TELECOM INFRA PROJECT

Open Core Network Project Group

Continuous Delivery and Continuous Deployment Infrastructure

Technical Requirements v1.0

Authors:

Rabi Abdel, Vodafone

- abdel.rabi@vodafone.com

Boris Brenski, FreedomFi

- brenski@freedomfi.com

Nick Chase, Mirantis

- nchase@mirantis.com

Joshua Braeger, Facebook Connectivity

- jbraeg@fb.com

Contributors:

Sagiv Draznin, Rakuten Mobile

- sagiv.draznin@rakuten.com

Jonathan Bryce, OpenStack Foundation

- jonathan@openstack.org

Editor:

Chris Morton, Isn't That Write

- chris@isntthatwrite.com

Change Tracking

Date	Revision	Author(s)	Comment
May 28th, 2020	v0.1	Boris Renski	Initial table of contents
June 4th, 2020	V0.1a	Boris Renski, Rabi Abdel	Approved table of contents / Open development section
June 11th, 2020	V0.1b	Nick Chase	Introduction, CI/CD for telecom operators section
June 18, 2020	v0.1c	Boris Renski, Joshua Braeger	Tagging strategy
June 25th, 2020	v0.1d	Sagiv Draznin, Jonathan Bryce	Edits to CD section, Edits to CI section
July 9th, 2020	v1.0	Boris Renski/Stephani Fillmon	Edits for readability and draft approval
Sept. 20, 2020	v1.01	Chris Morton	Cleaned up formatting. Proofreading and line editing. Minor editorial revising. Checked conformance with (US) Plain Writing Act of 2010. Graphics manipulation and captioning.

Table of Contents

Table of Figures

Chapter 1 – Introduction

This document describes the Open Core Network (OCN) continuous integration and continuous deployment (CI/CD)—the development workflow and tooling used to build and deploy OCN microservices—starting from initial code commit all the way to deployment in a lab or proof of concept (POC) environment of an operator.

OCN CI/CD will cover tools and processes for the following three areas: 1) the actual development of OCN code, 2) testing in a staging environment, and 3) deployment in pilot environments within an operator infrastructure.

