



Long term reproducibility

1	Contents	
2	Background	2
3	Apertis artifacts and release channels	2
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	Recommendations for product teams	10
11	Implementation plan	11
12	Snapshot the package archive	11
13	Version control external artifacts	11
14	Link to the tagged sources	11
15	How to reproduce a release build and customize a package	11
16	Reproduce the build	11
17	Customizing the build	12
18	Example 1: OpenSSL security fix 2 years after release v1.0.0	12
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](#)¹ 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](#)² targeting the [reference hardware platforms](#)³, 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.

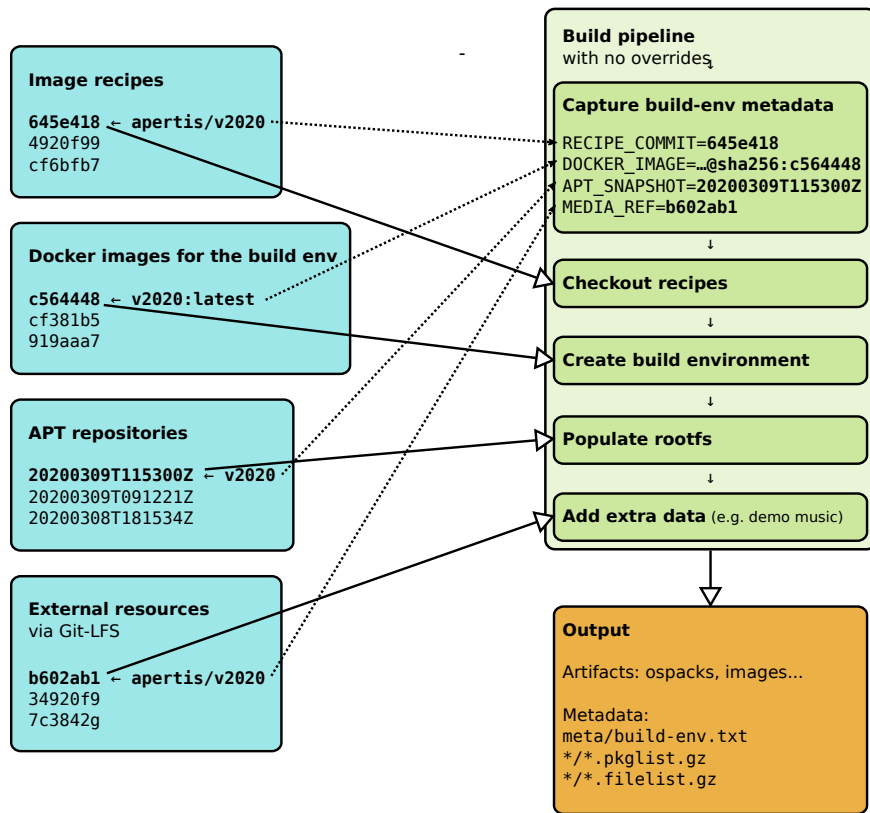
¹<https://jwd.pages.apertis.org/apertis-website/policies/release-flow/>

²<https://jwd.pages.apertis.org/apertis-website/policies/images/>

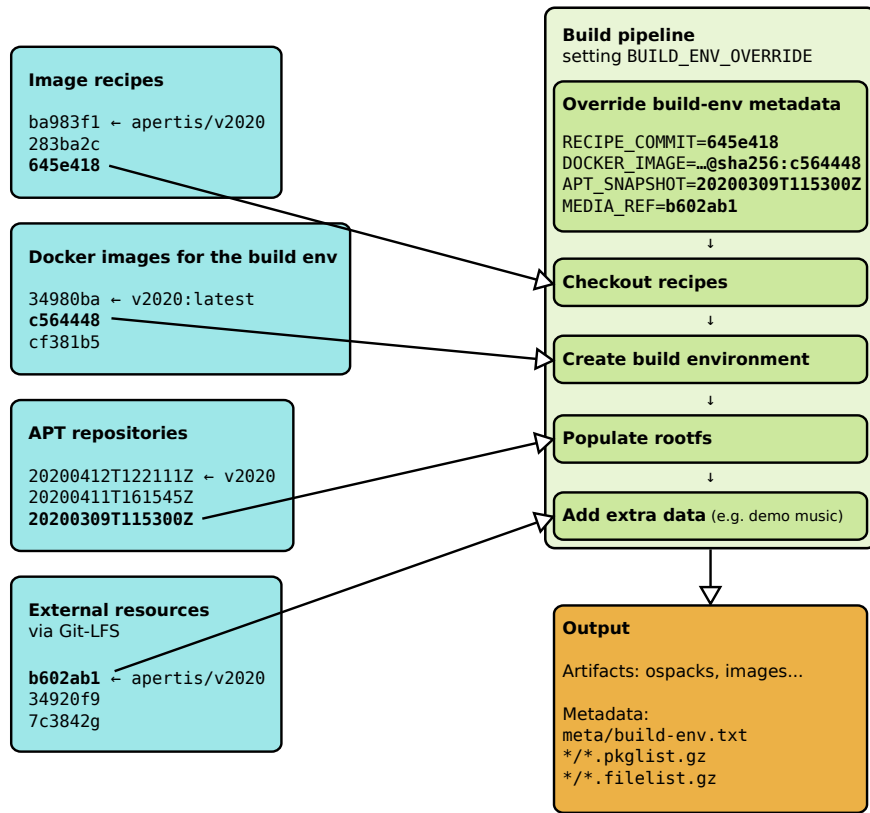
³https://www.apertis.org/reference_hardware/

