



Long term reproducibility

1	<b>Contents</b>	
2	<b>Background</b>	<b>2</b>
3	<b>Apertis artifacts and release channels</b>	<b>3</b>
4	Reproducible build environments . . . . .	5
5	Build recipes . . . . .	6
6	Packages and repositories . . . . .	7
7	External artifacts . . . . .	8
8	Main artifacts and metadata . . . . .	9
9	Package builds . . . . .	10
10	<b>Recommendations for product teams</b>	<b>10</b>
11	<b>Implementation plan</b>	<b>11</b>
12	Snapshot the package archive . . . . .	11
13	Version control external artifacts . . . . .	11
14	Link to the tagged sources . . . . .	11
15	<b>How to reproduce a release build and customize a package</b>	<b>11</b>
16	Reproduce the build . . . . .	11
17	Customizing the build . . . . .	12
18	<b>Example 1: OpenSSL security fix 2 years after release v1.0.0</b>	<b>12</b>
19	Getting started with Apertis: one year before release 1.0.0 . . . . .	13
20	Creating the list of golden components: the day of the release 1.0.0 . .	14
21	Using the golden components two years after release 1.0.0: Creating	
22	the new release . . . . .	16
23	Reproduce the build . . . . .	16
24	Customizing the build . . . . .	17

## 25 Background

26 One of the main goals for Apertis is to provide teams the tools to support their  
27 products for long life cycles needed in many industries, from civil infrastructure  
28 to automotive.

29 This document discusses some of the challenges related to long-term support  
30 and how Apertis addresses them, with particular interest in reliably reproducing  
31 builds over a long time span.

32 Apertis addresses that need by providing stable release channels as a platform for  
33 products with a clear trade-off between leading-edge functionality and stability.  
34 Apertis encourages products to track these channels closely to deploy updates  
35 on a regular basis to ensure important fixes reach devices in a timely manner.

36 Stable release channels are supported for at least two years, and product teams

37 have three quarters of overlap to rebase to the next release before the old one  
38 reaches end of life. Depending on the demand, Apertis may extend the support  
39 period for specific release channels.

40 However, for debugging purposes it is useful to be able to reproduce old builds  
41 as closely as possible. This document describes the approach chosen by Apertis  
42 to address this use case.

43 For our purposes bit-by-bit reproducibility is not a goal, but the aim is to be  
44 able to reproduce builds closely enough that one can reasonably expect that no  
45 regressions are introduced. For instance some non essential variations involve  
46 things like timestamps or items being listed differently in places where order  
47 is not significant, cause builds to not be bit-by-bit identical while the runtime  
48 behavior is not affected.

## 49 Apertis artifacts and release channels

50 As described in the [release flow](#)<sup>1</sup> document, at any given time Apertis has mul-  
51 tiple active release channels to both provide a stable foundation for product  
52 teams and also give them full visibility on the latest developments.

53 Each release channel has its own artifacts, the main one being the [deployable](#)  
54 [images](#)<sup>2</sup> targeting the [reference hardware platforms](#)<sup>3</sup>, which get built by mixing:

- 55 • reproducible build environments
- 56 • build recipes
- 57 • packages
- 58 • external artifacts

59 These inputs are also artifacts themselves in moderately complex ways:

- 60 • build environments are built by mixing dedicated recipes and packages
- 61 • packages are themselves built using dedicated reproducible build environ-  
62 ments

63 However, the core principle for maintaining multiple concurrent release channels  
64 is that each channel should have its own set of inputs, so that changes in a  
65 channel do not impact other channels.

66 Even within channels sometimes it is desirable to reproduce a past build as  
67 closely as possible, for instance to deliver a hotfix to an existing product while  
68 minimizing the chance of introducing regressions due to unrelated changes. The  
69 Apertis goal of reliable, reproducible builds does not only help developers in  
70 their day-to-day activities, but also gives them the tools to address this specific  
71 use-case.

