
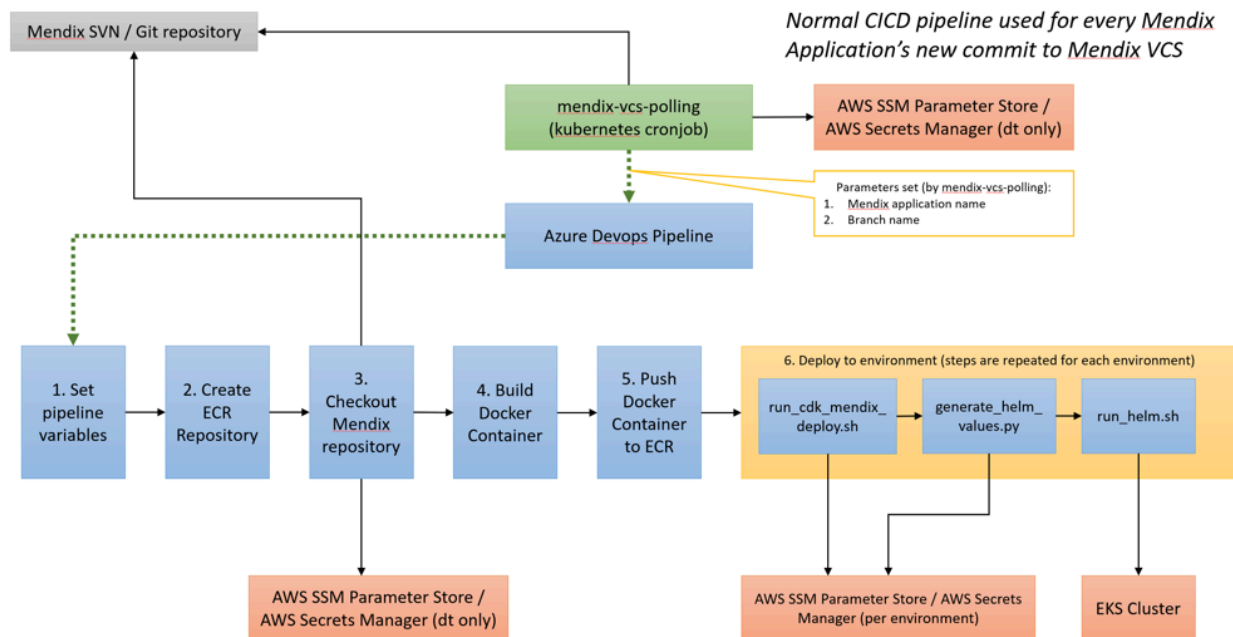


## Normal pipeline run

This page describes the details of a normal pipeline run. A normal pipeline means the normal workflow that is Continuous Integration capable; a Mendix application developer makes a change, commits this to the Mendix VCS repository, and this commit needs to be built and deployed to the dev/test environment automatically. This means that their change should become visible in the dev/test environment in a reasonable timeframe without any human interaction.

In addition, for deployment to acceptance and production an approval is required by an authorized individual to authorize changes going to acceptance and production. This is, under normal circumstances, the only human interaction required for this workflow to work. Let's go over each step of the drawing below in detail. While this normal pipeline run is usually an automated process triggered by `mendix-vcs-polling`, you can run this manually as well.

 A normal pipeline run in this context means the combination of the build and the release pipeline. It does not represent a single physical pipeline.



This drawing contains both the build and release pipeline in a schematic, workflow-like fashion. It does not represent a physical pipeline

## Phase 1: Developer commit and triggering of the pipeline

As you may have read in other pages, the `mendix-vcs-polling` is a script that runs as a Kubernetes cronjob and polls every activated Mendix application (meaning: the migration has completed and registered in the SSM parameter store under `/dsm/mendix-vcs-polling`) if there are changes. It does this in the following fashion:

1. Use the SSM Parameter Store API to retrieve a list of Mendix applications found in the `/dsm/mendix-vcs-polling` parameter. This is a comma-separated list of names.
2. For each name found, it retrieves the stored commit hash / revision by querying the `/dsm/mendix/apps/<app_name>/vcs/revision` parameter, the URL found in `/dsm/mendix/apps/<app_name>/vcs/url` and the branch to check found in `/dsm/mendix/apps/<app_name>/vcs/branch`
3. It then executes either `svn info` or `git ls-remote` to determine upstream changes
4. If there are any changes, call the Azure DevOps REST API to trigger the “Mendix Generic CICD Pipeline” pipeline (pipeline\_id 428). It sets the parameters of the pipeline to the name of the app found in step 1, and the name of the branch found in step 2.
5. Repeat steps 2 to 4 for every application found in step 1

This process runs every 5 minutes and can therefore take a little time for the pipeline to be triggered.

