



Apertis test strategy

1	Contents	
2	Real life challenges in embedded Linux projects	3
3	Development workflow in binary package distribution	4
4	Testing in binary package distribution	5
5	Classifications	5
6	Components	5
7	Component metrics	6
8	Loops and types	7
9	Priorities	9
10	Constraints	9
11	Strategy	10
12	Classify components	10
13	Define tests required for each component	11
14	Current status and gaps	12
15	Considerations for product teams	13
16	Follow up tasks	14
17	Apertis is an Open Source project which consists of multiple parts that are	
18	reflected in the current structure of Apertis Gitlab ¹ :	
19	• Packages as the fundamental building blocks of the images	
20	• Infrastructure to provide the tools and automation to build the images	
21	• Tests which ensure that Apertis provides high quality standards	
22	This structure also shows that tests are one of the pillars of this distribution.	
23	The QA process takes advantage of the tests to confirm that the behavior of each	
24	component is the expected one. During the testing any deviation is reported	
25	for further investigation, as described in the Apertis QA page ² .	
26	For a successful QA process a test strategy should be followed in order to:	
27	• Make sure all the relevant parts are tested, with more focus in critical ones	
28	• Provide reliable reports of the status of the different components	
29	• Provide a reliable base to build additional checks	
30	The goal of this document is to provide a strategy to maximize the profits of	
31	testing by putting the focus in the components with higher impact in case of an	
32	issue.	

¹<https://gitlab.apertis.org>

²<https://www.apertis.org/qa/>

33 Real life challenges in embedded Linux projects

34 Testing is what ties all the pieces together in a project to convert it in a success.
35 Without testing, a project will most probably fail, since the output of one stage
36 won't meet the expectations of a next one. Also, management and risk assess-
37 ment is not possible for projects where a test strategy does not provide certainty.
38 In the end, a product derived from this type of project will be a failure due to
39 different possible reasons:

- 40 • The product might fail to meet the expectations of the consumer
- 41 • The budget associated to the project will be overspent
- 42 • The times constraints associated to the project will not be met

43 This is true in general; however, in embedded Linux projects there are spe-
44 cific challenges to take into account. Traditionally, embedded Linux projects
45 are thought as monolithic software, which basically consists in building full im-
46 ages from several pieces of software with product specific customizations on top.
47 While for small projects this usually includes only small customizations and a
48 custom application on top, usually not requiring any special feature, for more
49 complex projects this approach does not scale well.

50 The reason behind this fact is that on complex projects there are many more
51 variables to be considered:

- 52 • Simple projects consist of standard embedded Linux image and an applica-
53 tion on top of it, while more complex ones usually require customizations
54 of many different pieces of software.
- 55 • Simple projects usually have a local team working on one piece of software,
56 while complex ones tend to have globally distributed teams working in
57 many different pieces of software.
- 58 • Simple projects are usually meant to run on well-known and reliable hard-
59 ware, while complex ones are likely to run in hardware which is also being
60 developed, adding extra uncertainty.
- 61 • Simple projects usually are self contained with little interaction with other
62 systems, on the other hand complex are challenged by interactions with
63 other embedded systems or with external services, such as cloud infras-
64 tructure.

65 It is clear that with so many variables involved, a way to decouple and validate
66 changes through testing is vital for the success of a project. With this in mind,
67 the Apertis test strategy is based on the concept of a binary package distribution,
68 from which Apertis inherits its strengths.

69 Development workflow in binary package distri- 70 bution

71 A test strategy is tied to a development workflow since it should provide cer-
72 tainty to the different stages of development. In this context it is important to
73 highlight the development workflow on Apertis, since it is quite different from
74 other embedded Linux projects.

75 Apertis is a binary package oriented distribution which means that development
76 is based on packages, which are the buildings blocks of images. This approach
77 makes it natural to develop new pieces of software or improving existing ones
78 by changing packages which can be tested isolated from the rest of the system.