---

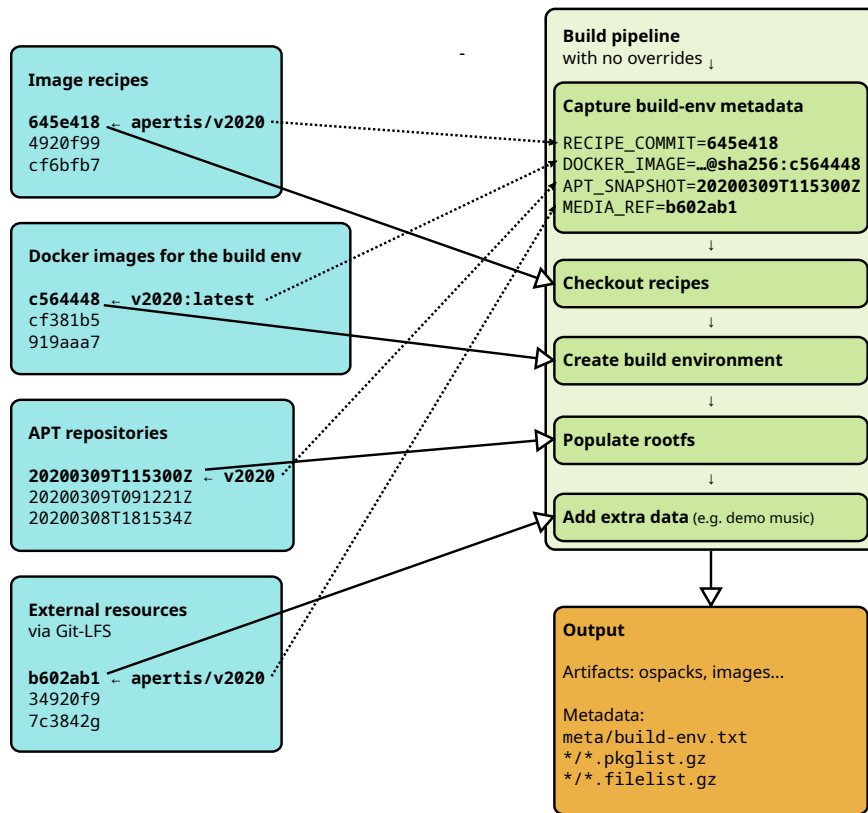
<sup>1</sup><https://www.apertis.org/policies/release-flow/>

<sup>2</sup><https://www.apertis.org/policies/images/>

<sup>3</sup>[https://www.apertis.org/reference\\_hardware/](https://www.apertis.org/reference_hardware/)

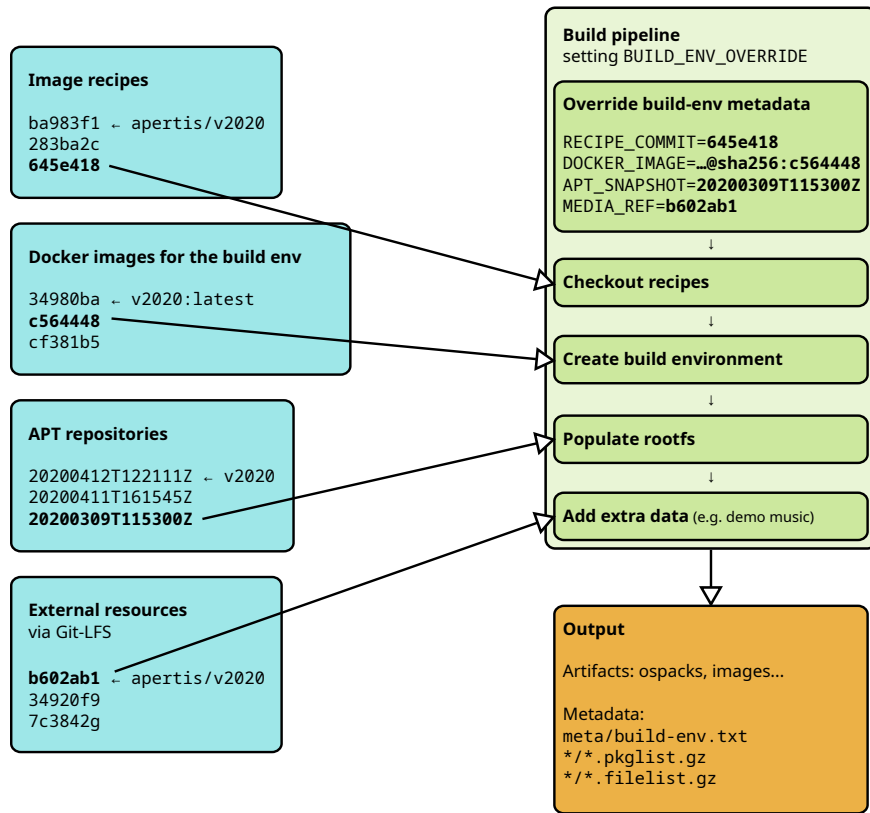
72 The first step is to ensure that all the inputs to the build pipeline are version-  
73 controlled, from the pipeline definition itself to the package repositories and to  
74 any external data.

75 To track which input got used during the build process the pipeline stores an  
76 identifier for each of them to uniquely identify them. For instance, the pipeline  
77 saves all the Git commit hashes, Docker image hashes, and package versions in  
78 the output metadata.



