CONTINUOUS DELIVERY IN TELECOMMUNICATION NETWORK ENVIRONMENTS

Continuous Delivery in Telecommunication Network Environments

by NGMN Alliance

Version:	**1.0**
Date:	**5th September 2019**
Document Type:	**Final Deliverable (approved)**
Confidentiality Class:	**P - Public**

Editor / Submitter:	**Nils Mäthner, Simon Baatz, Wolfgang Wirths - Deutsche Telekom** **Jaesuk Ahn, Taeil Choi - SK Telecom**
Approved by / Date:	**NGMN Board, 19th September 2019**

Address:
ngmn e. V.
Großer Hasenpfad 30 • 60598 Frankfurt • Germany
Phone +49 69/9 07 49 98-0 • Fax +49 69/9 07 49 98-41

Executive Summary

Telecommunications industry is undergoing a fundamental transformation to become more software-centric and cloud-based, paving the way towards a more agile service environment and higher operational automation.

On the journey to DevOps and applying practices such as Continuous Integration (CI) and Continuous Delivery (CD), network operators at some point in time hit a challenge we called the 'air gap'. It is caused by the fact, that production networks are strictly segregated from test networks and development environment networks for security and operational reasons. This gap between CI automation in one network zone and deployment automations in other network zones hinders seamless CD automation.

This white paper focuses on solving the 'air gap' challenge.

In subsequent sections this white paper

- Outlines the target scenario for CD in telecommunication network environments
- Identifies key concepts for a model-based and reusable solution
- Examines various aspects of lifecycle management, putting CD into an even larger context

The work builds on practical experience of automating the delivery chain for multiple applications at DT, complemented by mapping the concept to SKT technology stack. It is motivated by identified needs for alignment with future industry activities around application and service orchestration and zero touch automation such as ZSM.

This document is primarily aimed at network service providers and VNF developers who are involved in establishing joint delivery pipelines for continuous software and configuration rollout. It is also useful to industry fora for building consensus and driving standardization. We expect this paper to be beneficial for anyone who wants to have a good reference they can develop their own customized system based upon.

The authors welcome feedback on this white paper and the evolution towards true continuous deployment. Contact details are provided at the end.

Glossary

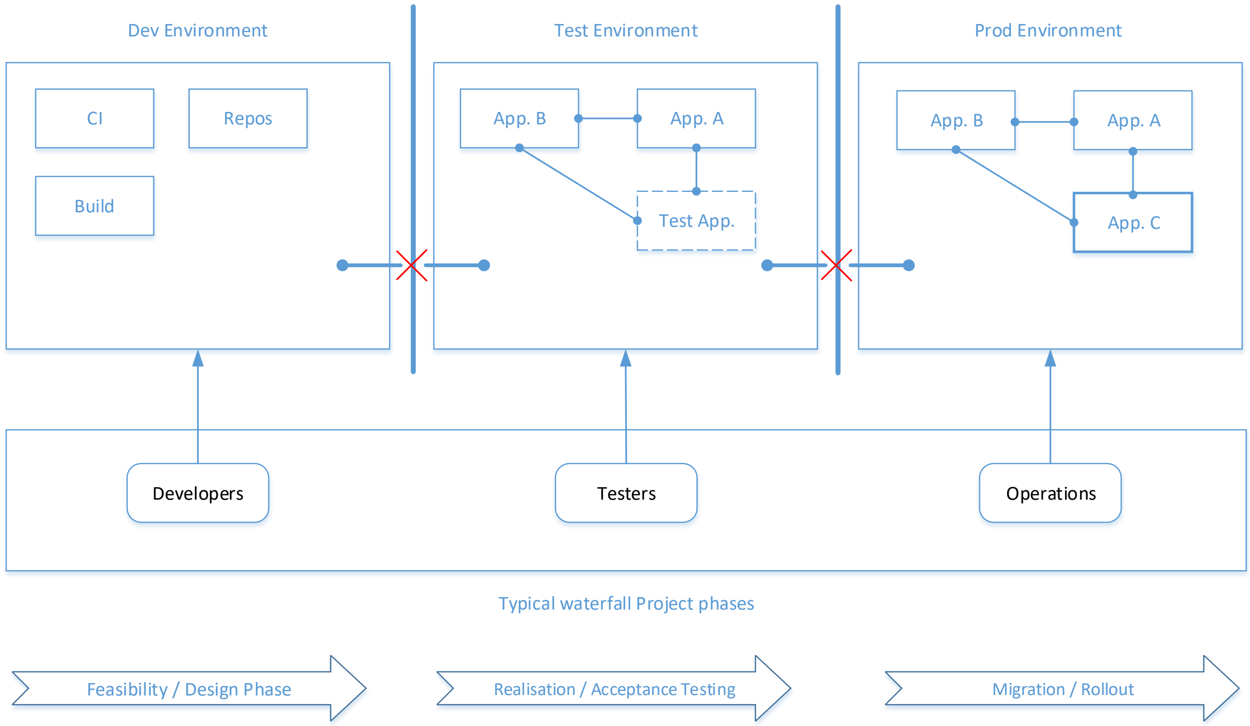
API	Application Programming Interface
CD	Continuous Delivery
CI	Continuous Integration
ED	Environment Descriptor
EMS	Element Management System
ETSI	European Telecommunications Standards Institute
GUI	Graphical User Interface
IETF	Internet Engineering Task Force
NFV	Network Functions Virtualization (specified by ETSI)
NFV-O	NFV-Orchestrator (part of ETSI NFV architecture)
RD	Release Descriptor
RDR	Release Descriptor Repository
RPM	Red Hat Package Manager (software package management format)
SDN	Software Defined Network
TOSCA	Topology and Orchestration Specification for Cloud Applications (specified by OASIS)
VM	Virtual Machine
VNF	Virtual Network Function (part of ETSI NFV architecture)
VNFC	VNF Component (part of ETSI NFV architecture)
VNF-M	VNF-Manager (part of ETSI NFV architecture)
YANG	(Yet Another Next Generation) data modelling language specified by IETF
ZSM	Zero touch network and Service Management (specified by ETSI)

