a local repository

If using a local repository, the first step is to load the NuGet dependencies. Until this is done the Business and Test solutions will not compile.

Place the packages in a folder. Open a powershell session in `Tools` and run the `Deploy_LoadAllNuGetPackages.ps1` script. Don't forget to authenticate first.

```
cd <folder>
nuget setapikey <nuget key> -source http://example.local/repository/nuget-hosted

# for each nuget file:
nuget push <file.nupkg> -source http://local.example/repository/nuget-hosted/
```

### Committing the code

If you did not run `init` and `new` inside a cloned git repository folder, you can add it as a remote in place. For the second approach, open a PS session where you ran the `init` and run:

```
git remote add origin https://<user>@dev.azure.com/<organization>/<project>/_git/<repository>
git push -u origin --all
```

### Loading pipelines and policies

To run the pipelines, some tasks need to be installed in Azure DevOps. These are:

- MSPPremier.PostBuildCleanup.PostBuildCleanup-task.PostBuildCleanup
- getza.replacetokens

You can now load the Pipelines and Policies into your project.

## 6.3 Feature package scaffolding

A traditional project structure targets the scenario where one team does all of the development for a project with a single target, e.g. a customization team deploying MES to a single factory or a set of factories with the exact same customization.

However, some projects are more complex. A single team can target a multi-site deployment in which the sites target a set of common features, but complement them with site-specific features, or the common features need site-specific data packages to configure them.

In these cases, projects are usually structured in a **Feature Packages** arrangement, in that some packages are self-contained, or in a **Layered Project** arrangement, in which e.g. a Site package, which is specific to a given site/plant, depends on a Core/Baseline package, which is common among all sites.

`cmf new` allows assembling these package in whatever structure is necessary, with a few limitations: - the test solution is always at the repository level, i.e. we do not have feature-level test packages - you cannot mix traditional packages, which exist in the repository root, with these packages - a feature package must always exist inside a global repository

`cmf new` will automatically register each package as a dependency of its parent (as per the folder structure).

### 6.3.1 Initialize the Repository

In case we are creating the first package in the repository, we need to initialize it in the same way as we do for a traditional project. Check the **Initialize the repository** section of the [traditional scaffolding instructions](#).

### 6.3.2 Creating a Feature Package

A **Feature Package** is a metapackage that can include one or more layer packages, e.g. we can have a Feature package that includes only a Business package.

A Feature package should include all necessary layer packages (business, UI, Help) needed for the Feature to work.

To create a new Feature Package, run the `new feature` command at your repository root, specifying the new package name: e.g. to create the Baseline package for your project, run:

```
cmf new feature Cmf.Custom.Baseline
```

The new feature package will be available in your repository at `Features\Cmf.Custom.Baseline`.

After creating a Feature package, it is not longer possible to create new layer packages in the repository root. Any `new` command for a layer package will fail:

```
> cmf new html
Cannot create a root-level layer package when features already exist.
```

### 6.3.3 Creating Layer packages in a Feature package

Creating layer packages works exactly in the same way as for a [traditional project](#), but the `cmf new` command should run from inside the feature package:

```
> cd Features/Cmf.Custom.Baseline
> cmf new business
The template "Business Package" was created successfully.
```

The layer package name will include the feature package name fragment to distinguish it from other packages in the same repository.

## 6.4 Infrastructure config file

---

### 6.4.1 Structure

Information regarding repositories and Azure projects usually don't change very often. For this set of information, `init` accepts a file with a few keys that specify:

1. The NPM repository URL
2. The NuGet repository URL
3. The Azure DevOps collection URL
4. The Azure DevOps project name
5. The type of Azure build agents used, Cloud or Hosted
  - a. Cloud means Microsoft hosted agents with can run a multitude of VMs
  - b. Hosted means self-hosted agents. Using self-hosted, for now, requires Windows agents with a set of pre-requisites installed.

### 6.4.2 Usage

The infrastructure file must be passed to the `init` command as an argument, e.g.:

```
cmf init --infrastructure <file> --config...
```

### 6.4.3 Example

```
{
  "ISOLocation": "path/to/MES/isos",
  "NPMRegistry": "http://local.example/repository/npm",
  "NuGetRegistry": "https://local.example/repository/nuget-hosted",
  "NuGetRegistryUsername": "user",
  "NuGetRegistryPassword": "password",
  "AzureDevopsCollectionURL": "https://azure.example.com/Projects",
  "AgentPool": "Projects",
  "AgentType": "Cloud"
}
```

