



Infrastructure monitoring and testing

1 Contents

2	The Apertis infrastructure	2
3	Deployment types	3
4	Traditional package-based deployments	3
5	Docker containers	3
6	Docker Compose	4
7	Kubernetes Helm charts	4
8	Maintenance, monitoring and testing	4
9	Ensuring all components are up-to-date	4
10	Minimizing downtimes	5
11	Reacting on regressions	5
12	Keeping the users' data safe	5
13	Checking that data across services is coherent	6
14	Providing fast recovery after unplanned outages	6
15	Verify functionality	6
16	Monitoring and communicating availability	7
17	Preventing performance degradations that may affect the user	
18	experience	7
19	Optimizing costs	7
20	Testing changes	8

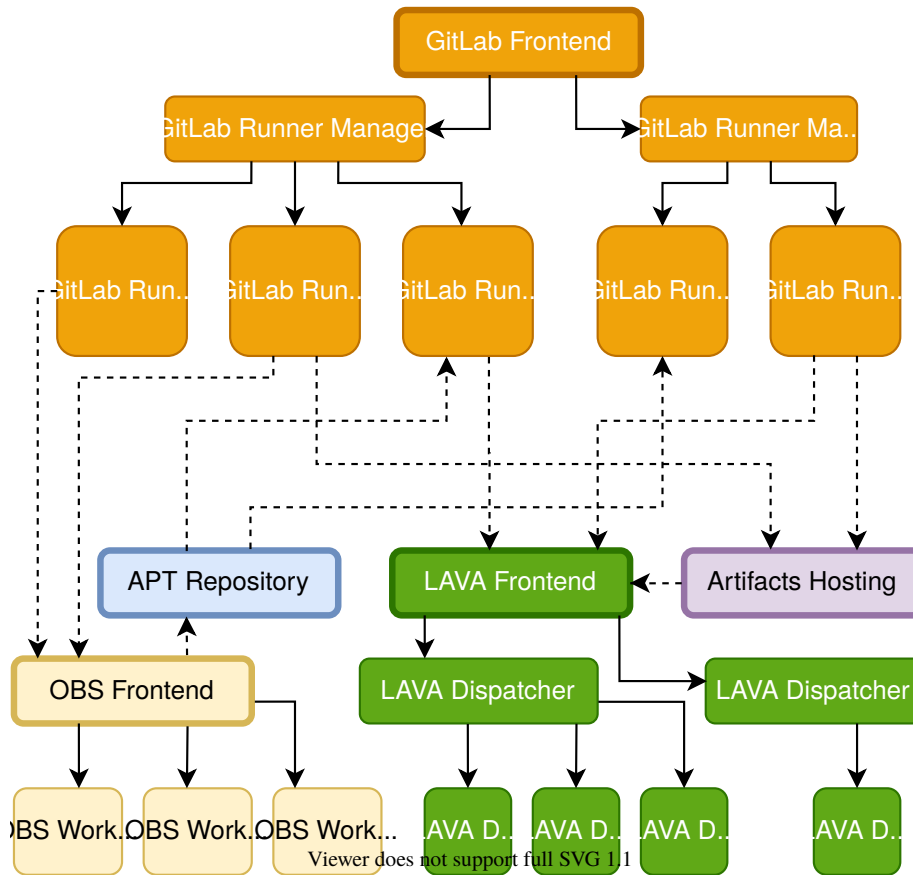
21 The Apertis infrastructure is itself a fundamental component of what Apertis
22 delivers: its goal is to enable developers and product teams to work and collab-
23 orate efficiently, focusing on their value-add rather than starting from scratch.

24 This document focuses on the components of the current infrastructure and
25 their monitoring and testing requirements.

26 The Apertis infrastructure

27 The Apertis infrastructure is composed by a few high level components:

- 28 • GitLab
- 29 • OBS
- 30 • APT repository
- 31 • Artifacts hosting
- 32 • LAVA



33

34 From the point of view of developers and product teams, GitLab is the main
 35 interface to Apertis. All the source code is hosted there and all the workflows
 36 that tie everything together run as GitLab CI/CD pipelines, which means that
 37 its runners interact with every other service.