## Phase 2: Start the pipeline and build the container

Now that the pipeline is running, we follow the green dotted line onto the first step in the pipeline. Let's go over them:

1. Set pipeline variables. This is mostly an initialization step of the pipeline to register Azure Devops Task variables, as well as changing the build name so that the pipeline becomes uniquely identifiable. No actual work is being done here, however it is a very important step as the rest of the pipeline assumes that everything is set correctly here.
2. Create ECR repository. This runs on the AWS build account and uses the `cdk deploy` command to synthesize and deploy the Cloudformation stack for this ECR repository. This will create an ECR repository in the build account with the right resource policy set so that all other accounts can pull from this ECR repository. Effectively this step would only have to be done once create this, but is kept in the pipeline for both idempotency sake as well as having the ability to update anything with regards to the ECR repo (think resource policy changes, for example)
3. Checkout mendix repository. This uses the `checkout_mendix_repo.py` script found in the *kubernetes-mendix-deployment* repository. Basically it retrieves the `/dsm/mendix/apps/<app_name>/vcs/url` parameter from SSM, based on the URL it performs a `git clone` or `svn checkout`. One additional feature of this script is updating the pipeline name to the checked out SVN revision / Git commit hash so in the list of pipelines you can see which revision / commit hash is used. This also sets additional Azure Devops variables like the Mendix Runtime version to pass it along to the next step.
4. Build Docker container. This executes the `docker build` command with input from the pipeline variables set in previous steps and builds the container. This is expected to take a few minutes.
5. Push Docker container to ECR. This step also does a login and a logout of ECR because the ECR credentials helper is not setup on the Microsoft hosted build agent pool.

- ⓘ If you run this pipeline as a manual run instead of an automated trigger by mendix-vcs-polling, you can optionally specify a revision number / commit hash to do a build and push of that specific revision / hash.

## Phase 3: Deploy to each environment

We have arrived at the yellow box in the drawing, which represents the release pipeline. The steps in this box are repeated for each environment.

1. First of all we run `run_cdk_mendix_deploy.sh`. This is a wrapper script around the `cdk` command to support the migration pipeline (it will log that it runs in normal mode vs migration mode). This step connects both to the SSM parameter store (because the CDK utilizes the parameter store) and the Secrets Manager to retrieve the database password.
  - a. This is basically done in two steps in two separate Cloudformation Stacks; the database password is separate so that CDK can manage and rotate database passwords (in the future). As we utilize an existing Lambda to update RDS database passwords and create RDS users, this must be using Cloudformation parameters to prevent leaking the password. The only way to do this in CDK is by passing it as a parameter on the command line. Not the prettiest solution but the most practical solution as this mimics the current process as well.
  - b. The second stack is all of the other application infrastructure. This contains the IAM user (and role for apps that support it), database + credentials creation, bucket, SSM parameters and ACM certificate. As the database password must come from the different stack, it uses the AWS CLI to pass the CF parameter.
2. Then we run `generate_helm_values.py`. This script is actually called from `run_helm.sh` but logically this script runs first. It uses the convention of the Helm parameters that live in `/dsm/mendix/apps/<app_name>/config` to generate the `values_generated.yaml` that contain all of the overridden values. This also connects to the Secrets Manager to retrieve passwords, Mendix license key etc. This all ends up in the previously mentioned yaml file. Any parameter not set will use the defaults as found in `values.yaml` in the repository. Refer to [Helm chart - MAAS Team Space - Confluence \(atlassian.net\)](#) for more info on this.
3. `run_helm.sh` runs which is a small wrapper script around the `helm` command as well as ensuring that `generate_helm_values.py` runs first. If the latter does not run the application will deploy with all default values which will not work. Helm will render the templates with the provided values and connects to the control plane to perform an equivalent of a `kubectl apply`.

At this stage the deployment is complete. If this is acceptance or production, approval must be given first before it moves on to the next stage in the pipeline.