Contents

1 INTRODUCTION

Telecommunications industry is undergoing a fundamental transformation; networks that traditionally relied on vendor-specific appliances become more software-centric and cloud-based. Network Functions Virtualisation (NFV) and Software Defined Networking (SDN) are two key concepts that are synonymous for this change. They pave the way towards a more agile service environment and higher operational automation. But the change is not only about technology, it is also about culture and methodologies. Operators' technology organizations start to transform from tayloristic structures into more DevOps friendly ones.

Practices such as Continuous Integration (CI) and Continuous Delivery (CD) are seen as typical ingredients to high-performing technology teams. We observe a massive uptake of CI adoption by teams moving along the path towards DevOps. That's great. But when it comes to CD, network operators' environments present a specific challenge: the air gap!



As operators' business is strongly related to the network, there is a strict segregation between production networks and test networks for operational as well as security reasons. This has also impact on data centers and the ability to adopt CD practices on Virtual Network Function (VNF) and Network Services levels. Established security rules advise to connect test systems or virtualized test VNFs to test networks only and to connect production VNFs to production networks only. Connectivity between production networks and test networks is usually not permitted. Development and office networks are often further segregated. So how can we move a software release from CI environment to test and production environments consistently, securely and with automation?

This is our air gap challenge. This paper discusses how we tackle it.

2 TARGET SCENARIO

To efficiently enable a DevOps style of working within cross-functional teams, we need to bridge the air gap between automated builds executed in CI environments and automated deployments executed in test and production environments. The architectural approach is driven by the following goals:

- Secure
  We aim for a solution which is able to move software releases being composed of multiple software artifacts across network boundaries in a controlled way. This implies that strict security controls can be established at network boundaries. Filtering of administrative traffic by web application firewalls is self-evident. Furthermore, attack surface should be minimized by placing activity controls into the network domain of higher sensitivity and trust, i.e. high trust area is preferably in control to pull data from lower trust areas.
- CI automation friendly
  Working with version control repositories is distinctive for state-of-the-art CI. We aim for a solution which smoothly picks up release builds from repositories and which can be seamlessly triggered by CI automation. This gives autonomy to the DevOps teams to decide which steps of delivery they want to automate, and which steps they prefer to keep under manual control.
- Deployment technology agnostic
  Typical for an operator's system landscape is to deal with elements from multiple vendors, each of them bringing in specific flavors of deployment and configuration automation. One might prefer Chef for configuration, others use e.g. Ansible. Some deliver virtual machines to be deployed on OpenStack using e.g. Heat API, others deliver Containers to be deployed via Kubernetes. Future proof CD solution should be able to support all those flavors by being sufficiently independent of them.
- Model driven approach
  Last but not least, CD solution should be easy to use from an application's point of view. A model-based approach provides abstraction between user perspective and implementation. Such model should be human readable as well as machine parseable.