## 6.5 Migration Paths

---

### 6.5.1 To 3x

---

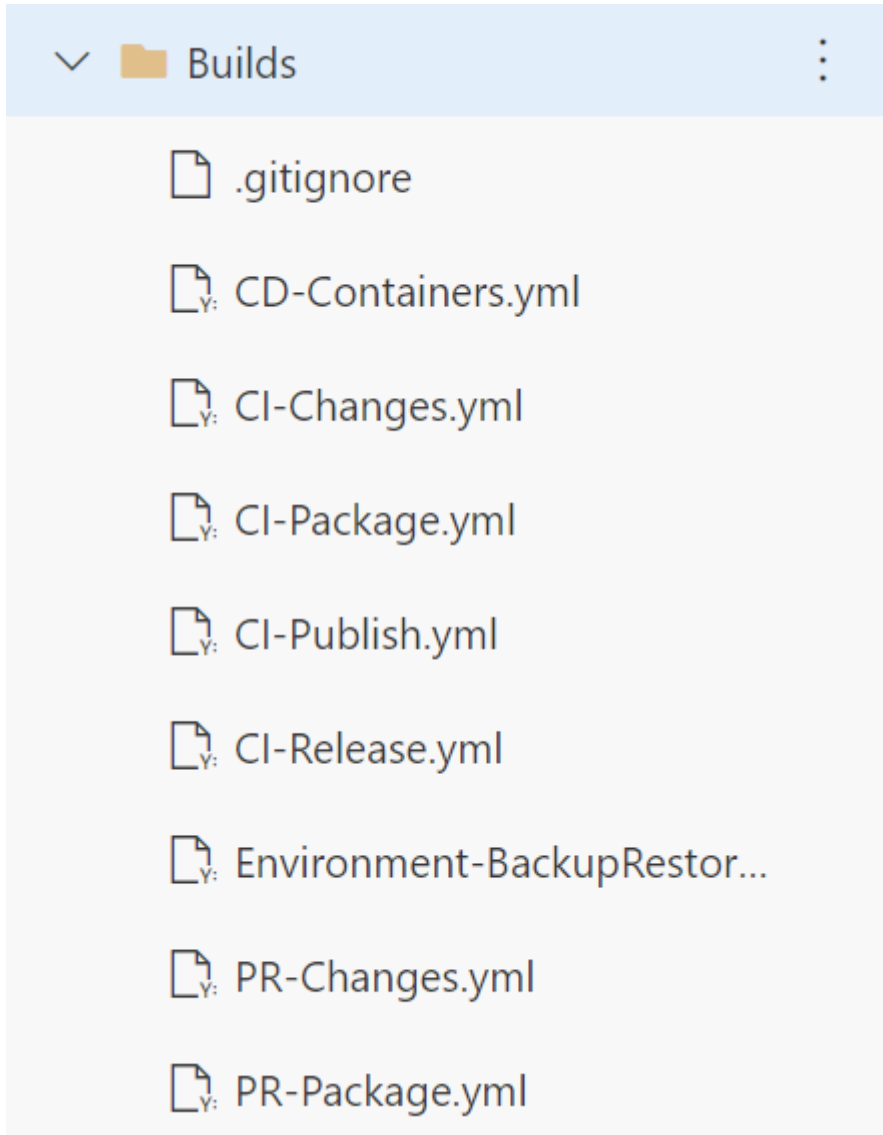
Default Steps for Business and Data Packages are now inject by the cli. If you are upgrading to cmf-cli 3x you **should remove** all the **steps** from the following packages:

- Business
- Data
- IoTData
- Tests MasterData

## 6.6 Pipeline Import

---

To import the Continuous Integration pipelines, first make sure you have scaffolded your project successfully and can see a folder in the project root named Builds with several YAML files in it:

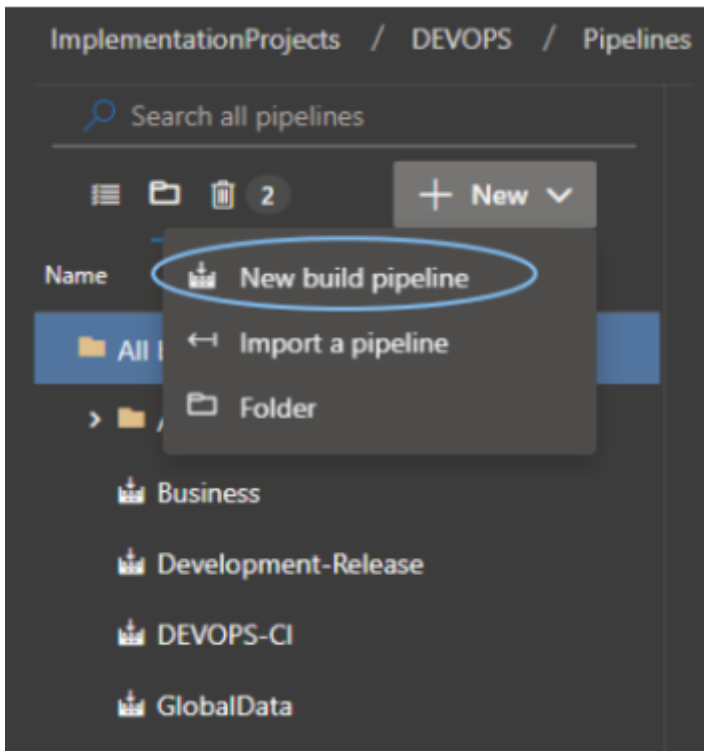


If so, we can proceed. Push your repository if you haven't yet. It is important that the default branch contains the pipeline YAML files. In our setup, the default branch is always **development**.

