Infrastructure monitoring and testing

Contents

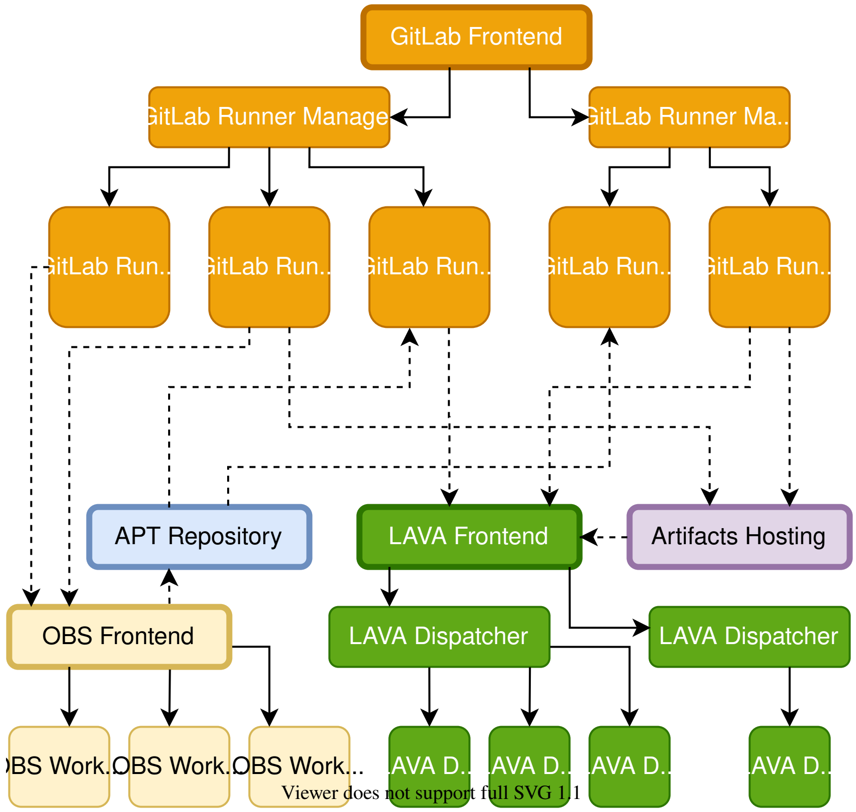
The Apertis infrastructure is itself a fundamental component of what Apertis delivers: its goal is to enable developers and product teams to work and collaborate efficiently, focusing on their value-add rather than starting from scratch.

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

The Apertis infrastructure

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

- GitLab
- OBS
- APT repository
- Artifacts hosting
- LAVA



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

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

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

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

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

Deployment types

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

Traditional package-based deployments

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

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

Docker containers

An alternative to setting up a dedicated virtual machine is to use services packaged as single Docker containers.

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

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

Docker Compose

More complex services may be available as a set of interconnected Docker containers to be set up with Docker Compose[2].

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

Kubernetes Helm charts

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

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

[1] https://docs.gitlab.com/omnibus/
[2] https://docs.docker.com/compose/
[3] https://docs.gitlab.com/charts/

Maintenance, monitoring and testing

These are the goals that drive the infrastructure maintenance:

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

Ensuring all components are up-to-date

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

Deploying updates timely is a fundamental step to addess this need.

Traditional setups can use tools like `unattended-upgrades`[4] to automatically deploy updates as soon as they become available without any manual intervetion.

For Docker-based deployment the `pull` command needs to be executed to ensure that the latest images are available and then the services need to be restarted. Tools like `watchtower`[5] can help to automate the process.

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

Minimizing downtimes

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

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

The team managing the much larger, Kubernetes-based installation used by 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

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

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

Reacting on regressions

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

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

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

Keeping the users'data safe

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

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

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

Checking that data across services is coherent

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

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

Providing fast recovery after unplanned outages

Unplanned outages may happen for a multitude of causes:

- hardware failures
- 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/

- ransomware attacks

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

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

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

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

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

On cloud-based setups it is strongly recommended to use automation tools like Terraform[10] to be able to quickly re-deploy full services from scratch, potentially on different cloud accounts or even on different cloud providers.

Verify functionality

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

Such automation is usually implemented though GitLab CI/CD pipelines. Since those are the tools that developers use in their day-to-day operation it is reasonable to assume that in most cases the pipelines do not need special provisions to ensure they work correctly and that developers will detect issues quickly. For instance, changes to the image recipes[11] are tested before landing and the pipelines are run on a daily schedule, which means that regressions can get caught timely.

Whilst this is generally the case, some pipelines may be more complex and critical so it is recommended to set up dedicated test procedures for them: for instance, the GitLab-to-OBS packaging pipeline now includes a fully automated test procedure[12] to detect issues before they impact developers. The packaging pipeline is by nature not self-contained as it operates on the packaging repositories: this makes setting up the test environment particularly difficult. By only relying on manual testing in the past many regression were not caught, so now

[9] http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/
[10] https://www.terraform.io/
[11] https://gitlab.apertis.org/infrastructure/apertis-image-recipes/
[12] https://gitlab.apertis.org/infrastructure/ci-package-builder/-/merge_requests/75

on each change a pipeline tests the actual packaging pipeline by emulating a developer that commits some changes to a packages and releases it to be built by OBS: this effort now allows us to catch issues before the affected changes get landed to the branch used by all packages.

Monitoring and communicating availability

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

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

The Apertis project uses the status page service provided by UptimeRobot[13] to track the availability of its user facing services. This is accessible at https://stats.uptimerobot.com/R8MlxtrZXO.

Preventing performance degradations that may affect the user experience

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

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

Among the many solutions available to create customizable dashboards out of metrics, Grafana is well integrated with GitLab and it is already included in the Omnibus distribution[14], making it a reasonable choice.

Metrics should then be configured and monitored, and monitors for other services, from OBS to artifacts storage, should be put in place to track the overall infrastructure.

Optimizing costs

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

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

- use different VM sizes for different purposes to avoid overspending on powerful machines that are underutilized

[13] https://uptimerobot.com/status-page/
[14] https://docs.gitlab.com/omnibus/settings/grafana.html

- use cloud container services to host applications rather than hosting them on a dedicated VM
- deploy multiple services on the same Kubernetes cluster, provided that there are no big trust boundaries between them: for instance, having the GitLab runners in the same cluster as the main GitLab instance is not a good idea as the runners are less trusted (they let developers run arbitrary code)
- on cloud setups, minimize the outgoing network traffic
- minimize storage consumption by reducing the artifacts size and with strict cleanup policies

Testing changes

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

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

For instance, GitLab is updated weekly and the Apertis instance is always using the last point release, making thinks easier for major updates as that's what the upstream documentation[15] suggests, and no significant issues have been registered.

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

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

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

[15] https://docs.gitlab.com/ce/policy/maintenance.html#upgrading-major-versions