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](#)⁴

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

⁴<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 By default the Docker registry where image are published keeps all the past
110 versions, so every build environment can be reproduced exactly.

111 Unfortunately this comes with a significant cost from a storage point of view,
112 so each team needs to evaluate the trade-off that better fits their goals in the
113 spectrum that goes from keeping all Docker images around for the whole lifespan
114 of the product to more aggressive pruning policies involving the deletion of old
115 images on the assumption that changes in the build environment have a limited
116 effect on the build and using an image version which is close to but not exactly
117 the original one gives acceptable results.

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

124 **Build recipes**

125 The process to the reference images is described by textual, YAML-based [Debos](#)
126 [recipes](#)⁵ Git repository, with a different branch for each release channel.

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

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

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

⁵<https://gitlab.apertis.org/infrastructure/apertis-image-recipes/>

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

136 Packages and repositories

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

141 Separate Git branches and OBS projects are used to track packages and versions
142 across different parallel releases, see the [release flow](#)⁶ document for more details.

143 For instance, for the v2020 stable release:

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

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

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

155 For security updates:

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

160 Similarly, for the general updates:

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

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

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

⁶<https://jwd.pages.apertis.org/apertis-website/policies/release-flow/>

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

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

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

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

186 Every time an update is published from OBS a snapshot is created, tracking the
187 full history of each archive. More advanced use-cases can be addressed using
188 the optional [Aptly HTTP API](https://www.apertis.org/docs/apertis/20200305T132100Z)⁷.

189 External artifacts

190 While the packaging pipeline effectively forbids any reliance on external arti-
191 facts, the other pipelines in some case include components not under the previ-
192 ously mentioned systems to track per-release resources.

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

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

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

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

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

⁷<https://www.apertis.org/docs/apertis/>

207 Main artifacts and metadata

208 The purpose of the previously described software items is to generate a set
209 of artifacts, such as those described on the [images](#)⁸ page. With the artifacts
210 themselves a few metadata entries are generated to help tracking what has been
211 used during the build.

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

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

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

219 For instance, here's a sample from the pipeline that [built the v2021dev3.0 re-](#)
220 [lease](#)⁹:

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

230 With the `RECIPES_URL` and `RECIPES_COMMIT` variables it is possible to find the exact
231 revision of the recipes [in the apertis-image-recipes project](#)¹⁰

232 The `DOCKER_IMAGE` variable captures the exact revision of the Docker image by
233 explicitly using the digest syntax, to ensure the build environment can be re-
234 produced perfectly. Care must be taken to ensure the retention policy of the
235 container registry preserves the used image for long enough. For the Apertis
236 reference image recipes we currently use a rather aggressive cleanup policy, only
237 preserving images built during the past week but this can be [easily customized](#)
238 [from the GitLab UI](#)¹¹. Improving the preservation of the images used for each
239 release is under discussion.