79 With a package centric approach each package is self contained, including:

- 80 • Source code
- 81 • Unit tests
- 82 • Documentation
- 83 • Custom patches
- 84 • Rules to build and install
- 85 • Copyright information
- 86 • Custom CI configuration

87 This isolation helps in different ways:

- 88 • Developers can apply changes on a package without being affected by
89 changes in other packages
- 90 • Developers can test their changes locally in an well known environment
91 decoupled from external systems
- 92 • Changes can be tested in CI in a well known environment before they get
93 merged
- 94 • Potential issues are caught earlier during the development process

95 Additionally, the fact that Apertis supports multiple architectures helps both
96 development and testing as changes in a package can be validated in a different
97 environment. Also, taking into account that Apertis SDK is built using the exact
98 same packages, the development and testing is straightforward. Therefore:

- 99 • Developers can build their changes in Apertis SDK
- 100 • Developers can test their changes locally in Apertis SDK by only installing
101 the new version of the package
- 102 • Changes can be tested in CI using a runner with a different architecture
103 before they get merged

104 Finally, integration is easily done by installing a custom set of packages to build
105 the desired image. Since packages are prebuilt, integration is a simple process
106 and packages can be reused to build different types of images, providing higher
107 flexibility as well as faster build times for images.

108 The mentioned characteristics from Apertis overcome the difficulties presented

109 in the traditional monolithic approach of embedded Linux, making it the best
110 option for complex projects.

111 Testing in binary package distribution

112 To take advantage of the benefits of a binary package distribution source pack-
113 ages needs to be designed to be self contained in terms of functionality and
114 testing. This means that a source packages should include not only the func-
115 tionality it is meant to provide, but also a way to validate it. Providing the test
116 functionality could be very challenging in some scenarios, but the benefits of it
117 are worth the price, since it allows scaling in complex projects.

118 The following guidelines allow source packages to provide the test functionality:

- 119 • From source packages several packages can be built, which could poten-
120 tially include:
 - 121 – Binary packages meant to be used in target devices
 - 122 – Alternative binary packages with limited functionality based on ar-
123 chitecture that can be used in development to test basic functionality
 - 124 – Alternative binary packages meant to be used in development which
125 provide functionality to emulate the interaction with other systems
 - 126 – Alternative binary packages meant to be used in development with
127 additional monitoring and diagnostic functionalities
- 128 • During development, the use of alternative packages allows testing the
129 core of the source code
- 130 • On building, unit tests should be run to ensure basic functionality
- 131 • During review, both the main and the test functionality should be checked
132 to provide as much coverage as possible
- 133 • Before integrating changes into main branches, basic automated integra-
134 tion tests on different hardware should be performed.
- 135 • After integrating changes into main branches, integration tests need to be
136 run to ensure no regressions are found.

137 Classifications

138 The first step towards solving a problem is to understand and describe it. This
139 section aims to do that by describing how different components are classified
140 and the criteria used for the classifications.

141 Components

142 For the purpose of this document the term **component** is used to refer to an
143 item to be tested. A component can match a package or a set of packages that
144 work together to provide a certain functionality. A component can be further
145 divided in sub-components if it is necessary to improve the testing of some
146 specific functionality.

