
Infrastructure maintenance automation

# Contents



# Infrastructure maintenance automation

## Introduction

This document describes the goals and the approaches for automating the management of the infrastructure used by Apertis. It will focus in particular on release branching since the new release flow implies that Apertis will need to go through that process two or three times more than in the past on each quarter.

## Goals

### Data-driven

Separating the description of the desired infratructure state from the tools to apply it nicely separates the two concerns: in most cases the tools won't need to be updated during branching, only the desired infrastructure state changes.

### Git-controlled

Basing everything on configuration stored in a Git repository has several advantages:

- all the changes are tracked over time

- the standard Apertis workflows based on GitLab merge requests can be used to review changes
- fine access controls can be configured via GitLab

### Idempotent

Every tool should compare the current state with the desired one and not produce errors when they already match. Administrator should be able to run the tools at any time, multiple times, without any ill effect.

### Scalable

The Apertis infrastructure is composed of enough services that a centralized list of things to update when branching is doomed to be outdated every quarter.

### Single source of truth

The duplication of the same information between modules should be minimized, such that updating the single source of truth automatically produces effects on the depending modules.

### Reproducible

Running the tools in a standardized, easily reproducible environment enables all the administrators to easily deploy changes without any special setup.

### Explicit

All the needed information should be explicitly encoded in metadata repository. The tools using it should strive to not make any assumption on the data and derive more information out of it. This is another facet of ensuring that the metadata repository remains the single source of truth.

## Basic approach

The basic approach aims at improving the current branching scripts to make them easier to test by developers, enabling more people to work on them, and to extend them to fully handle the complete branching process.

### Add test mode for current branching scripts

At the moment the quarterly release branching is done through a set of scripts[1] that get invoked manually by one the Apertis infrastructure team member from their machine.

They act directly on the live services using the caller's accounts.

[1] https://gitlab.apertis.org/infrastructure/apertis-infrastructure/tree/apertis/v2020pre/release-scripts

The first step for improving the branching automation is to add a "dry-run" mode to all the current release scripts to let developers and admin run them

### Improve coverage of current branching scripts

The scripts currently in charge of reducing the manual intervention during the branching process do not cover all services and repositories which are part of Apertis.

Once the "dry-run" mode is in place, new steps need to be added to the branching scripts to cover the missing services and repositories.

## Longer term approach

Larger refactorings are needed to align the current infrastructure to the goals previously described.

The following sections describe the steps needed to further improve the infrastructure maintenance to make it more robust and require less effort to manage.

### Centralized metadata

A new git repository contains the principal metadata about the whole Apertis infrastructure describing:

- the currently active release branches
  - e.g. `v2020pre`, `v2019`, etc.
- their phase in the release lifecyle
  - e.g. `development`, `preview`, `stable`
- their release status
  - e.g. `frozen`, `release-candidate`, `released`
- the release from which they get branched from:
  - e.g. `2019pre` for both `v2019` and `v2020dev0`
- the matching git branch name
  - e.g. `apertis/v2019`
- the APT components they ship
  - e.g. `target`, `development`, `sdk`, `hmi`
- etc.

This provides a git-controlled single source of truth: tools are updated to fetch the information they need from this repository.

For instance, the creation of OBS projects should be handled by a tool that:

- fetches the above YAML
- checks the current OBS configuration
- computes the changes needed compared to the desired state, if any
- applies the changes, if any, to align the actual state to the desired state, providing an idempotent solution

- runs from a GitLab pipeline, providing a reproducible environment that can be either triggered by changes in the main data repository or manually

The current infrastructure encodes a lot of information about the releases in several places: tools should be changed to fetch such information on the fly from the main data repository or GitLab pipelines should be configured to monitor the main data repository and automatically apply changes accordingly.

For instance, the LAVA job templates encode the branch name of the release they track in multiple places. Either the templates can be enhanced to fetch the information on the fly from the main data repository, or a pipeline should be configured in a dedicated branch in the repository to monitor the main data repository and branch/update the repository accordingly.

The change compared to the current approach is to minimize the amount of information that needs to be branched and distribute the branching logic closer to the entity to be branched. This is meant to avoid the recurring issues where the current centralized branching scripts failed to branch things properly or did not include new components to be branched at all.

### Per-repository branching operations

For most repositories it is sufficient to add a new git ref when branching for a new release. In particular, nearly all the the packaging ones do not need any change to the repository contents and creating a new ref is enough.

Other repositories need instead some changes to be applied to the contents once a new release branch is created. A common reason is that the release name is encoded in some file, which means that the file needs to be updated and the change needs to be committed and pushed.

By making branching self-contained in the repositories, moving and renaming them no longer cause breakage. It also gives full control over the branching of a repository to the people managing that repository, rather than those who manage the centralized repository. This can be especially useful for components not managed by the core Apertis team, owned instead by product-specific teams.

In general, keeping the branching operation in the same place as the rest of the contents helps in keeping them coeherent and makes testing easier.

## Implementation

### Add test mode for current branching scripts

Setting the `NOACT=y` environment variable causes the branching scripts to run in test mode, without actually launching the branching commands.

### Improve coverage of current branching scripts

New actions need to be taken when branching a new release.

This is a non exhaustive list:

- branch LAVA job templates;
- update the configuration on GitLab repositories to create the new release branch, make it the default, etc.;
- create the relevant `:snapshots` repositories on OBS;
- add support for the `security`, `updates` and `backports` repositories when branching stable releases.

### Centralized metadata

The centralized information can be modeled as YAML, for instance:

```
.common_components: &common_components
  - target
  - development
  - sdk
  - hmi

projects:
  apertis:
    releases:
      v2019:
        lifecycle: stable
        status: released
        branched-from: v2019pre
        branch-name: apertis/v2019
        upstream: debian/buster
        obs-build-suffix: v2019.0
        suites:
          v2019:
            obs-pattern: '$project:$release:$component'
            components: *common_components
          v2019-updates:
            obs-pattern: '$project:$release:updates:$component'
            components: *common_components
          v2019-security:
            obs-pattern: '$project:$release:security:$component'
            components: *common_components
        infrastructure-packages:
          obs: apertis:infrastructure:v2019
          suite: infrastructure-v2019
          components:
            - buster
      v2020dev0:
        lifecycle: development
        status: frozen
        branched-from: v2019pre
        branch-name: apertis/v2020dev0
        upstream: debian/buster
        obs-build-suffix: v2020dev0
        suites:
          v2020dev0:
            obs-pattern: '$project:$release:$component'
            components: *common_components
        infrastructure-packages:
          obs: apertis:infrastructure:v2019
          suite: infrastructure-v2019
          components:
            - buster
```

### Per-repository branching operations

A `release-branching` step should be added to the GitLab CI pipeline YAML in the repository with the purpose of ensuring that the release-specific contents match the branch name.

GitLab does not provide any way to execute an action only when a new ref is created so the best strategy is to ensure that the `release-branching` script is idempotent and gets run when changes land to any `apertis/*` refs: if no changes are detected the step succeeds with no further operations, otherwise it commit and push the changes automatically, or it submits a MR to be reviewed before landing the changes.