3 SOLUTION APPROACH

3.1 Release Model

Modelling starts with the question: What is a release?

On a semantic level, we define a release as the complete set of items needed for instantiating a software-based function. Granularity of such function depends on decisions of responsible DevOps team how they want to shape their release strategy and dependency management. The team might prefer to manage a complete VNF as single release with all internal dependencies resolved by CI or to deliver VNF components as independent releases and manage dependencies outside.

On a structural level, a release is described by a list of artifacts, each of them with its individual version. We call this list a release descriptor. A release descriptor can be compared to a packaging list; when we receive a shipment from an online store, it comes with a list of items that should be inside. In our case the release descriptor is a yaml file for sake of being human readable and machine parseable.

Following is an example of a simple release descriptor:

```
descriptor_version: 1
name: application
version: 1.1.0
summary: The main application
vendor: Acme
artifacts:
  helm/application-1.1.0.tgz:
    type: file
    size: 1329
    sha256: fa454198b29cb4adbdaf2a26169a45b72a548e29b77aa157530b33420d264192
  acme/application_exporter@sha256:ca2b7dd0b0ce0575581b78412281daca98e19faf78e63737c2
dee41b1272777f:
    type: docker
    tags:
    - application-1.1.0
    - 0.2.7
  acme/application@sha256:841efc80d56e4b67b223d0a07525ba622587433bef7820fef9e9089f386
00282:
    type: docker
    tags:
    - application-1.1.0
    - 1.1.0
```

A release is identified by its `name` and `version`, i.e. `application` and `1.0.0` in this example. As a common practice, we use semantic versioning for releases. Additionally, the release descriptor has meta-data fields that allow to describe the release further, like `summary` and `vendor`.

The `artifacts` mapping is the ‘core’ of the release descriptor defining the artifacts the release is comprised of. Our example release consists of one file and two Docker images. An important property of release descriptors is to pin the artifacts by a cryptographic digest. This allows to verify the integrity of a release before delivering it. The digest must be given explicitly by an attribute (like for file artifacts) or may be part of the native notation (like for Docker artifacts)

However, the Docker CAS (Content Addressable Storage) notation is cumbersome to work with. Therefore, the release descriptor allows to specify `tags` that may be used to retrieve the respective image.

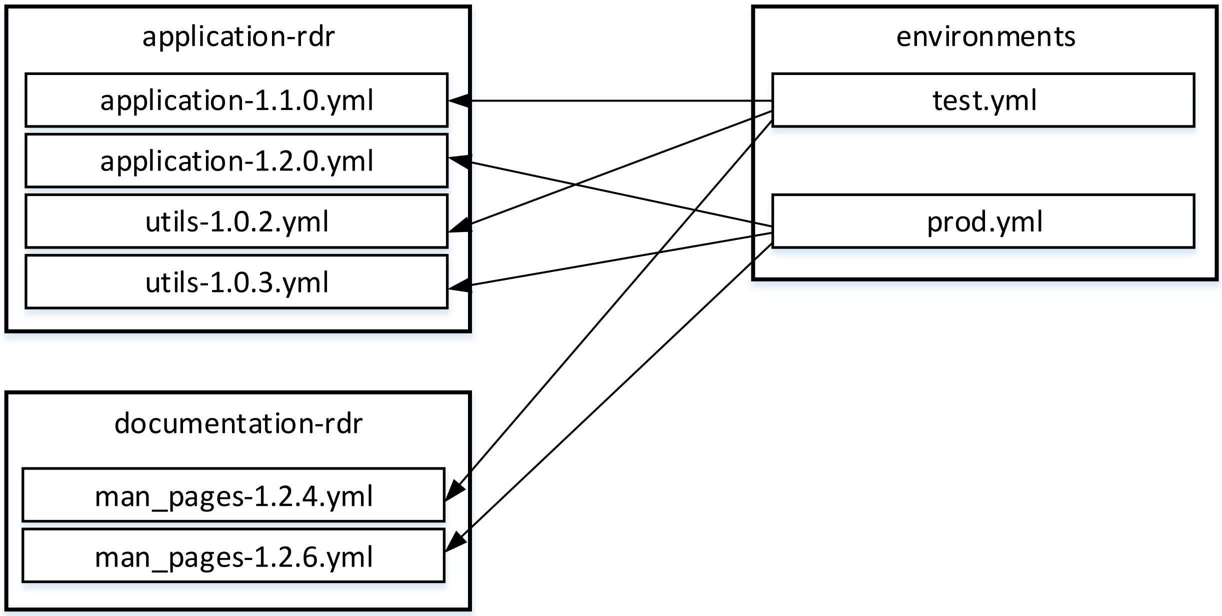
To ease the generation of release descriptors in the proper format, we provide additional tooling to developers and vendors. It can be used to generate release descriptors based on local files, artifacts in repository servers and Docker images in Docker registries.