79

80 While the pipeline defaults to using the latest version available in a specific  
81 channel for each input, it is possible to pin specific version to closely reproduce  
82 a past build using the identifiers saved in its metadata.



83

## 84 Reproducible build environments

85 A key challenge in the long term maintenance of a complex project is the ability  
86 to reproduce its build environment in a consistent way. Failing to do so means  
87 that undetected differences across build environments may introduce hard to  
88 debug issues or that builds may fail entirely depending on where/when they get  
89 triggered.

90 In some cases, losing access to the build environment effectively means that a  
91 project can't be maintained anymore, as no new build can be made.

92 To be able to avoid these issues as much as possible, Apertis makes heavy use  
93 of isolated containers based on Docker images<sup>4</sup>

94 All the Apertis build pipelines run in containers with minimal access to external  
95 resources to keep the impact of the environment as low as possible.

96 For the most critical components, even the container images themselves are

<sup>4</sup><https://gitlab.apertis.org/infrastructure/apertis-image-recipes/#building-in-docker>

97 created using Apertis resources, minimizing the reliance on any external service  
98 and artifacts.

99 For instance, the `apertis-v2020-image-builder` container image provides the re-  
100 producible environment to run the pipelines building the reference image arti-  
101 facts for the v2020 release, and the `apertis-v2020-package-source-builder` con-  
102 tainer image is used to convert the source code stored in GitLab in a format  
103 suitable for building on OBS.

104 Each version of each image is identified by a hash, and possibly by some tags.  
105 As an example the `:latest` tag points to the image which gets used by default for  
106 new builds. However, it is possible to retrieve arbitrary old images by specifying  
107 the actual image hash, providing the ability to reliably reproduce arbitrarily old  
108 build environments.

109 To prevent space consumption to grow unboundedly, images that are not pointed  
110 by any tag are periodically garbage-collected and removed. To ensure that the  
111 needed images are preserved, product teams must ensure that there's **at least**  
112 **one tag** pointing to them.

113 Each container image build should be tagged with its build id, for instance  
114 `:build-20200103.0112` at build time; at release time, the container image used to  
115 build the artifacts should be additionally tagged with a release tag, for instance  
116 `:v2020.3` for the v2020.3 release.

117 [Cleanup policies](#)<sup>5</sup> must be set up to make the build tags expire after some time,  
118 to ensure that the unused container images can be reclaimed during garbage-  
119 collection.

120 To further make build environments more reproducible, care can be taken to  
121 make their own build process as reproducible as possible. The same concerns  
122 affecting the main build recipes affect the recipes for the Docker images, from  
123 storing pipelines in Git, to relying only on snapshotted package archives, to  
124 taking extra care on third-party downloads, and the following sections address  
125 those concerns for both the build environments and the main build process.

## 126 Build recipes

127 The process to the reference images is described by textual, YAML-based [Debos](#)  
128 [recipes](#)<sup>6</sup> Git repository, with a different branch for each release channel.

129 The textual, YAML-based GitLab-CI pipeline definitions then control how the  
130 recipes are invoked and combined.

131 Relying on Git for the definition of the build pipelines make preserving old  
132 versions and tracking changes over time trivial.

---

<sup>5</sup>[https://docs.gitlab.com/ee/user/packages/container\\_registry/reduce\\_container\\_registr  
y\\_storage.html#cleanup-policy](https://docs.gitlab.com/ee/user/packages/container_registry/reduce_container_registry_storage.html#cleanup-policy)

<sup>6</sup><https://gitlab.apertis.org/infrastructure/apertis-image-recipes/>

133 Rebuilding the `v2020` artifacts locally is then a matter of checking out the recipes  
134 in the `apertis/v2020` branch and launching `debos` from a container based on the  
135 `apertis-v2020-image-builder` container image.

136 By forking the repository on GitLab the whole build pipeline can be reproduced  
137 easily with any desired customization under the control of the developer.

## 138 Packages and repositories

139 The large majority of the software components shipped in Apertis are packaged  
140 using the Debian packaging format, with the source code stored in GitLab that  
141 OBS uses to generate prebuilt binaries to be published in a APT-compatible  
142 repository.

143 Separate Git branches and OBS projects are used to track packages and versions  
144 across different parallel releases, see [therelease flow](#)<sup>7</sup> document for more details.

145 For instance, for the `v2020` stable release:

- 146 • the `apertis/v2020` Git branch tracks the source revisions to be landed in  
147 the main OBS project
- 148 • the `apertis:v2020:{target,development,sdk}` projects build the stable pack-  
149 ages
- 150 • the `deb https://repositories.apertis.org/apertis/ v2020 target develop-`  
151 `ment sdk entry points apt` to the published packages

152 For most of the time the stable channel is frozen and updates are exclusively  
153 delivered through the dedicated channels described below.