38 The Open Build Service (OBS) manages the build of every package, dealing
 39 with dependency resolution, pristine environments and multiple architectures.
 40 For each package, GitLab CI/CD pipelines take the source code hosted with
 41 Git and pushes it to OBS, which then produces binary packages.

42 The binary packages built by OBS are then published in a repository for APT,
 43 to be consumed by other GitLab CI/CD pipelines.

44 These pipelines produce the final artifacts, which are then stored and published
 45 by the artifacts hosting service.

46 At the end of the workflow, LAVA is responsible for executing integration tests
 47 on actual hardware devices for all the artifacts produced.

48 **Deployment types**

49 The high-level services often involve multiple components that need to be de-
50 ployed and managed. This section describes the kind of deployments that can
51 be expected.

52 **Traditional package-based deployments**

53 The simplest services can be deployed using traditional methods: for instance
54 in basic setups the APT repository and artifacts hosting services only involve a
55 plain webserver and access via SSH, which can be easily managed by installing
56 the required packages on a standard virtual machine.

57 Non-autoscaling GitLab Runners and the autoscaling GitLab Runners Manager
58 using Docker Machine are another example of components that can be set up
59 using traditional packages.

60 **Docker containers**

61 An alternative to setting up a dedicated virtual machine is to use services pack-
62 aged as single Docker containers.

63 An example of that is the [GitLab Omnibus Docker container](#)¹ which ships all
64 the components needed to run GitLab in a single Docker image.

65 The GitLab Runners Manager using Docker Machine may also be deployed as
66 a Docker container rather than setting up a dedicated VM for it.

67 **Docker Compose**

68 More complex services may be available as a set of interconnected Docker con-
69 tainers to be set up with [Docker Compose](#)².

70 In particular OBS and LAVA can be deployed with this approach.

71 **Kubernetes Helm charts**

72 As a further abstraction over virtual machines and hand-curated containers
73 most cloud providers now offer Kubernetes clusters where multiple components
74 and services can be deployed as Docker containers with enhanced scaling and
75 availability capabilities.

76 The [GitLab cloud native Helm chart](#)³ is the main example of this approach.

¹<https://docs.gitlab.com/omnibus/docker/>

²<https://docs.docker.com/compose/>

³<https://docs.gitlab.com/charts/>

77 Maintenance, monitoring and testing

78 These are the goals that drive the infrastructure maintenance:

- 79 • ensuring all components are up-to-date, shipping the latest security fixes
80 and features
- 81 • minimizing downtime to avoid blocking users
- 82 • reacting on regressions
- 83 • keeping the users' data safe
- 84 • checking that data across services is coherent
- 85 • providing fast recovery after unplanned outages
- 86 • verify functionality
- 87 • preventing performance degradations that may affect the user experience
- 88 • optimizing costs
- 89 • testing changes

90 Ensuring all components are up-to-date

91 Users care about services that behave as expected and about being able to use
92 new features that can lessen their burden.

93 Deploying updates timely is a fundamental step to address this need.

94 Traditional setups can use tools like [unattended-upgrades](#)⁴ to automatically de-
95 ploy updates as soon as they become available without any manual intervention.

96 For Docker-based deployment the `pull` command needs to be executed to ensure
97 that the latest images are available and then the services need to be restarted.

98 Tools like [watchtower](#)⁵ can help to automate the process.

99 However, this kind of automation can be problematic for services where high
100 availability is required, like GitLab: in case anything goes wrong there may be
101 a considerable delay before a sysadmin becomes available to investigate and fix
102 the issue, so explicitly scheduling manual updates is recommended.

103 Minimizing downtimes

104 To minimize the impact on users of the downtime due to the updates it is
105 recommended to schedule them during a window where most users are inactive,
106 for instance during the weekend.