147 Component metrics

148 The level of testing required in each case should be determined taking into
149 account different aspects:

- 150 • **Component source:** One of the key elements to understand the level of
151 test required is the source of the component. Under this category we can
152 find different cases:
 - 153 – Upstream components, for example **systemd**.
 - 154 – Upstream components with significant Apertis-specific changes, like
155 the **Linux kernel**.
 - 156 – Apertis-specific components, such as the **Apertis Update Man-**
157 **ager**.
- 158 • **Upstream activity:** Another key element to evaluate is how much a
159 component is actively developed:
 - 160 – High upstream activity, as an example the mainline **Linux kernel**,
161 **systemd**, **rust-coreutils**.
 - 162 – Medium upstream activity, like **OpenSSL** or **GnuPG**.
 - 163 – No or minimal upstream activity, like the tool **lqa**.
- 164 • **Component commonality:** Some components are more common than
165 others, depending on the functionality they provide, thus having them
166 used by a wider range of users:
 - 167 – High: Components under this category are common to any Apertis
168 image. A good example of this is **systemd**.
 - 169 – Normal: Components that are common to an important set of use
170 cases, such as **Docker**.
 - 171 – Low: This component has a very specific use case, like the **Maynard**
172 **graphical shell** (the reference shell).
- 173 • **Component criticality:** Some components are more critical than oth-
174 ers, depending on the functionality they provide and the use case. Since
175 Apertis is an Open Source distribution the criticality is evaluated from a
176 general perspective. However, product teams and Apertis derivatives in
177 general are encouraged to adjust this metric according to their specific
178 needs/use cases. The different criticalities used by Apertis are:
 - 179 – High: Components under this category provide a critical functionality
180 which is essential for the system. A good example of this is the **Linux**
181 **kernel**.
 - 182 – Normal: Components under this class provide a functionality that is
183 not critical for the system, but still required. For instance, **tracker**.
 - 184 – Low: This group provides functionality desirable but not required for
185 the system. An example for this category is **cups**.
- 186 • **Component target:** We use this category to differentiate components

187 based on their target environment:

- 188 – Target: Components aimed to be shipped on target devices.
- 189 – Development: Components specific to development environment.

190 Loops and types

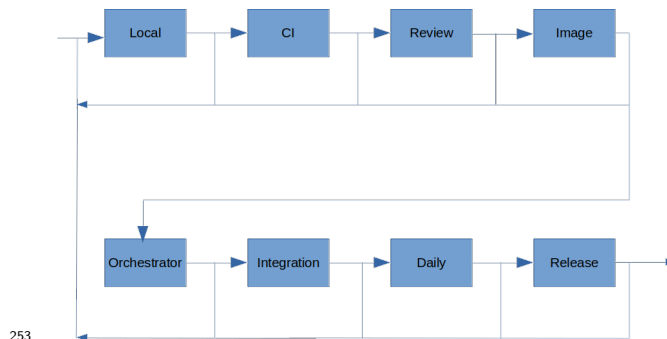
191 There are different stages of testing in the QA process which help to define
192 different level of loops. The purpose of testing is to spot deviations from the
193 expected behavior which is the first part in the loop. The second part is to
194 correct such deviations to provide the desired behavior. The iterations in each
195 loop are run until the result of the tests show the expected behavior. It is
196 important to note that every loop includes the previous ones as prerequisites,
197 making sure that any change in the code is evaluated in all the defined loops.
198 Additionally, it is interesting to note that, inner loops have less impact, since
199 they affect smaller groups.

200 The current defined loops are:

- 201 • Local loop: This loop is the closer one to development, which includes
202 developer testing and unit tests that are used during development. Based
203 on the results and after several iterations, the developer improves the
204 quality of the changes he is preparing before submitting a Merge Request.
205 As result of this loop a Merge Request is submitted.
- 206 • CI loop: This loop includes the previous one and goes a step beyond,
207 taking advantage of the Gitlab CI and its OBS integration. The proposed
208 changes in a Merge Request are tested with linters, license scanners and
209 built in OBS, which includes running its unit tests. Additionally simple
210 integration tests can be run to confirm the changes will not introduce any
211 regression. As result of this loop all the pipelines associated to the Merge
212 Request pass.
- 213 • Review loop: The review process is a key element in the Open Source cul-
214 ture, which allows developers to receive feedback of the proposed changes.
215 During this process the reviewer can suggest small changes/fixes or even
216 a complete different approach to reach the same goal. The feedback needs
217 to be addressed and any change will trigger additional iterations in this
218 loop. As a result the Merge Request is then merged or discarded/replaced.
- 219 • Image loop: This next loop focus in the image generation and initial in-
220 tegration testing, and is the first loop going beyond component isolation.
221 Here, some aspects of integration are evaluated, like the installation, pack-
222 age dependencies availability check, as well as license compliance checks.
223 As a result a set of reference images are made available for all the sup-
224 ported architectures.
- 225 • Orchestrator loop: One step further, a new loop is formed by the daily or-
226 chestrator runs, which builds the different types of images used by Apertis,

227 including Docker images used for development and CI, toolchains, flatpak
 228 runtimes, apart from the standard Apertis images.