3.2 Environment Model

Next question is to model where the release should go to. Releases are deployed to different environments which may vary by project. Most projects have at least one test environment and one production environment. In addition, there might be a need for a kind of stable test environment used by other teams or a pilot environment used for friendly user validation in pre-production scenarios.

We model assignment of releases to environments by means of a declaration we call environment descriptor. Placing a release reference (i.e. the reference to a release descriptor) into an environment descriptor semantically means to approve that the release shall be visible to this environment. Releases are always made available as a whole. Our solution verifies the availability and integrity of the specified artifacts before adding it to an environment.

On a structural level, an environment descriptor is a list of releases, in our case a yaml file too.



The environment descriptor on a first level refers to a release descriptor repository (in this example `application-rdr`). On the next level, a given release is referred with all versions that shall be visible in the target environment in this case `application-1.2.0`, `utils-1.0.3` and `man_pages-1.2.6`:

Name of the descriptor file: prod.yaml

```
application-rdr:
  application:
    -1.2.0
```

```
  utils:
    -1.0.3

documentation-rdr:
  man_pages:
    -1.2.6
```

While the above example describes the general model, the technology stack you are based on, can motivate variations in RD and ED detailed design. Aiming on support of multiple technology stacks favors RD design to be agnostic of technology stack as described above. Optimizing for a dedicated technology stack may prefer deeper integration and reuse of its native concepts.

For example, in SKT scenario delivering "OpenStack and related Infrastructure software" on top of Kubernetes cluster, a concept of explicit ED is applied in combination with RD based on native package descriptions such as helm charts. Note that the versions in the example are arbitrary version, not a real version.

Deployment-1.0.0

- docker images 1.0.0 (artifact with versioning)
- helm charts 1.0.0 (a package descriptor, managed as artifact with versioning)
- armada manifests 1.0.0 (a manifest, managed as artifact with versioning)
- kubespray 2.0.0 (deployment code for kubernetes, it also has list of necessary docker images)
- ceph-ansible 2.1.0 (deployment code for ceph)
- bifrost-ansible 2.2.0 (deployment code for baremetal provisioning)

Armada is one of sub-projects in Airship project. Armada manifest contains list of helm charts with their versions, and in turn, each helm chart contains docker image tags used in that chart. In this way, deployment 1.0.0 acts as an explicit ED and each element has its native RD.

Currently, SKT's looking into cloud native technologies to do everything in declarative way from bare metal provisioning up to OpenStack and other related software. In other words, a standard manifest will be given a set of APIs to deploy the entire infrastructure and related software. In this way, ED and RD role will be easily combined.

3.3 Version Control and CI Integration

Another key concept is putting release descriptors and environment descriptors under version control. They are stored in git repositories for the following reasons:

- Access control and authorization to git repositories is a "solved problem" in enterprise environments. Often there are requirements to differentiate access to steering repositories. It is usually desirable to enable suppliers to directly upload artifacts and related release descriptors to the CI collaboration platform. But policies controlling artifact propagation into production environment should be limited to operator's operational staff. Furthermore, multi-vendor scenarios might require segregating vendor-related repositories from each other. These requirements can be met by means of GitLab role-based

access and a repository structure adjusted to organizational needs.
The release descriptor repository and the environment repositories can be split and managed by dedicated subgroups that can add/modify/delete release and environment descriptors.