107 For example, every Saturday the Apertis sysadmin team checks if a new GitLab
108 stable release has been published and applies the update, currently using the
109 Omnibus container.

110 The team managing the much larger, Kubernetes-based [installation used by](#)
111 [freedesktop.org](#)⁶ have a policy where new patch versions are deployed with no

⁴<https://wiki.debian.org/UnattendedUpgrades>

⁵<https://github.com/containrrr/watchtower>

⁶<https://gitlab.freedesktop.org>

112 prior testing during the week, while new minor/major versions are deployed
113 during a weekend time window.

114 To minimize downtime the Kubernetes-based cloud-native install lets sysadmins
115 stagger component upgrades to reduce downtime, for instance by upgrading the
116 Gitaly component at a different time from the Rails frontend.

117 **Reacting on regressions**

118 Some updates may fail or introduce regressions that impact users. In those cases
119 it may be necessary to roll back a component or an entire service to a previous
120 version.

121 Rollbacks are usually problematic with traditional package managers, so this
122 kind of deployment is acceptable only for service where the risk of regressions
123 is very low, as it is for standard web servers.

124 Docker-based deployment make this much easier as each image has a unique
125 digest that can be used to control exactly what gets run.

126 **Keeping the users' data safe**

127 In cloud deployments the object storage services is a common target of attacks.

128 Care must be taken to ensure all the object storage buckets/accounts have strict
129 access policies and are not public to prevent data leaks.

130 Deleting unused buckets/accounts should also be done with care if other resource
131 point to them: for instance, in some cases it can lead to [subdomain takeovers](#)⁷.

132 **Checking that data across services is coherent**

133 With large amounts of data being stored across different interconnected services
134 it's likely that discrepancies will creep in due to bugs in the automation or due
135 to human mistakes.

136 It is thus important to cross-correlate data from different sources to detect
137 issues and act on them timely. The [Apertis infrastructure dashboard](#)⁸ currently
138 provides such overview ensuring that the packaging data is consistent across
139 GitLab, OBS, the APT repository and the upstream sources.

140 **Providing fast recovery after unplanned outages**

141 Unplanned outages may happen for a multitude of causes:

- 142 • hardware failures
- 143 • human mistakes

⁷<https://www.we45.com/blog/how-an-unclaimed-aws-s3-bucket-escalates-to-subdomain-takeover>

⁸<https://infrastructure.pages.apertis.org/dashboard/>

144 • ransomware attacks

145 To mitigate their unavoidable impact a good backup and restore strategy has
146 to be devised.

147 All the service data should be backed up to separate locations to make them
148 available even in case of infrastructure-wide outages.

149 For services it is important to be able to re-deploy them quickly: for this reason
150 it is strongly recommended to follow a “cattle not pets”⁹ approach and be able
151 to deploy new service instances with minimal human intervention.

152 Docker-based deployment types are strongly recommended since the recovery
153 procedure only involves the re-download of pre-assembled container images once
154 data volumes have been restored from backups.

155 Traditional approaches instead involve a lengthy reinstallation process even
156 if automation tools such as Ansible are used, with good chances that the re-
157 provisioned system differs significantly from the original one, requiring a more
158 intensive revalidation process.

159 On cloud-based setups it is strongly recommended to use automation tools like
160 Terraform¹⁰ to be able to quickly re-deploy full services from scratch, potentially
161 on different cloud accounts or even on different cloud providers.

162 **Verify functionality**

163 Apertis strongly pushes for automating as much as possible every workflow, to
164 let developers focus on adding value rather than wasting time on repetitive tasks
165 and to reduce the chance of manual errors.

166 Such automation is usually implemented through GitLab CI/CD pipelines. Since
167 those are the tools that developers use in their day-to-day operation it is reason-
168 able to assume that in most cases the pipelines do not need special provisions
169 to ensure they work correctly and that developers will detect issues quickly.