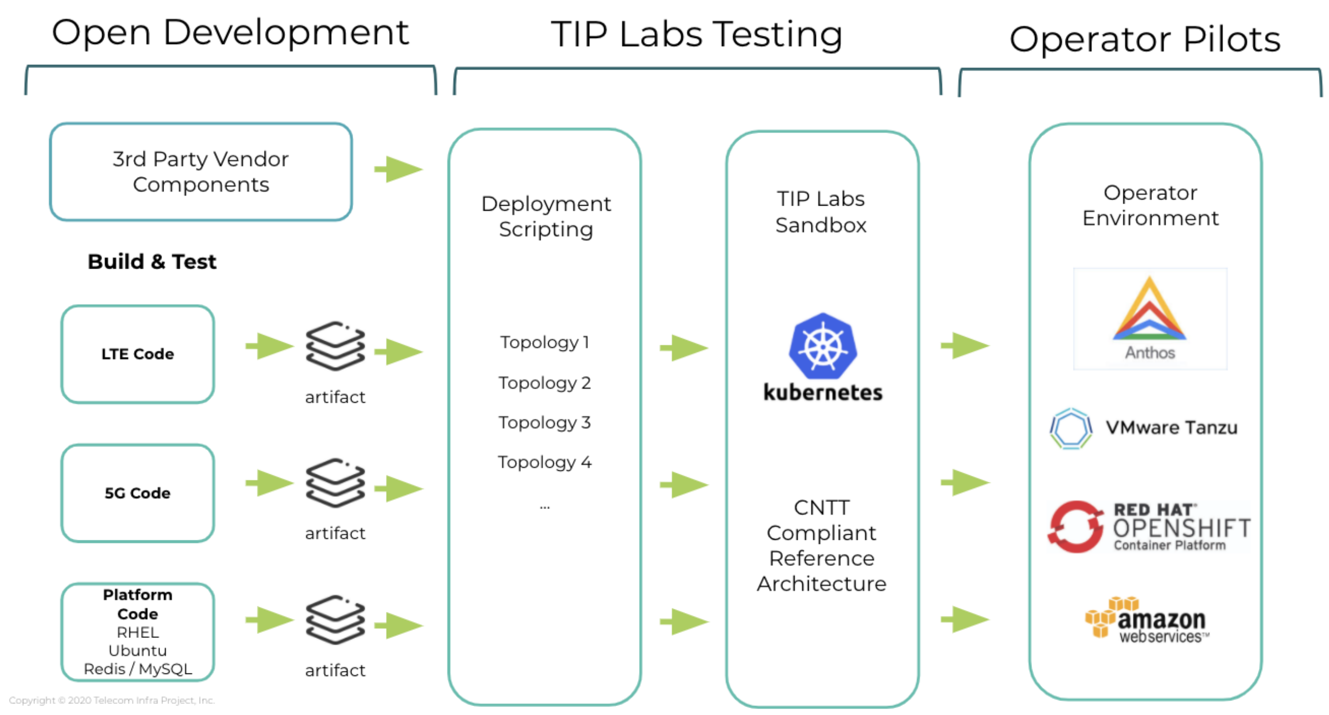


Figure 8 – OCN CI/CD

1.1 Goal

The goal of this document is to 1) make it possible for all OCN components to use the same CI/CD methodologies and tools to prevent conflicts when developing and testing individual pieces, and 2) make it as easy as possible for an operator to deploy POCs using the publicly available code and deployment scripts.

1.2 Document Scope

This document defines requirements for CI/CD workflow and tooling, including the following:

- Overall CI/CD workflow and selection of cloud-native tools for each process stage
- Summary of governance for contributing code to OCN and requirements for documentation/tutorials required for onboarding new development partners

- Description of the continuous integration process, including CI infrastructure, artifact tagging strategy, tools, and sample workflows
- Description of the continuous deployment process, including approach to integration with third-party CD systems used by telecom operators

The following items are out of scope:

- Cloud-native Kubernetes cluster architecture used as an NFVi for OCN microservices. (This is described in a separate document)
- Languages, libraries, and development IDEs used to build OCN microservices. While it's advisable that parties to the OCN development process try to avoid creating an unnecessary "zoo of development stacks," it's largely left up to individual developers to pick the right development tool
- Software and toolkits for post-deployment functional testing. OCN will be using a number of tools for traffic generation and radio interface emulation, but expect that operators may have their own tooling preference.

1.3 Requirements Approach

The approach to determining requirements for this framework is to look at systems and processes in use by projects both in and adjacent to OCN. In addition, it examines best practices currently in place with various operators to ultimately find a common ground that provides the most value with the smallest learning curve.

As such, we'll attempt to go with the simplest system possible, and revisit it if features and/or capabilities turn out to be missing.

1.4 Document Overview

This document includes the following chapters:

- **Overview of CI/CD for Telecom Operators** – Briefly explains how CI/CD works, why it's necessary, and some of the unique aspects of its use in the telecom environment
- **Description of Workflow & Tool Choices** – Includes a high-level description of the CI/CD workflow, from code commit to deployment in a staging lab, including the various tools used at each stage
- **Open Development Process** – Provides the basic repository structure, as well as lays out documentation that will ultimately be needed and basic governance requirements
- **Continuous Integration and Testing** – Explains the process by which code is built and tested, looks at various platform seed components and workflows, and examines a tagging strategy for the software.
- **Continuous Deployment** – In addition to explaining the continuous deployment process, we look at its implications for the business processes of a telecom operator.

Chapter 2 – CI/CD for Telecom Operators

Continuous integration/continuous delivery (CI/CD) is a means for creating software that ensures that at any point in the development process, the software is always deployable and always reliable.

Continuous integration involves ensuring that all developers use a common source control manager (SCM) repository and commit their changes as often as possible. When code is committed, the SCM kicks off a series of automated tests to ensure the code is functional and hasn't caused errors in any of the existing codebase. If this build process fails, developers are expected to stop what they're doing and find and fix the errors, thereby ensuring that the codebase is always in a functional state. Because the changeset is typically small, errors are easier to find.

Continuous delivery is an automated process that takes the code created during the CI phase and deploys it to multiple environments where it’s needed. CD also performs testing on these environments to ensure the software is working as expected. In most cases, delivery is initially performed on a staging environment to prevent any errors from affecting the production environment. If errors are detected, the deployment is aborted and the changes reverted. Once the software is deployed to production, it’s tested again and—in the unlikely event that problems are detected—changes are immediately rolled back.

Historically, telecom infrastructure was procured from incumbent network equipment manufacturers (NEPs) that pre-integrated hardware and software. This meant that introducing upgrades and improvement to the network often required swapping old hardware for new—making network upgrades slow and expensive.