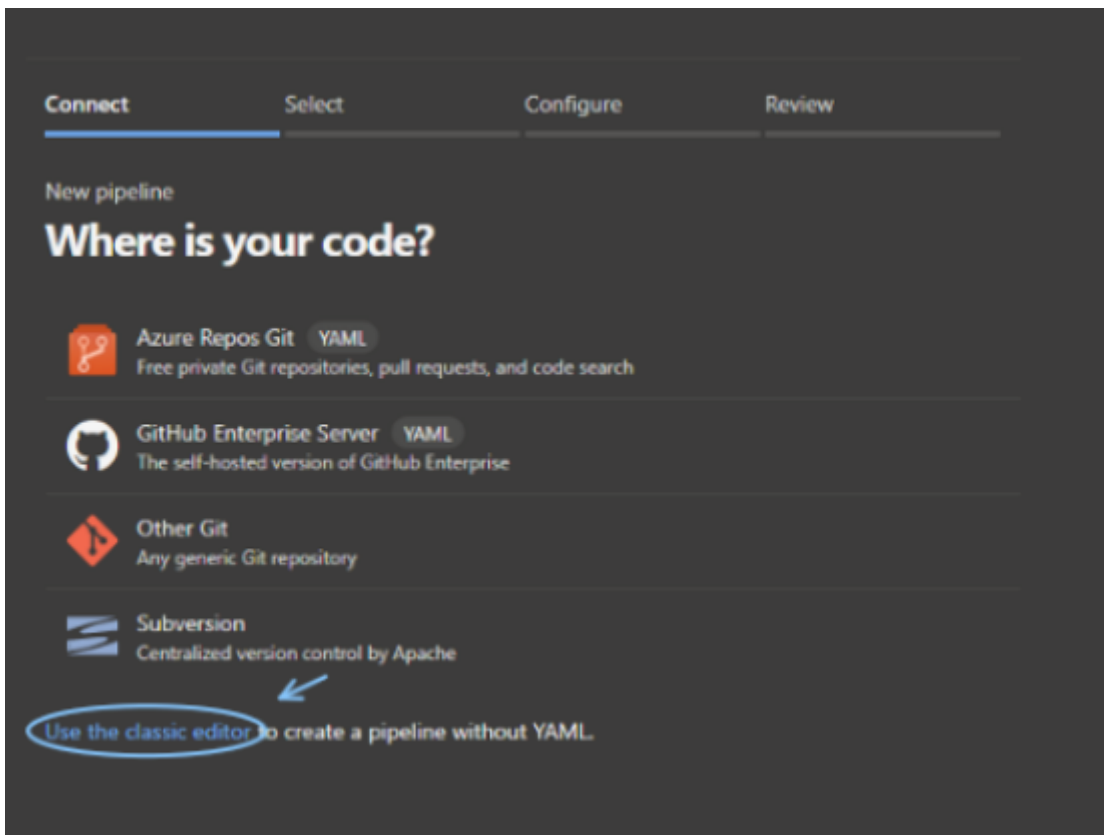
For each of **PR-Changes**, **PR-Package**, **CI-Changes** and **CI-Package**, repeat these steps:



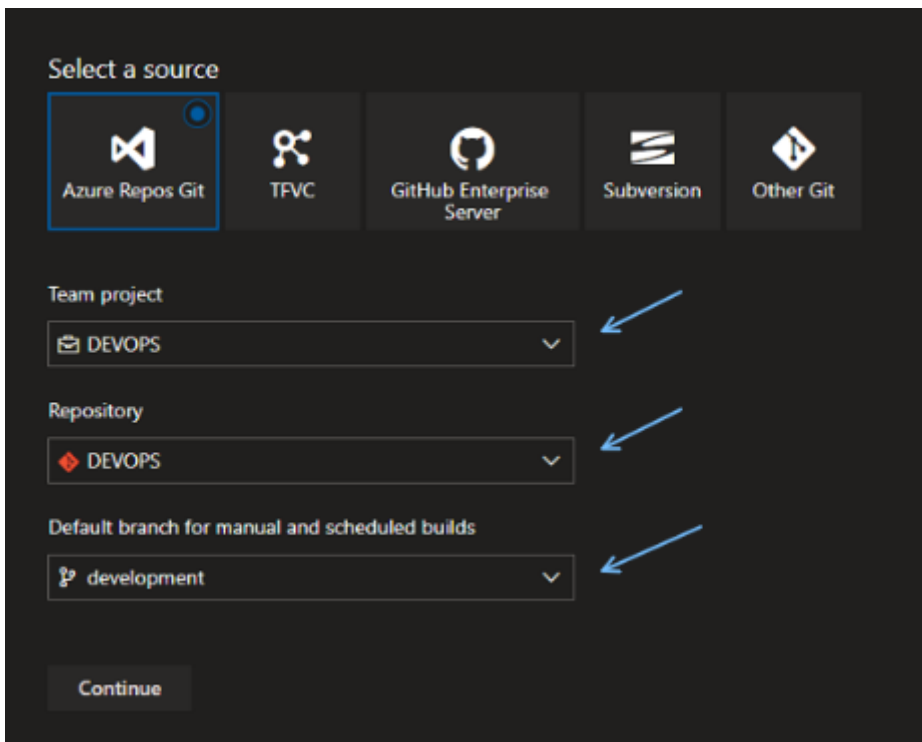
1. create a new pipeline



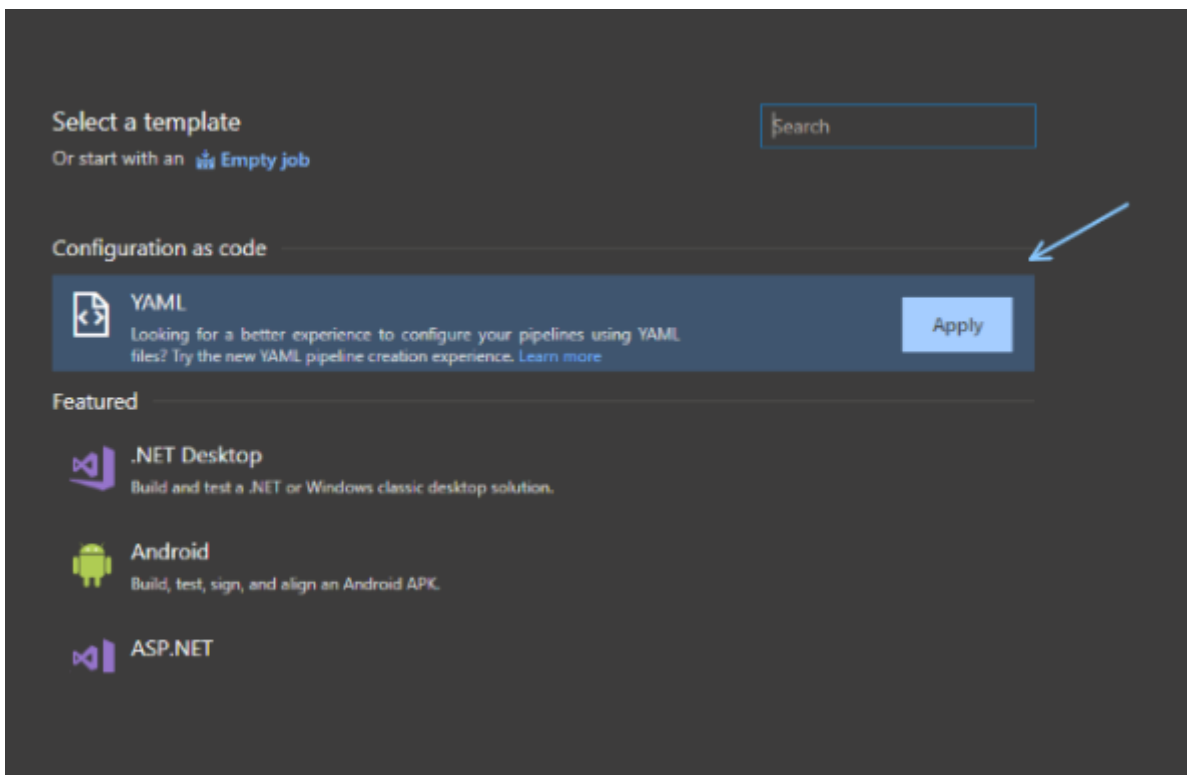
2. Choose **Use the classic editor**



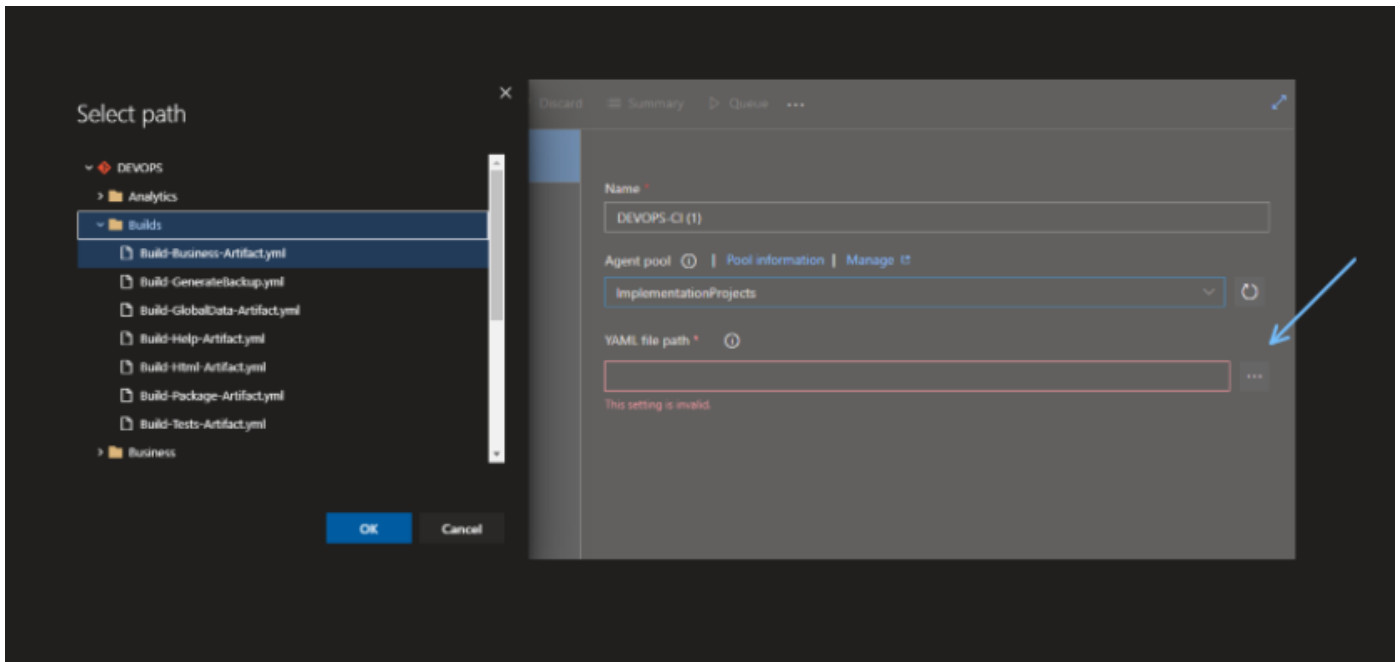
3. Choose your Team Project, repository and default branch. Our generated pipelines are expecting the default branch to be **development**