170 Whilst this is generally the case, some pipelines may be more complex and
171 critical so it is recommended to set up dedicated test procedures for them: for
172 instance, the GitLab-to-OBS packaging pipeline now includes a [fully automated](#)
173 [test procedure](#)¹¹ to detect issues before they impact developers.

174 **Monitoring and communicating availability**

175 Timely detecting unplanned outages is as important as properly communicating
176 planned downtimes.

⁹<http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/>

¹⁰<https://www.terraform.io/>

¹¹https://gitlab.apertis.org/infrastructure/ci-package-builder/-/merge_requests/75

177 A common approach is to set up a global status page that reports the availability
178 of each service and provides information to users about incidents being addressed
179 and planned downtimes.

180 The Apertis project uses the status page service provided by [UptimeRobot](https://uptimerobot.com/R8MlxtrZXO)¹²
181 to track the availability of its user facing services. This is accessible at <https://stats.uptimerobot.com/R8MlxtrZXO>.
182

183 **Preventing performance degradations that may affect the user experience** 184

185 As the project grows, the needs of the infrastructure grow as well to keep the
186 user experience good.

187 Collecting metrics and tracking them over time is important to spot the area
188 that need interventions.

189 Among the many solutions available to create customizable dashboards out of
190 metrics, Grafana is well integrated with GitLab and it is [already included in the Omnibus distribution](#)¹³.
191

192 **Optimizing costs**

193 Part of infrastructure maintenance is the continuous effort to efficiently use the
194 available budget, optimizing cost without negatively affecting the user experi-
195 ence. This is particularly important on cloud deployments which provide a large
196 portfolio of options with wildly different and somewhat hard to anticipate costs.

197 There are many ways to improve budget efficiency, here are a few examples in
198 no particular order:

- 199 • use different VM sizes for different purposes to avoid overspending on
200 powerful machines that are underutilized
- 201 • use cloud container services to host applications rather than hosting them
202 on a dedicated VM
- 203 • deploy multiple services on the same Kubernetes cluster, provided that
204 there are no big trust boundaries between them: for instance, having the
205 GitLab runners in the same cluster as the main GitLab instance is not a
206 good idea as the runners are less trusted (they let developers run arbitrary
207 code)
- 208 • on cloud setups, minimize the outgoing network traffic
- 209 • minimize storage consumption by reducing the artifacts size and with
210 strict cleanup policies

¹²<https://uptimerobot.com/status-page/>

¹³<https://docs.gitlab.com/omnibus/settings/grafana.html>

211 **Testing changes**

212 Applying changes to production services can be risky if not done with care, as
213 it may introduce regressions or, in extreme cases, data losses.

214 So far Apertis has been relying on services with proven track records of stable
215 updates and the overall architecture of the infrastructure has been quite stable
216 since the introduction of GitLab, so no big configuration change has ever been
217 required. In this scenario, closely tracking stable upstream releases and deploy-
218 ing them on a weekend not long after they get published has worked well with
219 no major incidents.

220 For instance, GitLab is updated weekly and the Apertis instance is always using
221 the last point release, making things easier for major updates as that's what
222 the [upstream documentation](#)¹⁴ suggests, and no significant issues have been
223 registered.

224 It is important to read the release notes before applying updates, to learn about
225 the pending deprecations and the versions in which they will become mandatory
226 transitions. In the case of GitLab, the only disruptive transition has been
227 a need to move from Postgres 6.x to 11.x as it required some action on the
228 database files. Even in that case GitLab supported both 11.x and 6.x in parallel
229 for approximately a year, giving administrators plenty of time to schedule the
230 activity. In addition, it was possible to do the migration out of band, to minimize
231 the downtime.

232 However, larger changes may be too risky to be introduced directly in produc-
233 tion. In these cases it is recommended to set up a test environment where the
234 changes can be evaluated without affecting users.

235 Automation tools like Terraform are recommended to be able to set up dedicated
236 test environments with little effort and to reliably reproduce the changes in
237 production once they are deemed safe.

¹⁴<https://docs.gitlab.com/ce/policy/maintenance.html#upgrading-major-versions>