154 Updates are split between small security fixes with low chance of regressions  
155 and updates that also address important but non security-related issues which  
156 usually benefit from more testing.

157 For security updates:

- 158 • the Git branch is `apertis/v2020-security`
- 159 • the OBS projects are `apertis:v2020:security:{target,development,sdk}`
- 160 • `deb https://repositories.apertis.org/apertis/ v2020-security target de-`  
161 `velopment sdk` is the APT repository

162 Similarly, for the general updates:

- 163 • the Git branch is `apertis/v2020-updates`
- 164 • the OBS projects are `apertis:v2020:updates:{target,development,sdk}`
- 165 • `deb https://repositories.apertis.org/apertis/ v2020-updates target de-`  
166 `velopment sdk` is the APT repository

167 On a quarterly basis the stable channel get unfrozen and all the updates get  
168 rolled in it, while the `security` and `updates` channel get emptied.

---

<sup>7</sup><https://www.apertis.org/policies/release-flow/>

169 This approach provides to downstreams and product teams a stable basis to  
170 build their product without hard to control changes. Products are recommended  
171 to also track the security channel for timely fixes, enabling product teams to  
172 easily identify and review the changes shipped through it.

173 The updates channel is not directly meant for production, but it offers to product  
174 teams a preview of the pending changes to let them proactively detect issues  
175 before they reach the stable channel and thus their products.

176 While the stability of the release channels is suitable for most use-cases, some-  
177 times it is desirable to reproduce an old build as close to the original as possible,  
178 ignoring any update regardless of their importance.

179 To accomplish that goal the package archives are snapshotted regularly, storing  
180 their full history. The image build pipeline accepts an optional parameter to use  
181 a specific snapshot rather than the latest contents. This results in the execution  
182 installing exactly the same packages and versions as the original run, regardless  
183 of any changes that landed in the archive in the meantime.

184 To use a snapshot it is sufficient to change the APT mirror address,  
185 for instance going from `https://repositories.apertis.org/apertis/` to  
186 `https://repositories.apertis.org/apertis/20200305T132100Z` and similarly  
187 for product-specific repositories.

188 Every time an update is published from OBS a snapshot is created, tracking the  
189 full history of each archive. More advanced use-cases can be addressed using  
190 the optional [Aptly HTTP API](https://www.apertis.org/docs/apertis/20200305T132100Z)<sup>8</sup>.

## 191 External artifacts

192 While the packaging pipeline effectively forbids any reliance on external artifacts,  
193 the other pipelines in some case include components not under the previously  
194 mentioned systems to track per-release resources.

195 For instance, the recipes for the HMI-enabled images include a set of example  
196 media files retrieved from a `multimedia-demo.tar.gz` file hosted on an Apertis  
197 web server.

198 Another example is given by the `apertis-image-builder` recipe checking out De-  
199 bos directly from the master branch on GitHub.

200 In both cases, any change on the external resources impacts directly all the  
201 release channels when building the affected artifacts.

202 A minimal solution for `multimedia-demo.tar.gz` would be to put a version in its  
203 URL, so that recipes can be updated to download new versions without affecting  
204 older recipes. Even better, its contents could be put in a version tracking tool,  
205 for instance using the Git LFS support available on GitLab.

---

<sup>8</sup><https://www.apertis.org/docs/apertis/>



206 In the Debos case it would be sufficient to encode in the recipe a specific revision  
207 to be checked out. A more robust solution would be to use the packaged version  
208 shipped in the Apertis repositories.

## 209 Main artifacts and metadata

210 The purpose of the previously described software items is to generate a set  
211 of artifacts, such as those described on the [images](#)<sup>9</sup> page. With the artifacts  
212 themselves a few metadata entries are generated to help tracking what has been  
213 used during the build.

214 In particular, the `pkglist` files capture the full list of packages installed on each  
215 artifacts along their version. The `filelist` files instead provide basic information  
216 about the actual files in each artifacts.

217 With the information contained in the `pkglist` files it is possible to find the exact  
218 binary package version installed and from there find the corresponding commit  
219 for the sources stored in GitLab by looking at the matching Git tag.