With 5G we see a shift in network architecture toward disaggregation between software and hardware. The biggest advantage of a 5G, software-centric network is its ability to do away with lift-and-shift upgrades. Instead the network evolves by upgrading software components though the introduction of small, regular changes—much like hyperscale Web 2.0 companies evolve their systems. Adopting this approach requires implementing CI/CD practices described above.

OCN is being built with CI/CD functionality, in that its cloud-native, software-based, and hardware agonistic 5G core is intended to be frequently updated in small increments using CI/CD tooling. Public GitHub repositories containing OCN code must also contain the tooling to enable operators to deploy OCN code. As new versions of OCN microservices come out, operators will be able to lean on the same repositories to continuously and gradually evolve their networks.

CI/CD works best when as much work and testing as possible is “shifted left”—that is, performed as early in the development process as possible. By creating confirmations and testing them at the development phase (as opposed to during deployment), bugs are easier to find and fix with little-to-no impact on users.

Chapter 3: CI/CD Workflow and Tool Choices

The overall CI/CD workflow consists of several tools used over multiple stages of the process. The CI/CD tool ecosystem is very broad and there is no one-size-fits-all approach. Therefore, one of the most common problems with CI/CD is a lack of consistency between development teams armed with disparate tools and workflows used for building and testing components of the same system.

The goal of this chapter isn't to create "the best" workflow with "the best tooling," but rather to define a common approach for OCN development parties to adhere to. This is to make it easier for a new contributor to get involved, as well as for an operator to deploy code by adopting a similar workflow and tools in their environment.

We logically split the entire flow from initial code commit to deployment into three general stages, each with a set of substages:

- **Open Development** – Practices, tools, and workflow for contributing new feature suggestions and code fixes to OCN
- **Continuous Integration & Testing** – Workflow and tooling for building and testing code, including interop between various OCN components
- **Continuous Deployment** – Workflow for deploying OCN components into lab or operator staging environments in a number of use case-specific topologies—including deployment strategies, deployment scripting, and post-deployment functional testing

Stage	Description
Open Development	
0. Feature Request/ Bug Report	A vendor or operator who found a bug or has a useful idea for a new feature for a OCN component files a detailed description about it in the public code repository
1. Code Commit	A developer who wrote a bug fix or a feature commits code to a public code repository
1.1 Automated Gating	As the number of code commits increase, over time an optional step of automated commit-gating runs a series of code tests and provides feedback to the committer. Only code that meets certain minimum quality standards contends for coder review cycles.
2. Code Review	Designated core reviewers having review rights for a given repository perform the code review and vote whether to accept it and initiate the automated CI jobs.

Continuous Integration & Testing	
3.1 Security Scanning	The CI/CD pipeline is triggered at this point. As an initial step, code scanning tools (security, static code analysis, code coverage) perform automated scans. Based on outcomes, the code build/merge stage is triggered, or the process is abandoned and possible issues/vulnerabilities are reported.
3.2 Static Code Analysis	
3.3. Code Coverage Scanning	
4. Build & Test	The OCN component with the newly committed code is built by using an appropriate build tool; the CI/CD engine kicks off a series of integration tests.
5. Artifact Version Control	1) A binary, VM image, or container is built by using container/image build tools (depending on component), 2) appropriately tagged, then 3) stored in an artifact repository with corresponding metadata.
Continuous Deployment	
6. Trigger Deployment to Staging	An authorized party manually triggers auto-deployment of the recently updated artifact into a staging environment, using a particular deployment strategy (blue/green or canary)
7.1 Execute Deployment Strategy	The CD engine executes deployment scripts to update the staging environment
7.1 Initiate Functional/ Acceptance Tests	The CD engine triggers automated acceptance/functional tests against the staging environment while monitoring key performance metrics (e.g., uptime, throughput)
7.3 Auto-Rollback on Fail	Optional: This step is automatically triggered by the CD engine when one or more acceptance thresholds (defined in the monitoring system) are broken post-deployment of a new component (for example, eNB fails to connect to MME or UEs unable to reach eNB)