- git provides a change history out of the box.
  The delivery pipeline service itself is automated using steering information that is kept under version control to control what artifact release shall be visible in a given target environment. Keeping the delivery pipeline control information under version control guarantees that any change that is rolled out is auditable and documented.

- git allows to sign commits, which may be used to verify the integrity of releases and to further authorize promotion of releases into environments. This is done by provisioning the delivery pipeline with the public keys of the project members who commit into the steering repositories, e.g. add a new release descriptor. The delivery pipeline service would first verify the commit signature before parsing the steering information.

Additionally, this concept nicely integrates with CI automation based on Jenkins or GitLab CI.
Although release descriptors and environment descriptors can be written manually by using any editor, the intention is to generate them automatically as far as possible. Especially writing release descriptors manually might become error prone if the number of artifacts gets large. Formatting the release descriptor file and writing it to the respective repository is automated by means of a generator script – so called rd tool – with input of the list of artifacts to be packaged. Since your CI automation is in possession of this information anyway, creating the release descriptor by a Jenkins or GitLab CI job is preferred by most teams.
Promoting a release to a specific environment means adding its reference to the respective environment descriptor. This is supported by rd tool as well and thus, enables automatic release propagation driven by a Jenkins or GitLab CI job.
Now a DevOps team has the freedom to easily decide which promotion steps they want to run automatically, and which steps they want to restrict to manual control.

3.4 Deployment Automation

We observe a huge variety of deployment and configuration automation solutions. This is due to the fact that a network operator makes use of network elements sourced from different suppliers, each of them having its own software technology strategy. Typical examples of technologies coming into play are

- Spinning up virtual machines by delivering VM images
- Installing application software on top of an operating system by means of RPMs
- Configuring software stacks by means of tools like Chef, Puppet, Ansible
- Configuring network elements by means of Netconf/YANG
- Managing VNF deployments via OpenStack Heat or TOSCA

- Managing container deployments via Kubernetes

The delivery pipeline should be able to support all of these options. Thus, enabling generic file formats such as zip, tar.gz and VM images is a minimum requirement. In addition, supporting frequently used repository formats such as RPM/yum and Docker registry API provides significant value to application DevOps teams.

Referring to container technology as an example and the release descriptor example described in section 3.1 above, the application container can be retrieved (after login with the proper credentials) in the production environment as follows:

```
docker pull acme-prod-docker.prod.registry.dt/acme/application:1.1.0
```

Often releases make use of multiple artifact formats being executed by different entities. As an example, application software might by delivered as RPMs to the application nodes directly, while configuration goes to an element manager or orchestrator as tar.gz. In any case, version dependencies across these places of execution need to be resolved.

Last but not least, it should be noted that the concept of environment descriptor can be easily extended to steer auto-deploy / auto-update capabilities of deployment automation functions.

3.5 Implementation

As outlined in the introduction, the main reason for the strict segregation between different network zones is due to security rules. In brief, it comes down to environments such as DEV, TEST and PROD shall not be connected, in order to apply sufficient control to any type of data ending up in production.

Our approach to deal with this is following:

We bridge the gap by establishing project / team specific delivery pipelines.

This is done by deploying a dedicated Application Staging Server which holds all software artifacts for a given project and mirror servers in management zones of the respective target environments.

The connectivity between the environments only exists in form a highly controlled and secured connectivity between the mirror servers and the staging servers.

Systems in the protected environments such a test and production don´t connect to systems in development environment or internet but instead retrieve the relevant artifacts from mirror servers in their respective management environment.