220 The `build-env.txt` file instead captures metadata about the build environment.

221 For instance, here's a sample from the pipeline that [built the v2021dev3.0 re-](#)  
222 [lease](#)<sup>10</sup>:

```
223 PIPELINE_VERSION=20200921.1223
224 DOCKER_IMAGE=registry.gitlab.apertis.org/infrastructure/apertis-docker-
225 images/v2021dev3-image-builder@sha256:50724ec3105f9ea840fa70b536768148722ae59e09b7861a9051ad1397b57f64
226 RECIPES_COMMIT=b4f1c5c85bd4603f2d9158f513c142a77a3c65c3
227 RECIPES_URL=https://gitlab.apertis.org/infrastructure/apertis-image-recipes/
228 PIPELINE_URL=https://gitlab.apertis.org/infrastructure/apertis-image-
229 recipes/-/pipelines/157555
230 UPLOAD_ROOT=/srv/images/public
231 IMAGE_URL_PREFIX=https://images.apertis.org
```

232 With the `RECIPES_URL` and `RECIPES_COMMIT` variables it is possible to find the exact  
233 revision of the recipes [in the apertis-image-recipes project](#)<sup>11</sup>

234 The `DOCKER_IMAGE` variable captures the exact revision of the Docker image by  
235 explicitly using the digest syntax, to ensure the build environment can be re-  
236 produced perfectly. Care must be taken to ensure the retention policy of the  
237 container registry preserves the used image for long enough. For the Apertis  
238 reference image recipes we currently use a rather aggressive cleanup policy, only  
239 preserving images built during the past week but this can be [easily customized](#)  
240 [from the GitLab UI](#)<sup>12</sup>. Improving the preservation of the images used for each  
241 release is under discussion.

---

<sup>9</sup><https://www.apertis.org/policies/images/>

<sup>10</sup><https://images.apertis.org/release/v2021dev3/v2021dev3.0/meta/build-env.txt>

<sup>11</sup><https://gitlab.apertis.org/infrastructure/apertis-image-recipes/commit/b4f1c5c85bd4603f2d9158f513c142a77a3c65c3>

<sup>12</sup>[https://docs.gitlab.com/ce/user/packages/container\\_registry/#cleanup-policy](https://docs.gitlab.com/ce/user/packages/container_registry/#cleanup-policy)

242 The metadata above can then be used to **reproduce the build**.  
243 The **implementation plan** section defines the remaining planned improvements.

## 244 **Package builds**

245 Package builds happen on OBS which does not have snapshotting capabilities  
246 and always builds every package on a clean, isolated environment built using  
247 the latest package versions for each channel.

248 Since the purposes taken in account in this document do not involve large scale  
249 package rebuilds, it is recommended to use the SDK images and the deviants  
250 in combination with the snapshotted APT archives to rebuild packages in an  
251 environment closely matching a past build.

## 252 **Recommendations for product teams**

253 Builds for production should:

- 254 1. pick a specific stable channel (for instance, `v2020`)
- 255 2. version control the build pipelines using branches specific to a stable chan-  
256 nel
- 257 3. in the build pipeline, use the latest Docker image for that specific channel,  
258 for instance `v2020-image-builder` or a product-specific downstream image  
259 based on that
- 260 4. use the main OBS projects for the release channel, for instance `aper-`  
261 `tis:v2020:target`, with the security fixes from `apertis:v2020:security:target`  
262 layered on top
- 263 5. store the product-specific packages in OBS projects targeting a specific  
264 release channel, layered on top of the projects mentioned in the previous  
265 point
- 266 6. use the matching APT archives during the image build process
- 267 7. deploy fixes from the stable channels as often as possible

268 Development builds are encouraged to also use the contents from the non-  
269 security updates (for instance, `apertis:v2020:updates:target`) to get a preview  
270 of non time-critical updates that will folded in the main archive on a quarterly  
271 basis.

272 The assumption is that products will use custom build pipelines tailored to the  
273 specific hardware and software needs of the product. However, product teams  
274 are strongly encouraged to reuse as much as possible from the reference Apertis  
275 build pipelines using the GitLab CI and Debos include mechanisms, and to fol-  
276 low the same best-practices about metadata tracking and build reproducibility  
277 described in this document.

## 278 **Implementation plan**

### 279 **Snapshot the package archive**

280 To ensure that build can be reproduced, it is fundamental to make the same  
281 contents available from the package archive.

282 The most common approach, also employed in Debian upstream, is to take  
283 snapshots of the archive contents so that subsequent builds can point to the  
284 snapshotted version and retrieve the exact package versions originally used.

285 To provide the needed server-side support, the archive manager need to be  
286 switched to the `aptly` archive manager as it provides explicit support for snap-  
287 shots. The build recipes then need to be updated to capture the current snapshot  
288 version and to be able to optionally specify one when initiating the build.

289 Due to the way APT works, the increase in storage costs for the snapshot is  
290 small, as the duplication is limited to the index files, while the package contents  
291 are deduplicated.

### 292 **Version control external artifacts**

293 External artifacts like the sample multimedia files need to be versioned just like  
294 all the other components. Using Git LFS and Git tags would give fine control  
295 to the build recipe over what gets downloaded.