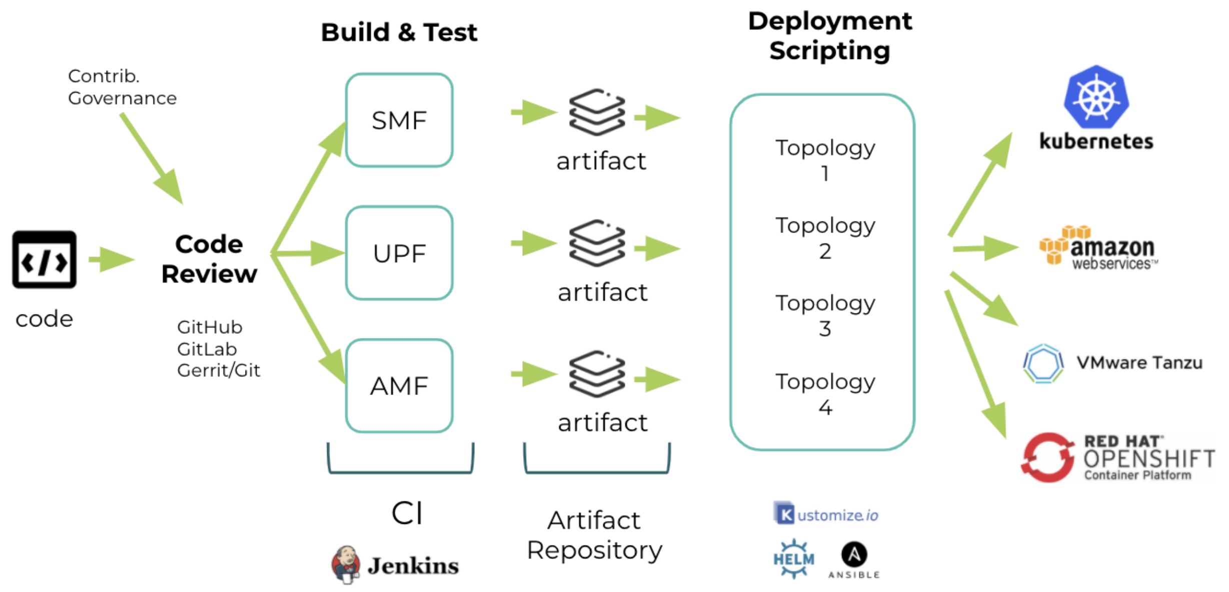


Figure 9 – CI/CD methodology

To facilitate a transparent and vendor-neutral development process, we've evaluated a variety of tool options and have outlined our decisions and selection rationale in subsequent subchapters. We adhered to the following principles in making the selections:

- **Kubernetes-friendly** – Because OCN aims to adhere to cloud-native principles, the tool should be well integrated with K8s and compatible with the ecosystem of related cloud-native tools
- **Open source** – Any operator or vendor should be able to replicate the CI/CD toolchain in their environment without introducing a vendor bias or incurring licensing costs. Therefore, we aimed to use mainly open source tools unless a given tool introduced adoption barriers or too much complexity
- **Healthy community** – For each open source tool, we looked for a healthy and diverse community of maintainers and good forward momentum behind the project (e.g., number of GitHub stars, contributors, frequency of contributions) such that it's unlikely to disappear a few months after we chose it

3.1 Issue/Feature Tracking Tools

We aim to facilitate a transparent development and contribution process for OCN. That process usually starts with a new user or vendor reporting a bug or suggesting a feature. It's therefore important that issue tracking is straightforward. In making the right choice, we looked at common tools across other open source communities such as CNCF, OpenStack Foundation, and the Open Networking Foundation. We reviewed the following options:

- GitHub Issues
- Jira
- Launchpad

We chose GitHub Issues because it's familiar to many open source developers. Using GitHub Issues also streamlines the contribution process, as we also use GitHub to store the code repository and for code review. Furthermore, many contributors' internal processes already use GitHub Enterprise, so using GitHub for OCN makes it easier for operators to extend our tooling into their environments.

3.2 Code Repository Tools

The code hosting infrastructure contains the source code manager (SCM). Because Git is currently the most popular SCM type, we'll use a Git-based SCM. Options we considered include:

- GitHub
- Gitlab
- Bitbucket
- A privately-hosted Git repository

We opted for GitHub for several reasons. In addition to being a ready-made system that provides all infrastructure and operational support for free (OCN is open source), it also provides a number of other integrated tools to satisfy other requirements. Plus the majority of developers have at least a passing familiarity with it.