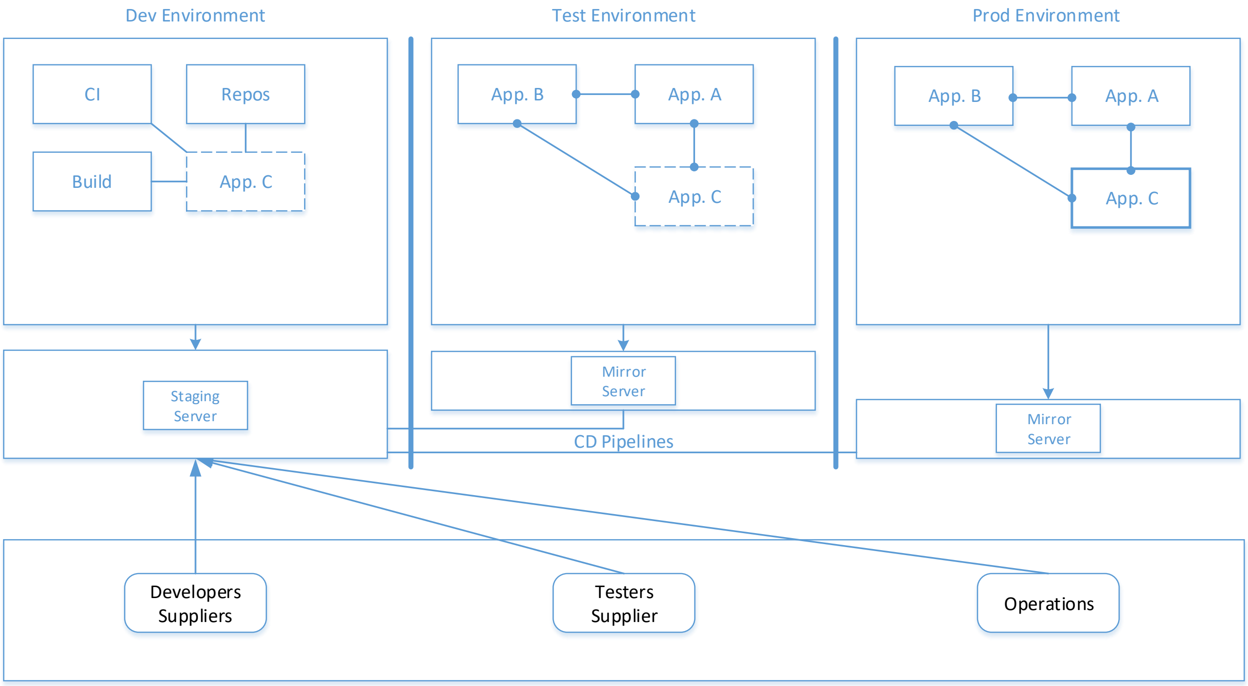
To reduce security risks to an absolute minimum several measures and principles are in place:

- Mirror servers and staging servers authenticate each other based on signed certificates
- Mirror servers only offer segregated repositories to a given project per environment.
- Access to the repositories on the mirror servers is based on digest authentication with username/password.
- Mirror Servers authenticate towards the project clients via certificates
- Connections are encrypted via SSL/TLS certificates
- All deployed elements including the web application firewalls use SELinux in “enforcing” mode.

This implementation proposal was agreed with security experts as an approach that allows agile software delivery teams to rollout change continuously (several times a day) in an automated way while maintaining a high level of security across the different network zones. Nevertheless, it is just one implementation approach for illustration and there may be others as well.

Solution architecture derived from above mentioned goals is depicted below. It is composed of the following elements:

- Staging server(s) implement the role of staging policy control. They are connected to upstream artifact repositories in order to pull artifacts from the artifact repositories (Artifactory in our case). Application DevOps teams can control their individual staging policies via steering repositories on GitLab – fully self-service.
- Mirrors are placed into the target network zones according to security and operational needs. This facilitates easy network access from deployment sites to the mirrors.
  Mirrors can offer multiple APIs such as simple web server, RPM repository and Docker registry.
- Network borders are protected by filtering functions, so called Web Application Firewalls (WAF). These only allow the specific paths to the repositories and log and deny any other activities.



(Web Application Firewalls are not depicted above)

The functionality of staging server outlined above can be implemented based on open source projects such as Pulp (https://pulpproject.org/). Pulp is a platform for managing repositories and does all the “heavy lifting” for our artifact repositories. The staging control is implemented as a service that monitors the steering repositories and manages repositories and artifacts accordingly using Pulp’s REST API.