4. After that, choose Configuration as code > YAML



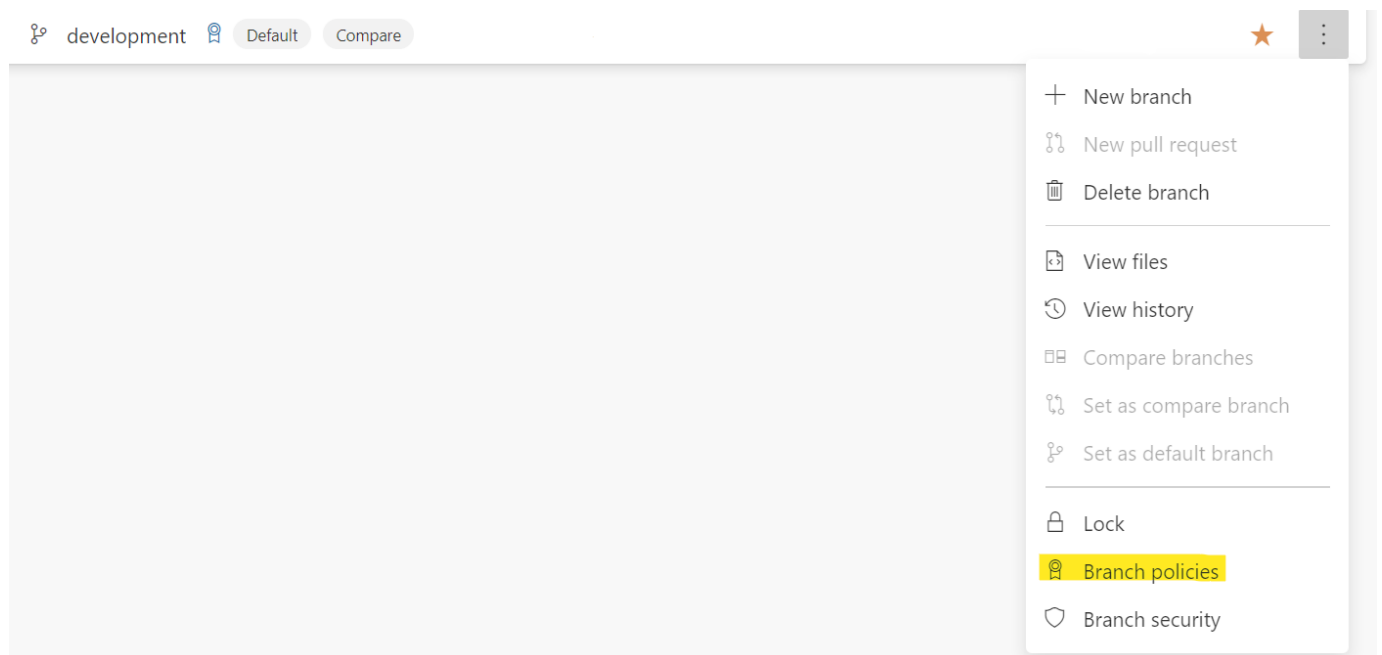
5. and choose the corresponding YAML file in the Builds folder



### 6.6.1 Pull Request policies

To effectively leverage the Pull Request flow our pipelines enable, Azure DevOps must be configured to protect the default (in our case, **development**) branch.

Go to Repos > Branches, on the **development** row, click the More button on the far right and select Branch Policies



The policy options per-se are dependent on the team configuration and working mode. Here is an example of our policies

## Protect this branch

- Setting a Required policy will enforce the use of pull requests when updating the branch
- Setting a Required policy will prevent branch deletion
- Manage permissions for this branch on the [Security page](#)

### ☒ Require a minimum number of reviewers

Require approval from a specified number of reviewers on pull requests.

Minimum number of reviewers

- ☐ Allow users to approve their own changes.
- ☐ Allow completion even if some reviewers vote "Waiting" or "Reject".
- ☐ Reset code reviewer votes when there are new changes.

### ☒ Check for linked work items

Encourage traceability by checking for linked work items on pull requests.

#### Policy requirement

- ☒ Required  
Block pull requests from being completed unless they have at least one linked work item.
- ☐ Optional  
Warn if there are no linked work items, but allow pull requests to be completed.

### ☒ Check for comment resolution

Check to see that all comments have been resolved on pull requests.


#### Policy requirement

- ☒ Required  
Block pull requests from being completed while any comments are active.
- ☐ Optional  
Warn if any comments are active, but allow pull requests to be completed.

### ☐ Limit merge types

Control branch history by limiting the available types of merge when pull requests are completed.

The important point here is to set the **PR-Changes** pipeline to run whenever a Pull Request is opened. To do so, add a new row in **Build validation** that looks like this

Build pipeline	Requirement	Path filter	Expiration	Trigger	
 PR-Changes	Required	No filter	Strict expiration	Automatic	<input checked="" type="checkbox"/> Enabled


This will ensure the PR pipelines will run each time code is pushed to a Pull Request, and will block merging if the build is failing.

## 6.6.2 Remaining pipelines

We do not support running the Continuous Delivery pipelines in a client setting. These pipelines are strongly opinionated on the environment in which they run and require virtual machines configured specifically for them. They are still provided as an example and guide so that each client or partner can create their own pipelines. However, they will not work out of the box and will hang on permissions issues if attempted to run.

## 6.6.3 Secrets

The **CD-Containers** pipeline requires that a secret is specified in a variable group to run successfully. The most important one is the Customer Portal PAT (Personal Access Token).