3.3 Code Review Tools

Part of the strength of open source development is that every change can be viewed by multiple developers, and in most projects (including OCN) it must be reviewed by multiple developers (including code owners). This workflow requires a system for code review, which:

- provides reviewers a place to see pending changes
- enables them to view and download code for testing
- make comments/request changes before indicating whether the change should be merged

Options we considered include:

- GitHub Pull Request
- Gerrit

We chose GitHub Pull Requests because it doesn't require additional software, it's already integrated with the Git repository, has a broad integrated development environment (IDE), and most developers are familiar with using it.

3.4 Security Scanning and Code Analysis Tools

It's natural that in a community-driven, collaborative development, some of the code submitted by developers might not have proper test coverage. It might have dependencies on libraries with known security vulnerabilities. Or there could be other problems that might not always be possible to catch through the peer review process.

To minimize the chance of poor code getting into the upstream codebase, a code scan should be performed by using a static code analyzer, or code vulnerability scanner, prior to kicking off the code build process and integration tests. Some code scanning tools include GuardRails, SonarQube, and JaCoCo. For OCN CI/CD we avoid making a prescriptive recommendation for such a tool because they're development language- and IDE-specific. In general, we plan to add the necessary tools and CI triggers for automated code scanning as the contribution base grows, deferring the actual tool selection to the development community down the line.

3.5 CI Workflow Engine and Build Tools

3.5.1 Incorporating Seed Code and Net New Projects

Running CI requires a system that can be configured to define build and test processes, running them either on command or when a triggered event occurs. Jenkins is the most commonly used tool for that purpose. We expect at least some of the seed code coming into OCN (such as converged-MME developed in collaboration between OpenAirInterface™ and Facebook Connectivity) to come with existing workflows and test scripts built around Jenkins and CircleCI.

Moreover, converged-MME might not be exclusive to initial seed code. It's possible that another open source or vendor project will eventually be contributed to the OCN public code repository, coming with its own CI workflows and scripts. Therefore, the approach that OCN CI/CD suggests is to start with existing workflows and scripts and move over to a standard CI workflow engine within 12 months.

3.5.2 Long-Term CI Standard

For a longer-term CI workflow engine, we analyzed a variety of cloud-native CI/CD systems, including Jenkins, Tekton and Argo CI, and have decided to standardize on Argo CI because it best meets selection criteria we outline for cloud-native tools. Argo CI is capable of doing both CI and CD (through Argo CD), which permits us to minimize tool diversity. With Argo CI, we can cover the entire CI/CD workflow with a single, cloud-native tool that is lightweight and supported by an active community.

3.5.3 Build Tools

We don't offer an opinionated prescription for code build tools, primarily because they're specific to the platform you're building against and the programming language. We expect there is no way to avoid build tool diversity and leave the choice up to the development community.

3.5.4 Distributed CI Engines

As the OCN community grows and frequency of contribution increases, running a single instance of a CI server might not be sufficient. A number of projects such as Zuul (from the OpenStack Foundation) and Prow (in CNCF) facilitate efficient CI for collaborative development in communities made up of thousands of developers triggering hundreds of CI jobs every day. For initial OCN implementation, we expect to run under 30 CI jobs per day, which doesn't yet warrant implementing a distributed CI system such as Prow. But as the community grows, we might revisit this.

3.6 Artifact Repository and Image Build Tools

To speed up and simplify the development process and assure consistency of deployments, various OCN components will be built and stored as either immutable VM images or containers after integration testing. Building and storing images requires an image build tool and an artifact repository. Packer is the most popular image build tool, capable of building both VM and container images ; it's what we intend to use for OCN development.

A number of artifact repositories exist. We considered Nexus, Harbor, and Artifactory. Nexus is the oldest and arguably the least cloud-native, while Harbor is on the other extreme—it's new, open source, and is close to the K8s community. We decided on Artifactory primarily because it's most commonly used in the cloud-native context today, supports a variety of artifact types, and is likely to be familiar to teams looking to start collaborating as part of the OCN community.

3.7 CD Tools

For CD we looked at several cloud-native tools, including Tekton, Spinnaker, Argo CD, and Jenkins X. An important consideration in choosing a continuous delivery tool is its ability to support declarative continuous delivery. This is where a Git repository can be used as a single source of truth for defining and managing all parameters of the desired target stage of the application and deployment environment. Given our previously outlined tool criteria, we chose Argo CD because it meets that criteria, unlike some of the other tools we considered.