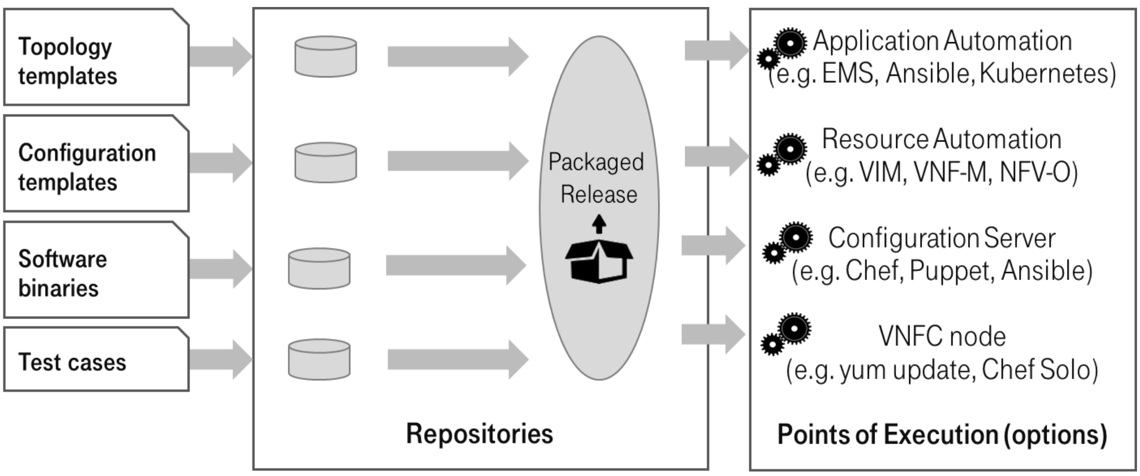
4 ADDITIONAL CONSIDERATIONS

4.1 Managing Dependencies across Technology Layers

Often releases are composed of artifacts coming from different sources and being executed by different entities. Typical scenarios are

- Application (VNF) is delivered by a supplier as world market product while at least part of configuration is managed by the operator.
- Application software is executed on service nodes while configuration packages might be executed by management nodes; there might be multiple different management nodes involved if cloud orchestration or service orchestration comes into play.

Respective dependencies need to be managed, see visual below.



Different versioning and packaging strategies can be applied to meet given organizational need. Teams practicing full DevOps may prefer to package software and configuration into a single release package. Advantage of this approach is that all dependencies can be resolved and tested by CI automation. Alternatively, teams facing a responsibility split between software delivered by a supplier's product unit in one release package and configuration delivered by an operator's integration team might prefer to keep delivery processes more decoupled by using independent versioning schema. As a consequence, they need to validate dependencies within delivery and deployment automation.