Library >  Docker Variables\*

Variable group |  Save  Clone  Security  Help

6

### Properties

Variable group name


Docker Variables

2

Description

☒ Allow access to all pipelines

3

☐ Link secrets from an Azure key vault as variables 

### Variables

Name ↑

Value

CustomerPortalPAT

myPAT





4

5

+ Add

1. In Azure DevOps, go to Pipelines > Library and create a new Variable Group
2. Name the variable group "Docker Variables"
3. Check "Allow access to all pipelines"
4. Add a new variable named **CustomerPortalPAT** and paste your Customer Portal PAT as the value
5. Click the lock icon to turn the variable into a secret
6. Save the variable group

The pipeline will automatically use the variable group.

It is possible to add extra secrets or variables into this group, to preserve some secrets from the environment settings. Any token in the format e.g. `#aToken#` will be replaced with the value of the **aToken** variable.

## 6.7 Pipelines description

---

This page gives an overview of our pipeline design and working mode. For specific details, you can consult the generated pipelines, by running the `cmf init` command as detailed [here](#).

Please remember that these pipelines are tailored to CM's internal structure and are not expected to run unmodified in your infrastructure. They are provided as examples and should not replace your own pipelines adapted to your specific process.

### 6.7.1 Pull Request (PR) process

---

The Pull Request process runs each time a Pull Request is opened or code is pushed to the request branch. It is comprised of two pipelines.

#### PR-Changes

The PR-Changes pipeline calculates which packages are affected by which git commits. It does this by comparing the HEAD commit of each `cmfpackage.json` file in the repository between the PR branch and the target branch. For each changed package, a PR-Package pipeline run is triggered.

#### PR-Package

This pipeline builds the package by invoking `cmf build --test`, making sure the changed packages build and their unit tests pass. If this build fails, the Pull Request is blocked. The developer then has to correct the code, push it to the PR branch, and the process starting with PR-Changes will run again automatically to validate the code.

### 6.7.2 Continuous Integration (CI) process

---

Our CI process consists of 2 pipelines responsible for generating packages anytime our main branch changes.

#### CI-Changes

The CI-Changes pipeline calculates which packages are affected by which git commits. It does this by comparing the last built HEAD for that specific package. For each altered package, a run of CI-Package is triggered.

#### CI-Package

This pipeline builds the package specified by its parameters, which are filled in automatically by CI-Changes.

The build process happens exclusively using the `cmf` CLI tool. The pipeline invokes `cmf build --test`, which builds the package and runs its automatic test suite. If successful on both steps, it then invokes `cmf pack` to generate a package that can be installed via DevOps Center or Critical Manufacturing Setup. This package is then placed in the CI repository (as defined in the variable `CIPackages` in `GlobalVariables.yml`) and the current git HEAD is registered as the last built HEAD for this package.

The pipeline process has no bearing on the package name: each package name and version are specified in their manifest (`cmfpackage.json`) file. As such, when developing new functionality on a released package, don't forget to [bump](#) its version first, regarding the type of the changes to bump the patch, minor or major numbers. The PR process will not allow merging a package with an ID/Version pair which has already been approved for release.

#### CI-Publish

At least once a day, this pipeline runs to calculate which packages should be included in a release run for E2E testing.

It does this by calculating our current package tree, starting from root, and checking which of these packages are already present in the Deployed repository. If they exist there, they were already released and should not be considered release candidates. All remaining packages are release candidates and should be included in a release run. This calculation is performed entirely by the `cmf` CLI tool, using the `cmf assemble` command.

The release candidates are packed and published as a build artifact, as our Package artifact. As well, the environment configurations are packed into our Configurations artifact, and the relevant test packages are packed as our Tests artifact.

### 6.7.3 Continuous Delivery (CD) process

---

Our CD process doesn't actually deliver to client environments, but to our production-like Integration environments. It can be executed by one of two pipelines, depending if we're targeting a Windows machine or a Container installation. However, both pipelines implement different approaches to the same process:

1. Restore the environment to a previous known good state - this can be either the same MES and customization version currently running in Production at the client if we're building cumulative packages (not recommended) or the last sprint's delivered versions for client testing when building iterative packages.
2. Install Release Candidate packages, and their MES dependency if needed - perform the actual package installations.
3. Create an environment restore point - this restore point will be used the next known good state if these release candidates are approved for release.
4. Load test master data - load additional data that helps with testing.
5. Create the Daily backup - backup only the ONLINE database, for development use. This backup can be loaded locally in our development machines to support our local environments. It contains the test master data, which is useful for development tests.
6. Run E2E tests - run the tests that require the full environment to be installed. We run the Backend, Frontend and IoT tests separately, though this is a technicality.
7. Approval gate - this is a manual step. If the team is happy with this run and with the quality of the Release Candidate packages, this run can be approved and the release candidate packages can be released to the client.
8. Set Known Good State - this only runs if the run is approved. It will set the Restore Point created in step 3 as the known good state for the next run.

The pipelines are as follows:

### CI-Release (for Windows machine installations)

This pipeline needs to backup the entire MES installation. As such, both the database and application layers are backed-up. The tool most used in this process is the CmfdDeploy client, which is part of the Deployment Framework that ships in the Critical manufacturing disks/ISOs. The details of each step are:

1. restore - this uses the CmfdBackupRestore.Tools (installed via CmfdDeploy) to restore both the database and the application layer to the backed-up state.
  - a. Running CmfdDeploy uses Powershell remoting to mount the ISO file in the target machine and run the installation from there, as this is a requirement when changing the application layer. This is true for all steps.
 