### 296 **Link to the tagged sources**

297 The package name and package version as captured in the `pkglist` files are  
298 sufficient to identify the exact sources used to generate the packages installed  
299 on each artifacts, as they can be used to identify an exact commit.

300 However, the process can be further automated by providing explicit hyperlinks  
301 to the tagged revision on GitLab.

## 302 **How to reproduce a release build and customize** 303 **a package**

### 304 **Reproduce the build**

- 305 1. Open the folder containing the build artifacts, for instance `v2021dev3.0/`<sup>13</sup>
- 306 2. Find the `build-env.txt` metadata, for instance `meta/build-env.txt`<sup>14</sup>
- 307 3. Find the project hosting the recipes with the `RECIPES_URL` variable in `build-`  
308 `env.txt`

---

<sup>13</sup><https://images.apertis.org/release/v2021dev3/v2021dev3.0/>

<sup>14</sup><https://images.apertis.org/release/v2021dev3/v2021dev3.0/meta/build-env.txt>

- 309 4. On GitLab, [fork](#)<sup>15</sup> the recipes project  
310 5. Create a [new branch](#)<sup>16</sup> in the recipes repository pointing to the commit  
311 saved in the `RECIPES_COMMIT` field of `build-env.txt`, for instance commit  
312 `b4f1c5c85bd4603f2d9158f513c142a77a3c65c3`<sup>17</sup>  
313 6. Go to Pipelines → Run Pipeline page on GitLab to [execute a CI pipeline](#)<sup>18</sup>  
314 7. [Configure a variable](#)<sup>19</sup> of type `File` named `BUILD_ENV_OVERRIDE`  
315 8. Paste the contents of `build-env.txt` there  
316 9. Be careful with `PIPELINE_VERSION`: to avoid overwriting an existing build it  
317 is recommended to set a custom one  
318 10. Run the pipeline

319 When the pipeline completes, the produced artifacts should closely match the  
320 original ones, albeit not being bit-by-bit identical.

## 321 Customizing the build

322 On the newly created branch in the forked recipe repository, changes can be  
323 committed just like on the main repository.

324 For instance, to install a custom package:

- 325 1. Check out the forked repository  
326 2. Edit the relevant `ospack` recipe to install the custom package, either by  
327 adding a custom APT archive in the `/etc/apt/sources.list.d` folder if avail-  
328 able, or retrieving and installing it with `wget` and `dpkg` (small packages can  
329 even be committed as part of the repository to run quick experiments  
330 during development)  
331 3. Commit the results and push the branch  
332 4. Execute the pipeline as described in the previous section

## 333 Example 1: OpenSSL security fix 2 years after 334 release v1.0.0

335 Today a product team makes the official release of version 1.0.0 of their software  
336 that is based on Apertis. Two years from now a critical security vulnerability  
337 will be found and fixed in OpenSSL. How can the product team issue a new  
338 release two years from now with the only change being the fix to OpenSSL?

339 It is important for product teams to consider their future requirements at the  
340 point they make a release. To ensure bug and security fixes can be deployed

<sup>15</sup>[https://docs.gitlab.com/ee/user/project/repository/forking\\_workflow.html#creating-a-fork](https://docs.gitlab.com/ee/user/project/repository/forking_workflow.html#creating-a-fork)

<sup>16</sup><https://docs.gitlab.com/ee/gitlab-basics/create-branch.html>

<sup>17</sup><https://gitlab.apertis.org/infrastructure/apertis-image-recipes/commit/b4f1c5c85bd4603f2d9158f513c142a77a3c65c3>

<sup>18</sup><https://docs.gitlab.com/ee/ci/pipelines.html#manually-executing-pipelines>

<sup>19</sup><https://docs.gitlab.com/ee/ci/variables/README.html#create-a-custom-variable-in-the-ui>

341 with minimal impact on users a number of artifacts need to be preserved from  
342 the initial release:

- 343 1. The image recipes
- 344 2. The Docker images used as build environment
- 345 3. The APT repositories
- 346 4. External artifacts

## 347 **Getting started with Apertis: one year before release 1.0.0**

348 Good news! A product team has decided to use Apertis as platform for their  
349 product. At this stage there are a few recommendations on how to get started  
350 that will make it easier to use Apertis long term reproducibility features.

351 The product team needs control over their software releases, and is important  
352 to decouple their releases from Apertis. One important objective is to give the  
353 product team control over importing changes from Apertis, such as package  
354 updates. We recommend using release channels for that.

355 A product team can have multiple release channels, each reflecting what is  
356 deployed for a specific product. And because release channels are independent  
357 and parallel deliveries, a single product may even have multiple release channels,  
358 for instance a stable channel and a development one.