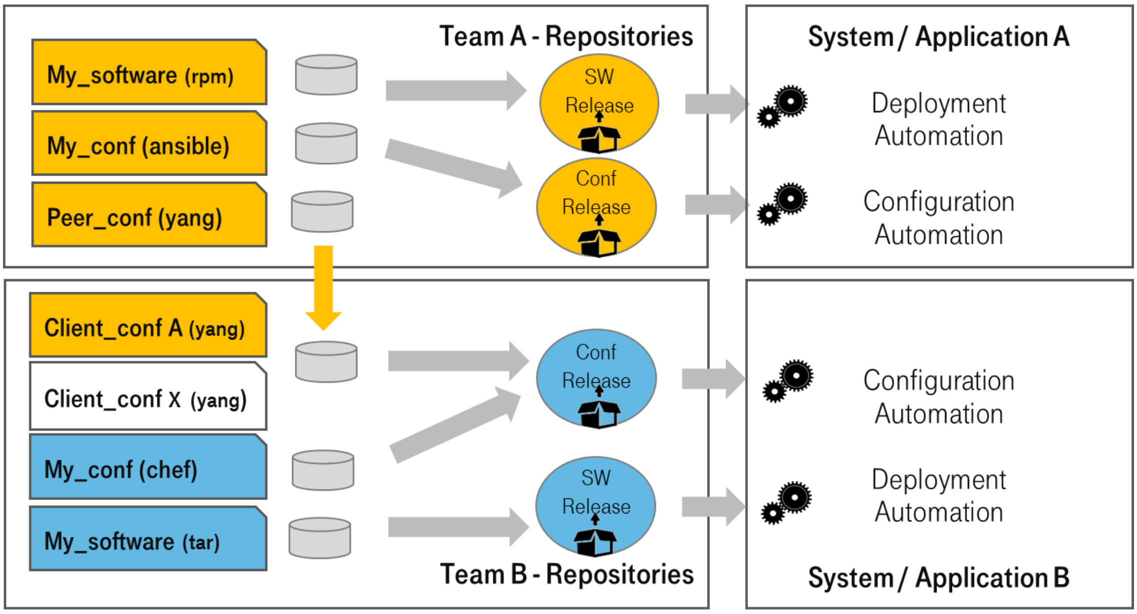
4.2 Managing Dependencies across Teams

When teams move to a DevOps way of working internally, cross-team collaboration needs to change as well, in order to not become an impediment. Traditional collaboration styles using ticketing systems create a lot of human workload on both sides, provider of a service as well as consumer of service.

An ideal target scenario is collaboration via API which allows to provide services by automation and to consume services by automation. Unfortunately, this is not always achievable with reasonable effort.

An interesting concept for streamlined cross-team collaboration is called GitOps as outlined by Thomas A. Limoncelli (see https://queue.acm.org/detail.cfm?id=3237207). It nicely plugs into our delivery pipelines and complements them as depicted below.

Example below depicts Limoncelli's use case where Team A cannot achieve their goal by changing their own application in an isolated way, but depend on a configuration change to be rolled out by Team B on their application too. Managing such team to team interaction via git pull request is the key idea of GitOps.



All in all, resilient dependency management across DevOps teams will involve two practices, 'forward' control via versioning as well as 'feedback' loop utilizing shared test cases, which again can be managed collaboratively via version control repositories.

4.3 Relation to other Automation Domains

As automation becomes ubiquitous in network operators' processes, it gets important to understand how CI/CD automation interferes with other automation areas, especially orchestration and ZSM.

Fundamentally, we distinguish between artifact lifecycle and instance lifecycle. Artifact lifecycle is subject to version control and managed via CI/CD automation. Network services and VNF instance lifecycle is the result of artifact execution by orchestration functions like NFV-Orchestrator or VNF Manager (as defined by ETSI MANO architecture), classic Element Management System (EMS), Kubernetes or so on. In order to efficiently interwork with CI/CD workflows, orchestration solutions need to clearly separate 'design-time' activities and artifacts from 'run-time' activities and states (terminology borrowed from ONAP architecture https://wiki.onap.org/display/DW/Architecture). This is essential for applying version control to all artifacts created at design-time and to execute run-time automation fully data-driven by repository content, not by GUI captured content directly. A good example for following this architecture paradigm is ONAP (https://www.onap.org/).

In contrast, some legacy orchestration solutions have been built GUI centric and tend to force the user to manually do same steps in a production environment as they have already been done (maybe with other data partially) on a test environment before. This introduces repeated and error-prone manual work. The safety of version control and acceleration potential of CI/CD automation would be not leveraged in such scenario.

5 CONCLUSIONS

On the journey to DevOps and applying CI/CD practices, network operators at some point in time hit a challenge we called the 'air gap'. It is caused by the fact, that production networks are strictly segregated from test networks and development environment networks for security and operational reasons. This 'air gap' between CI automation in one network zone and deployment automations in other network zones hinders seamless CD automation.
Finding a solution for this challenge requires to meet strict security requirements while bridging network boundaries.
The concept described above has been designed in close collaboration of security experts and CI/CD automation experts. Is enables smooth and seamless automation pipelines connecting software suppliers, internal DevOps teams and deployment environments such as test environments and production environments. The concept takes advantage of cloud infrastructures and cloudified applications, but it is not limited to them. It also works well with classic data centers providing virtual machines or even physical network elements.
Although primarily focusing on delivery pipelines per application (as a service to respective DevOps team), the concept provides capabilities having the potential to also improve cross-team collaboration and dependency management, which is a crucial area for network operator technology organizations on their way of DevOps transformation.
The implementation is based on open source to large extend. Community project Pulp is complemented by some own code implementing business logic for pipeline handling. This code is private for now but could be published under open source license if interest arises.

For further questions and feedback please contact the NGMN Office.