Beware that the application layer part includes only the business tier, the HTML and Help sites. Which means these restores do not work if the MES version changes: if we upgraded MES and then restored to a backup in a previous version, we'd get a mixed environment that will most likely break.
2. Install the Release Candidate root package using CmfdDeploy. All already installed packages will be skipped, and only new packages will be installed.
3. Use CmfdBackupRestore.Tools to create a temporary restore point including all databases and application the 3 layer solutions (check the NOTE at step 1).
4. Load the test master data files (using CmfdDeploy). The test master data files are also DF packages, though not usually shipped to the client. They are installed the same way as the actual system upgrades.
5. Create the Daily backup - use CmfdDeploy with CmfdBackupRestore.Tools to backup only the ONLINE database
6. Run E2E tests - this is the only part that runs on the build server: it uses the VsTest azure task to run all E2E tests, from each layer in turn, Business, HTML, IoT.
7. Approval Gate
8. Set Known Good State - rename and move the backups so they are picked up by the next run

### CD-Containers

Unlike the Windows machine pipeline, this pipeline always reinstalls MES by using brand new containers. This makes the process more reliable and the backups quicker. The most used tool in this process is `portal-sdk`, a `cmf` CLI plugin that can be installed from `npm`.

1. restore - for containers, we don't actually restore anything, we simply drop all databases to reinstall MES
2. Installation - create a new version of our Integration stack by using the `portal-sdk` plugin. Set as target package the release candidate root package. This package depends on a specific MES version which will get installed first. This installation also uses the Known Good State backups to maintain the database contents as they were on the last approved release run.
3. Use CmfdDeploy to create a temporary database backup, by using the CmfdDeploy `backup` command.
4. Load the test MD by creating a new version of our environment using as target the Test MD package. If we have multiple test MD packages, we do this once for each one.
5. Create the Daily backup by the exact same process as step 3, but only backup the ONLINE database.
6. Run E2E tests - this is similar to the Windows machine scenario
7. Approval Gate
8. set known good state, by moving the backups created in step 3 (ONLINE, ODS and DWH) to a known location which the next run will use as a known good state. These backups can't be renamed, so we just keep them separate until approval.



## 6.8 Exported Objects package

---

Even though the CLI does not provide scaffolding for an `ExportedObjects` package, as it generally is better to include these in Master Data packages, it is possible to create such a package and the tool will pack it as with any other package type.

As an example, an `ExportedObjects` package manifest looks like this:

```
{
  "packageId": "Cmf.Custom.Baseline.ExportedObjects",
  "version": "1.0.0",
  "description": "Baseline Exported Objects Package",
  "packageType": "ExportedObjects",
  "isInstallable": true,
  "isUniqueInstall": false,
  "contentToPack": [
    {
      "source": "${version}/*",
      "target": "ExportedObjects"
    }
  ]
}
```

You would create a `cmfpackage.json` file inside the objects folder with this content. This will pack any XML file in a folder named after the current package version, so in this case `1.0.0` and place it in the package file in an `ExportedObjects` folder.

Afterwards, do not forget to add this new package as a dependency of your root/feature package, to make sure it gets installed when required.

## 6.9 Post-scaffolding package tailoring

The packages generated by `cmf new` are as neutral as possible, so to be compatible with as many deployment scenarios as possible. However, some tailoring is advised for specific targets. This tailoring allows the packages to better adapt to their deployment target, eliminating the need of manual steps when installing.

Below are some tailoring options for specific targets. It's recommended that these changes are applied according to your target environment.

### 6.9.1 Version 3x:

#### 6.9.2 Containers:

- **Tests MasterData Package**
- Add the property

```
"isToSetDefaultSteps": false
```

### 6.9.3 Version 2x:

#### 6.9.4 Containers:

- **Root Package**
- Add to dependencies

```
"dependencies": [
  { "id": "Cmf.Environment", "version": "8.3.0" }
]
```

- **Business Package**
- Add to steps:

```
"steps": [
  { "order": "1", "type": "DeployFiles", "ContentPath": "**/!(Cmf.Custom.*.BusinessObjects*).dll" }
]
```

### 6.9.5 Deployment Framework:

- **Root and IoT Root Package**
- Add to dependencies

```
"dependencies": [
  { "id": "CriticalManufacturing.DeploymentMetadata", "version": "8.3.0" }
]
```

- **Business Package**
- Add to steps:

```
"steps": [
  { "order": 1, "type": "Generic", "onExecute": "$(Agent.Root)/agent/scripts/stop_host.ps1" },
  { "order": 2, "type": "DeployFiles", "ContentPath": "**/!(Cmf.Custom.*.BusinessObjects*).dll" },
  { "order": 3, "type": "Generic", "onExecute": "$(Agent.Root)/agent/scripts/start_host.ps1" }
]
```

- **Data, IoTData and Tests MasterData Package**

- Add to steps:

```
"steps": [
  { "order": 1, "type": "Generic", "onExecute": "$(Agent.Root)/agent/scripts/stop_host.ps1" },
  { "order": 2, "type": "TransformFile", "file": "Cmf.Foundation.Services.HostService.dll.config", "tagFile": true },
  { "order": 3, "type": "Generic", "onExecute": "$(Agent.Root)/agent/scripts/start_host.ps1" },
  { "order": 4, "type": "DeployFiles", "ContentPath": "GenerateLB0s.ps1" },
  { "order": 5, "type": "Generic", "onExecute": "$(Package[Cmf.Custom.Package].TargetDirectory)/GenerateLB0s.ps1" }
]
```

**NOTE:** Make sure that the order of the steps is the same referenced in this document.

## 6.10 Traditional scaffolding

A "traditional project" does not contain [feature packages](#), is developed entirely by one team in one repository and is delivered directly to one customer.