359 In turn each product release channel is based on an Apertis release chan-  
360 nel. As an hypothetical example the `automotive` product team may have an  
361 `automotive/cluster-v1` release channel for delivering stable updates to their  
362 `cluster` product, and an `automotive/cluster-v2` release channel for development  
363 purposes, both based on the same `apertis/v2020` release channel.

364 Git repositories need to use a different branch for each release channel, and each  
365 release channel has its own set of projects on OBS. However only the components  
366 that the product team need to customize have to be branched or forked. To  
367 maximize reuse, it is expected that the bulk of packages used by every product  
368 team will come directly from the main Apertis release channels.

- 369 1. **What:** Create a dedicated release channel
- 370 2. **Where:** GitLab and OBS
- 371 3. **How:** Create release channel branches in each Git repository that diverges  
372 from the ones provided by Apertis; set up OBS projects matching those  
373 release channels to build the packages

374 In this way the product team has complete control on the components used to  
375 build their products:

- 376 • Source code for all packages is stored on GitLab with full development  
377 history
- 378 • Compiled binary packages are tracked by the APT archive snapshotting  
379 system for both the product-specific packages and the packages in the  
380 main Apertis archive.

381 The previous step took care of the Apertis layer of the software stack, but there  
382 is one important set of components missing: the product team software. We  
383 suggest that product teams use one of Apertis recommended ways for shipping  
384 software which consists of using .deb packages or Flatpaks. For this example  
385 we are going to use .deb packages.

386 While there are multiple ways of handling product team specific software, for  
387 this example we are going to recommend the product team to create a new APT  
388 suite and a few APT components, and host them on the Apertis infrastructure.  
389 We will call the new suite cluster-v1. The list of APT repositories will then be:

```
390 deb https://repositories.apertis.org/apertis/ v2020 target development sdk  
391 deb https://repositories.apertis.org/automotive/ cluster-v1 target
```

392 For reference, in [APT terminology](#)<sup>20</sup> both `v2020` and `cluster-v1` are suites or  
393 distributions, and `target`, `development`, and `sdk` are components.

394 The steps are:

- 395 1. **What:** Create new APT suite and APT components for the product team
- 396 2. **Where to host:** Apertis infrastructure

## 397 **Creating the list of golden components: the day of the** 398 **release 1.0.0**

399 As we mentioned earlier each component is identified by a hash, and it is also  
400 possible to create tags. We recommend using hashes for identification of specific  
401 revisions because hashes are immutable. Tags can also be used, but we recom-  
402 mend careful evaluation as most tools allow tags to be modified after creation.  
403 Modifying tags can lead to problems that are difficult to debug.

404 The image recipe is usually a small set of files that are stored in a single Git  
405 repository. Collect the hash of the latest commit of the recipe repository.

- 406 1. **What:** Image recipe
- 407 2. **Where:** Apertis GitLab
- 408 3. **How:** Collect the Git hash of the latest commit of the recipe files

409 The Docker containers used for building are stored in GitLab Container Registry.  
410 The Registry also allow to identify containers by hashes.

411 There are expiration policies and clean-up tools for deleting old versions of  
412 containers. Make sure the golden containers are protected against clean-up and  
413 expiration.

- 414 1. **What:** Docker containers used for building: `apertis-v2020-image-builder`  
415 and `apertis-v2020-package-source-builder`
- 416 2. **Where:** GitLab Container Registry

---

<sup>20</sup><https://manpages.debian.org/testing/apt/sources.list.5.en.html>

- 417 3. **How:** On the GitLab Container Registry collect the hash for each con-  
418 tainer used for building
- 419 4. **Do not forget:** Make sure the expiration policy and clean-up routines  
420 will not delete the golden containers

421 From the perspective of APT clients, such as the tools used to create Apertis  
422 images, APT repositories are simply a collection of static files served through the  
423 web. The recommended method for creating the golden set of APT repositories  
424 is to create snapshots using `aptly`. `aptly` is used by Debian upstream and is  
425 capable of making efficient use of disk space for snapshots. `aptly` snapshots are  
426 identified by tags. Something along the lines of:

```
427 aptly snapshot create v1.0.0 from mirror target
```

428 Repeat the command for `target`, `development`, `sdk`, and `cluster-v1`.

429 It is important to mention that the product team needs to create a snapshot  
430 **every time a package is updated**. This is the only way to keep track the  
431 full history of the APT archive.