240 The metadata above can then be used to [reproduce the build](#).

241 The [implementation plan](#) section defines the remaining planned improvements.

⁸<https://jwd.pages.apertis.org/apertis-website/policies/images/>

⁹<https://images.apertis.org/release/v2021dev3/v2021dev3.0/meta/build-env.txt>

¹⁰<https://gitlab.apertis.org/infrastructure/apertis-image-recipes/commit/b4f1c5c85bd4603f2d9158f513c142a77a3c65c3>

¹¹https://docs.gitlab.com/ce/user/packages/container_registry/#cleanup-policy

242 **Package builds**

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

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

250 **Recommendations for product teams**

251 Builds for production should:

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

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

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

276 **Implementation plan**

277 **Snapshot the package archive**

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

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

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

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

290 **Version control external artifacts**

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

294 **Link to the tagged sources**

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

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

300 **How to reproduce a release build and customize** 301 **a package**

302 **Reproduce the build**

- 303 1. Open the folder containing the build artifacts, for instance `v2021dev3.0/`¹²
- 304 2. Find the `build-env.txt` metadata, for instance `meta/build-env.txt`¹³
- 305 3. Find the project hosting the recipes with the `RECIPES_URL` variable in `build-`
306 `env.txt`

¹²<https://images.apertis.org/release/v2021dev3/v2021dev3.0/>

¹³<https://images.apertis.org/release/v2021dev3/v2021dev3.0/meta/build-env.txt>

- 307 4. On GitLab, [fork](#)¹⁴ the recipes project
- 308 5. Create a [new branch](#)¹⁵ in the recipes repository pointing to the commit
- 309 saved in the `RECIPES_COMMIT` field of `build-env.txt`, for instance commit
- 310 `b4f1c5c85bd4603f2d9158f513c142a77a3c65c3`¹⁶
- 311 6. Go to Pipelines → Run Pipeline page on GitLab to [execute a CI pipeline](#)¹⁷
- 312 7. [Configure a variable](#)¹⁸ of type `File` named `BUILD_ENV_OVERRIDE`
- 313 8. Paste the contents of `build-env.txt` there
- 314 9. Be careful with `PIPELINE_VERSION`: to avoid overwriting an existing build it
- 315 is recommended to set a custom one
- 316 10. Run the pipeline

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

319 Customizing the build

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

322 For instance, to install a custom package:

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

331 Example 1: OpenSSL security fix 2 years after 332 release v1.0.0

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

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

¹⁴https://docs.gitlab.com/ee/user/project/repository/forking_workflow.html#creating-a-fork

¹⁵<https://docs.gitlab.com/ee/gitlab-basics/create-branch.html>

¹⁶<https://gitlab.apertis.org/infrastructure/apertis-image-recipes/commit/b4f1c5c85bd4603f2d9158f513c142a77a3c65c3>

¹⁷<https://docs.gitlab.com/ee/ci/pipelines.html#manually-executing-pipelines>

¹⁸<https://docs.gitlab.com/ee/ci/variables/README.html#create-a-custom-variable-in-the-ui>

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

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

345 **Getting started with Apertis: one year before release 1.0.0**

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

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

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

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

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

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

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

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

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

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

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

390 For reference, in [APT terminology](#)¹⁹ both v2020 and cluster-v1 are suites or
391 distributions, and target, development, and sdk are components.

392 The steps are:

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

395 **Creating the list of golden components: the day of the** 396 **release 1.0.0**

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

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

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

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

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

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

¹⁹<https://manpages.debian.org/testing/apt/sources.list.5.en.html>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

458 **Using the golden components two years after release 1.0.0:** 459 **Creating the new release**

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

465 For our example the product team wants the second release to include a fix for
466 OpenSSL that corrects a security vulnerability, but be as identical as possible
467 otherwise. A note of caution here is that deterministic builds, or the ability to
468 build packages that are byte-by-byte identical in different builds, is not expected
469 to happen naturally and is outside the scope of this guide. A good source of
470 information about this topic is the [Debian Reproducible Builds²⁰](#) page.

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

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

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

481 **Reproduce the build**

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

- 484 1. Retrieve the `build-env.txt` from the golden build
- 485 2. On GitLab [create a new branch²¹](#) on the previously identified recipe repos-
486 itory. The branch should point to the golden commit which should be
487 captured in the `RECIPES_COMMIT` field.

²⁰<https://wiki.debian.org/ReproducibleBuilds>

²¹https://docs.gitlab.com/ee/user/project/repository/web_editor.html#create-a-new-branch-from-a-projects-dashboard

488 3. [Execute a CI pipeline](#)²² on the newly created branch, reproducing or
489 customizing the original build environment by creating a variable called
490 `BUILD_ENV_OVERRIDE` into which the contents from `build-env.txt` should be
491 pasted, modifying it as desired.

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

494 Customizing the build

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

497 For instance, to install a custom package:

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

²²<https://docs.gitlab.com/ee/ci/pipelines.html#manually-executing-pipelines>