These projects are usually composed of Business, UI, Help and Master Data customization, with optionally Exported Objects and IoT.

The objective is to obtain a structure equivalent to what `solgen` provided.

Please consult each commands help page for details of what each switch does.

### 6.10.1 Initialize the repository

These types of projects usually fully own their git repository and as such need to be initialized to obtain the base repo structure, as well as the build pipelines if we are targeting Azure DevOps.

This is done with the `cmf init` command:

#### Classic      Containers

```
cmf init Example `
  --infra ..\COMMON\infrastructure\CMF-internal.json `
  -c Example.json `
  --repositoryUrl https://tfs.criticalmanufacturing.com/Projects/Test/_git/Test `
  --MESVersion 8.2.1 `
  --DevTasksVersion 8.2.0 `
  --HTMLStarterVersion 8.0.0 `
  --yoGeneratorVersion 8.1.1 `
  --nugetVersion 8.2.1 `
  --testScenariosNugetVersion 8.2.1 `
  --deploymentDir \\vm-project.criticalmanufacturing.com\Deployments `
  --ISOLocation \\setups\CriticalManufacturing.iso `
  --version 1.0.0
```

```
cmf init Example `
  --infra ..\COMMON\infrastructure\CMF-internal.json `
  -c Example.json `
  --repositoryUrl https://tfs.criticalmanufacturing.com/Projects/Test/_git/Test `
  --MESVersion 8.2.1 `
  --DevTasksVersion 8.2.0 `
  --HTMLStarterVersion 8.0.0 `
  --yoGeneratorVersion 8.1.1 `
  --nugetVersion 8.2.1 `
  --testScenariosNugetVersion 8.2.1 `
  --deploymentDir \\vm-project.criticalmanufacturing.com\Deployments `
  --ISOLocation \\setups\CriticalManufacturing.iso `
  --version 1.0.0 `
  --releaseCustomerEnvironment EnvironmentName `
  --releaseSite EnvironmentSite `
  --releaseDeploymentPackage \@criticalmanufacturing\mes:8.3.1 `
  --releaseLicense EnvironmentLicense `
  --releaseDeploymentTarget EnvironmentTarget
```

`EnvironmentTarget` can take any value recognized by the Portal SDK, which can be found [here](#).

Note: The ``` character escapes multiline commands in `powershell`. For bash, the `\` character does the same thing.

The infrastructure file specifies the repositories to be used to get the project dependencies. If you are scaffolding a Deployment Services project, there is a CMF-internal.json infra file which specifies our internal infrastructure. It's in the **COMMON** project, **Tools** repository, at `/infrastructure`. If scaffolding a customer or partner project, you will need to create this infrastructure file first. Check [Infrastructure](#) for more details.

As in previous scenarios, the versions for the various input options must be previously determined. Unlike with `solgen`, `cmf init` will not assume default/current values for these options.

This will also create a root package, which may or may not be shipped to the customer. Unlike with `solgen`, this root package has no dependencies, initially. Each time a layer package is created, it will be registered in the higher level package found. For a traditional repository, this will be the root package.

If you are using version `cmf-cli` version 2x, follow the instructions defined in the [Post-scaffolding package tailoring](#). You will not be able to generate the layer packages before doing this. In version 3, this is already done by the CLI.

## 6.10.2 Layer packages

Each application layer is deployed in a different package. This allows the team to deliver only what was actively modified during a sprint, and keep the previous versions of the unchanged layers in an installed system.

### Business

The business package is straightforward and is generated with the `cmf new business` command:

```
cmf new business --version 1.0.0
```

This creates a [.NET](#) solution for backend development. Please note that unlike with `solgen`, the Actions project is not included in the business solution.

### Master Data

The Master Data package includes also the Exported Objects. Unlike in a `solgen` solution, Exported Objects are loaded via Master Data and not using a specific ExportedObjects sub-package.

As the Master Data package also includes the Process Rules, it can optionally register the Actions package in a specific Business solution. For the traditional scenario, the command would be:

```
cmf new data --version 1.0.0 --businessPackage .\Cmf.Custom.Business\
```

### UI

The UI and Help packages are also scaffolded differently from a `solgen` project. To fully scaffold these solutions, the corresponding Deployment Framework package needs to be specified. These can be found in the MES ISO/disk. Make sure you use the correct version: a mismatch may cause all kinds of problems when running.

The corresponding commands are:

#### HTML

```
cmf new html --version 1.0.0 --htmlPackage H:\packages\Cmf.Presentation.HTML.8.2.1.zip
```

If you require NPM registry authentication, the current procedure is to include the auth information in the `apps\customization.web.npmrc` file as is standard.

#### HELP

```
cmf new help --version 1.0.0 --documentationPackage H:\packages\Cmf.Documentation.8.2.1.zip
```

### IoT

The IoT package contains both `IoTData` and `IoTPackages` as sub-packages. They are always created together.

```
cmf new iot --version 1.0.0
```

## Database

The database package contains both Pre, Post and Reporting sub-packages.

```
cmf new database --version 1.0.0
```

## Tests

The tests package is generated and built like any other layer package, but it is not installable. It is also not usually delivered to customers, unless requested.

```
cmf new test --version 1.0.0
```

## 6.10.3 Demo

---

This demo show the usual initial setup for a new project. For the first sprints, which focus heavily on modelling, the Data package is of the most importance. Obviously a Tests package is also needed. As an extra, the Business package is also initialized. This allows the Process Rules in the Data package to have a .NET solution where they can be compiled for checking.

Note that the GIF is quite large and can take a bit to load.