- 432 1. **What:** APT repositories:

```
433 deb https://repositories.apertis.org/apertis/ v2020 target development sdk  
434 deb https://repositories.apertis.org/automotive/ cluster-v1 target
```

- 435 2. **Where:** `aptly`

- 436 3. **How:** create a snapshot for each repository using `aptly`

- 437 4. **Do not forget:** create a snapshot for every package update

438 External artifacts should be avoided, but some times they are required. An  
439 example of external artifacts are the multimedia files Apertis uses for testing.  
440 Those files are currently simply hosted on a web server which creates two prob-  
441 lems: no versioning information, and no long term guarantee of availability.

442 To address this issue we recommend creating a repository on GitLab, and copy  
443 all external artifacts to it. This gives the benefit of using the well defined  
444 processes around versioning and tracking that are already used by the other  
445 components. For large files we recommend using Git LFS.

- 446 1. **What:** External artifacts: files that are needed during the build but that  
447 are not in Git repositories
- 448 2. **Where:** A new repository in GitLab
- 449 3. **How:** Create a GitLab repository for external artifacts, add files, use Git  
450 LFS for large files, and collect the hash pointing to the correct version of  
451 files

452 Notice that the main idea is to collect hashes for the various resources used for  
453 building. The partial exception are external resources, but our suggestion is to  
454 also create a Git repository for hosting the external artifacts and then collect  
455 and use the Git hash as a pointer to the correct version of the content.

456 At the time of writing there is work planned to automate the collection of  
457 relevant hashes that were used to create an image. The outcome of the planned  
458 work will be the publication of text files containing all relevant hashes for future  
459 use.

## 460 **Using the golden components two years after release 1.0.0:** 461 **Creating the new release**

462 We recommend product teams to make constant releases, for example in a quar-  
463 terly basis, to cover security updates and to minimize the technical debt to  
464 Apertis upstream. However in some cases a product team may decide to have  
465 a much longer release cycle, and for our example, the product team decided to  
466 make the second release two years after the first one.

467 For our example the product team wants the second release to include a fix for  
468 OpenSSL that corrects a security vulnerability, but be as identical as possible  
469 otherwise. A note of caution here is that deterministic builds, or the ability to  
470 build packages that are byte-by-byte identical in different builds, is not expected  
471 to happen naturally and is outside the scope of this guide. A good source of  
472 information about this topic is the [Debian Reproducible Builds<sup>21</sup>](#) page.

473 Our aim is to be able to reproduce builds closely enough so that one can reason-  
474 ably expect that no regressions are introduced. For instance some non essential  
475 variations could be caused by different time stamps or different paths for files.  
476 These variations cause builds to not be byte-by-byte identical while the runtime  
477 behavior is not affected.

478 For our example the product team will import the updated OpenSSL package  
479 from Apertis, build the OpenSSL package, and build images for the new v1.0.1  
480 release.

481 The first step is to rescue all the hashes that were collected on the day of the  
482 build.

## 483 **Reproduce the build**

484 The `build-env.txt` produced by the build pipeline should capture all the infor-  
485 mation needed to reproduce it as closely as possible:

- 486 1. Retrieve the `build-env.txt` from the golden build
- 487 2. On GitLab create a new branch<sup>22</sup> on the previously identified recipe repos-  
488 itory. The branch should point to the golden commit which should be  
489 captured in the `RECIPES_COMMIT` field.

---

<sup>21</sup><https://wiki.debian.org/ReproducibleBuilds>

<sup>22</sup>[https://docs.gitlab.com/ee/user/project/repository/web\\_editor.html#create-a-new-branch-from-a-projects-dashboard](https://docs.gitlab.com/ee/user/project/repository/web_editor.html#create-a-new-branch-from-a-projects-dashboard)



490 3. [Execute a CI pipeline](#)<sup>23</sup> on the newly created branch, reproducing or  
491 customizing the original build environment by creating a variable called  
492 `BUILD_ENV_OVERRIDE` into which the contents from `build-env.txt` should be  
493 pasted, modifying it as desired.

494 When the pipeline completes, the produced artifacts should closely match the  
495 original ones, albeit not being bit-by-bit identical.

## 496 Customizing the build

497 On the newly created branch in the forked recipe repository, changes can be  
498 committed just like on the main repository.

499 For instance, to install a custom package:

- 500 1. Check out the newly-created branch
- 501 2. Edit the relevant ospark recipe to install the custom package, either by  
502 adding a custom APT archive in the `/etc/apt/sources.list.d` folder if avail-  
503 able, or retrieving and installing it with `wget` and `dpkg` (small packages can  
504 even be committed as part of the repository to run quick experiments  
505 during development)
- 506 3. Commit the results and push the branch
- 507 4. Execute the pipeline as described in the previous section

---

<sup>23</sup><https://docs.gitlab.com/ee/ci/pipelines.html#manually-executing-pipelines>