- 229 • Integration tests, automated (LAVA) or manual: This next loop also in-
 230 cludes automated tests that are run in LAVA on actual devices of different
 231 architectures to confirm some behavior works as expected. Manual inte-
 232 gration tests aid to the process to cover for some functionality that cannot
 233 be tested automatically for example. In this loop platform specific tests
 234 can be added to validate hardware specific functionality.
- 235 • General use of daily images: An important loop is the general use
 236 by the community, developers and downstream distributions of
 237 daily/development images, which can fill the gap in case deviations
 238 from the expected behavior are detected and reported. Daily images
 239 use `-security` and `-updates` repositories which provide newer versions
 240 of the available packages. The distinction between these two types of
 241 repositories is important, since `-security` is used to publish high critical
 242 updates that should be applied without delay, while `-updates` is used
 243 for non-critical ones. The recommendation for production is to use
 244 the base repository plus `-security` to provide a reliable platform, while
 245 development can take advantage of the newer features already available
 246 in `-updates` which will include in the next release.
- 247 • Common use of release images: Similar to the previous loop, this one takes
 248 advantage of the common use of release images. The main difference here
 249 is the audience, since release images are the recommended ones in general
 250 and thus have a bigger userbase. During a release, the folding is applied,
 251 which consists in merging the changes from `-updates` and `-security` into
 252 the main branch, used as a starting point for the next release.



254 During these loops different types of tests are performed:

- 255 • Functional: Tests aimed to confirm that the desired functionality behaves
 256 as expected.
- 257 • Performance: Tests meant to verify the performance parameters are within
 258 a defined range.

- Security: Tests used to confirm that no known security vulnerability is found.

The way of implementing these types of tests is tied to the component under analysis, but all the aspects described above should be taken into account. As a result, guideline tests should be able to mimic real life usage as much as possible including very unlikely ones. It is a common issue that a component is tested only taking into account the functional aspect of it, but later on when it is tested under real life conditions and stress, performance parameters do not comply with the expectations.

The main purpose of testing is to spot deviations from the expected behavior which is the first step in any loop. The second step is to correct such deviation to provide the desired behavior.

Priorities

The testing process will result in either a success or a failure, in the last case, a bug report should be filled and triaged in order to prioritize the more critical issues:

- Critical: Deviations from the expected behavior in critical components that are unacceptable for a release use this priority. This type of issue is considered a release blocker and should be addressed with the highest priority. A good example of such issue would be when an image fails to boot.
- High: Deviations from the expected behavior in critical components that are not considered release blockers are triaged under this category. One example of such issue would be a crash in a critical component that only happens in a very specific scenario.
- Normal: Deviations from the expected behavior in non-critical components are triaged under this category. An issue in the language support could be a good example of this type of issue.
- Low: Deviations that do not affect the expected behavior fall into this category. As an example a log entry not expected or a minimal visual deviation.

Constraints

To develop a sustainable test strategy the constraints for testing also need to be taken into account, in order to provide the best possible trade off. Having this in mind, the following list describes the possible constraints.

- Environment availability: Tests require some type of environment to be executed, depending on the type of test this can include:
 - Development computer.
 - Server/Virtual Machine: e.g. Gitlab runners.

- 298 – External service: Gitlab, LAVA.
- 299 – Reference boards: iMX6 Sabrelite, Renesas R-Car M3, UP
- 300 Squared 6000. From the previous list, the availability of reference
- 301 boards to run a test is the most challenging one, since it implies hav-
- 302 ing boards of different types, models and architectures, in order to
- 303 be able to confirm the expected behavior of each test.
- 304 • Time availability: Even with the right environment time is always a chal-
- 305 lenge, due to different reasons:
 - 306 – Environment shared among different projects.
 - 307 – Test periodicity, since some tests are meant to be run regularly.
- 308 • Maintenance costs: The number of tests and supported boards/environments
- 309 have a direct impact in the maintenance costs of both software and hard-
- 310 ware.

311 Strategy

312 Since both the number of components and possible tests is huge, plus the con-
 313 straints involved, it is not possible to test everything, be it functionality, behav-
 314 ior or component. Based on this, the test strategy should provide a guidance
 315 to where to put the focus on in order to maximize the cost-benefit.

316 Additionally the test strategy should provide a reference to triage any issue
 317 found during the testing.

318 The strategy should also take advantage of the loops previously defined in order
 319 to spot any issue in the loop nearest to the local one in order to reduce its
 320 impact.

321 Classify components

322 To help selecting what tests to include or to support, the strategy suggests
 323 classifying each component or component group as follows:

324 Source

- 325 • 1: Apertis specific components
- 326 • 2: Upstream components with significant Apertis specific changes
- 327 • 3: Upstream components that are unmodified or with minimal changes

328 Upstream activity

- 329 • 1: Component with no or with minimal upstream activity
- 330 • 2: Component with medium upstream activity
- 331 • 3: Component with high upstream activity

332 Commonality

- 333 • 1: Component has high commonality

- 2: Component has normal commonality
- 3: Component has low commonality

Criticality

- 1: Component has high criticality
- 2: Component has normal criticality
- 3: Component has low criticality

Target

- 1: Component is meant to be used on target devices
- 2: Component is specific to development environment

The following table uses a set of components as example to illustrate the approach:

Component	Source	Activity	Commonality	Criticality	Target
Linux	3	3	1	1	1
Linux UML	3	1	1	1	1
AUM	1	1	1	1	1
OSTree	3	3	1	1	1
connman	3	3	1	1	1
rust-coreutils	2	2	1	1	1
dnsmasq	3	3	1	3	2
QA Report App	1	1	1	2	2
Pipewire	3	3	1	1	1
Bluez	3	3	2	1	1
Flatpak	3	2	2	1	1

Define tests required for each component

Based on the previous evaluation the recommended tests for each component needs to be evaluated using a clear guideline.

- Local loop
 - Developer tests
 - * Required: all components
 - Unit tests
 - * Required: components which support unit tests
 - * Encouraged: components that are under development, typically this is the case of Apertis specific components
- CI loop
 - Linters
 - * Encouraged: components that are under development, typically this is the case of Apertis specific components
 - License scan

- * Required: all components included in target images
- OBS build
 - * Required: all components
- Small integration tests
 - * Required: components with low community activity and high commonality/criticality that are under development, typically this is the case of Apertis specific components
 - * Encouraged: components with high commonality/criticality
 - * Desired: components with high commonality/criticality
- Review loop
 - Required: all components
- Image loop
 - Installation
 - * Required: components with normal or high commonality/criticality
 - * Desired: all components
 - License compliance
 - * Required: all components included in target images
- Orchestrator loop
 - Installation
 - * Required: components with normal or high commonality/criticality
 - * Desired: all components
- Integration tests automated(LAVA) or manual
 - Functional tests
 - * Required: all normal or high commonality/criticality
 - * Desired: all components
 - Performance tests
 - * Required: Apertis specific components
- Common use of daily images
 - Desired: User to test all components
- Common use of released images
 - Desired: User to test all components

Current status and gaps

The following table summarizes for each component in the sample the status according to the guidelines previously presented:

- 0: Some requirements are not meet for this loop
- 1: All the required tests are performed, additional tests should be encouraged
- 2: All the encouraged tests are performed, additional improvements can be done
- 3: All the desired tests are run

Component	Local	CI	Image	Orchestrator	Integration
Linux	3	1	3	3	2
Linux UML	1	0	3	3	2
AUM	2	0	3	3	3
OSTree	3	1	3	3	3
connman	2	1	3	3	3
rust-coreutils	2	0	3	3	3
dnsmasq	2	1	3	3	2
QA Report App	2	1	-	-	3
Pipewire	2	1	3	3	3
Bluez	2	1	3	3	3
Flatpak	2	1	3	3	3

Considerations for product teams

The previous sections provide general concepts around the Apertis test strategy from a general distribution perspective. However, since Apertis is used to build products these concepts needs to be applied in a way that supports the development process of such products.

The flexibility given by Apertis is vital to create an efficient workflow, and to make that happen some guidelines should be followed:

- Development should follow the **Apertis workflow** to enforce self containment and isolation, adding unit tests and Gitlab CI customizations to packages
- Components under development require special attention, so all the loops mentioned need to be used to maximize the benefits.
- Components under development need to include unit tests which exercise different aspects of the software.
- Components under development should include CI tests that are run in LAVA in the target hardware(s) where applicable. These tests should be run before changes are merged to confirm that certain functionality and/or performance of the new version are according to expectations.
- Since multiple teams work on the same product, it is important that tests are designed also based on other teams' expectations on functionality and performance.
- Close iteration between teams is needed when a team spots a regression introduced by other team. In that regard LAVA provides a single reference point to share tests, results and logs.
- Experience gained during development should be used not to only improve the component itself but also to improve the tests around it. This is a good way to avoid having the same issue in the future.
- Integration tests on target systems needs to be run, either automated or manual to validate the resultant image.

431 Apertis provides the infrastructure to support these guidelines and already im-
432 plements them. However, each product team needs to decide how to implement
433 them since each project has its own restrictions, requirements and scope.

434 As an example, a product team working in [IOT project scenario](#)³ can use Apertis
435 Fixed Function image recipe as reference and add the additional packages to
436 build its reference image. In such case, the product takes advantage of an
437 already well proven base reference, but needs to follow the above guidelines to
438 make sure that no regressions and to extend the test coverage to include the
439 new features.

440 In such scenario, new packages to provide application specific logic should:

- 441 • Include [unit tests](#)⁴
- 442 • Include [CI tests running in LAVA](#)⁵ if possible
- 443 • Run [Apertis test](#)⁶ for the still valid functionality
- 444 • Run [additional either automate or manual tests](#)⁷ to check the new func-
445 tionality

446 Follow up tasks

447 As this strategy provides general guidelines to avoid gaps between expectations
448 and actual results some follow up tasks are suggested:

- 449 • Identify testing gaps: This document provides metrics for components and
450 sets expectations regarding the test loops that should be in place. Based
451 on these statements a more elaborated list of components and testing gaps
452 needs to be built.
- 453 • Provide CI integration tests to run before merging: As described, compo-
454 nents under development are the ones that could add instability to the
455 development of a product. To minimize this risk, CI should run different
456 types of tests before merging changes. The support for these kind of test
457 is described in [Apertis package centric tests](#)⁸ which takes into account the
458 following considerations:
 - 459 – Should be based in pre-hooks to make it easier to extend to add
460 additional checks, such as new linters.
 - 461 – Should include tests on LAVA on the target hardware when applica-
462 ble.

³https://apertis-website-0b3586.pages.apertis.org/overview/platform__overview/#industrial-iot-scenario

⁴https://apertis-website-0b3586.pages.apertis.org/guides/testing/unit_testing/

⁵<https://apertis-website-0b3586.pages.apertis.org/guides/testing/apertis-packages-testing/>

⁶<https://apertis-website-0b3586.pages.apertis.org/qa/test-data-reporting/>

⁷https://apertis-website-0b3586.pages.apertis.org/qa/test_cases_guidelines/

⁸<https://apertis-website-0b3586.pages.apertis.org/guides/testing/apertis-packages-testing/>

- 463 – Should be configurable to be able to avoid the overhead of building
464 and testing components if it is not necessary, for instance during the
465 folding.
- 466 • Provide guidelines for developers to run local tests on different architec-
467 tures using emulation, such as QEMU virtual machines or docker images.
- 468 • Provide a way to work with an interactive remote hardware environment
469 for developers for debugging. LAVA is not meant to run in an interactive
470 session with developers, however, a low level service could be implemented
471 to allow developers to share the hardware and run debugging sessions.
- 472 • Provide a way to run visual regression tests. This type of test is very useful
473 when developing applications that provide user interfaces since it allows
474 catching unexpected changes earlier. An initial task should be to identify
475 tools to be used to provide this types of tests and provide a sample test
476 for a package.
- 477 • Robot Framework integration with LAVA has been planned, from which
478 an implementation phase should be started.