3.8 Functional & Acceptance Testing Tools

Upon deployment of the newly built artifacts into the OCN staging environment, the CD system triggers functional and acceptance tests on the staging system to determine if the new updates are production ready. The tests can cover the entire system or just a few specific components, such as the UPF function.

Similar to static code analyzers and code build tools, we expect the system will use a variety of functional test tools. For OCN seed code components, at a minimum S1APTester will initially be used to emulate the eNodeB interface and DS Tester. Over time, additional tools might be added to perform more thorough automated E2E acceptance testing, with open source Robot Framework (commonly used in other operator environments) being a top candidate.

3.9 Monitoring Tools & Rollback Triggers

Given the frequent pace of change rollouts typical in cloud-native environments, some problematic code will invariably make it into production. Being able to perform an automated system state rollback to the previous, well-functioning known state is essential if certain malfunctions are detected during acceptance testing. Because we aim for staging environments to emulate production as closely as possible, similar monitoring thresholds and rollback mechanisms should also be applied in staging.

For implementing deployment strategies into staging with rollback occurring on monitoring triggers, OCN will use Argo Rollouts CRD (a Kubernetes custom resource definition) and Prometheus—Argo, because we're already using it for managing workflow and rollouts enabled to execute various deployment strategies; and Prometheus, because it's the most common, open source, cloud-native monitoring tool already used within some OCN seed code.

Chapter 4: Open Development Process

OCN's success depends on building a diverse group of collaborators to build a converged core network. That means recruiting third-party participation is crucial, so we must have clear and straightforward means for anyone to get involved. We'll capture some of the key aspects of this requirement in the following sections.

4.1 GitHub Repo Structure

Per section 3.2 of this document, OCN software development will use GitHub in general, and the magma public repository in particular, for development of various OCN software components. To facilitate the process and make it easy for all parties to onboard and deploy code, it's essential that the GitHub directory structure is well documented and transparent to third parties.

Initial OCN seed code will come from the Magma project The initial Git repo will inherit original structure from https://GitHub.com/facebookincubator/magma. But near term this repo structure will evolve to the following and reside at https://GitHub.com/magma

- magma
    - cn // Core Network
        - include // headers visible across all tasks (4G) and functions (5G)
        - lib
            - 3gpp // headers for 3gpp specs of 4G and 5G
            - bstr // string handling
            - common // logging, Redis and other utils
            - itti // InTer Task Interface for sending messages between tasks
            - openflow
        - protos // protobuf definitions for GRPC services and Redis state
        - submodules // reference to external repos, such as FreeDiameter, etc
        - ran
            - sctpd
            - enodebd // may be renamed to rand or ranconfigd ??
        - access
            - mme_app task
            - nas4G task
            - s1ap task
            - s6a task (including s6a_proxy)
            - sctp task
            - sgs task
            - grpc_service task
            - (ngap)

```
                -   (amf)
                -   (nas5G)
        -   session
                -   session_manager
                -   sgw task        // Combines SGW and PGW control plane
                -   mobilityd
        -   policy
                -   policydb
        -   user plane
                -   (upf)
                -   pipelined
        -   subscriber
                -   subscriberdb
                -   oai_hss
        -   federation
                -   feg
        -   analytics
                -   service303
        -   tests
                -   unit_tests
                -   integ_tests     // tests for s1ap-tester and TTCN
-   nms
-   orc8r
        -   cloud
        -   gateway
                -   Configs
        -   protos // protobuf definitions specific to cloud services, e.g. metrics
-   deploy
        -   dev     // Ansible tasks and Docker containers for development
                -   fwa
                -   cwf
        -   release // scripts for packaging, release, deployment, etc.
                -   fwa
                -   cwf
-   ci                  // tools, scripts for CI pipelines
-   third-party
-   docs
-   CODE_OF_CONDUCT.md
-   CONTRIBUTING.md
-   LICENSE
-   README.md
```

It’s expected that this structure will change as new software components are contributed by the community, but any changes to the repository should follow the peer review process. As part of that, it’s important that newly introduced branches and directories remain well documented. Some directories that are especially important to understand include:

- **magma/cn –** This is the base directory for the core network; anything not directly related should be outside this directory
- **magma/cn/libs –** Code used across multiple core functions, e.g., 3GPP-specific data structures, logging, custom string library, Redis client
- **magma/cn/submodules –** Third-party modules that should be present on the system
- **magma/cn/access –** Core component that manages user registration with the core, e.g., NAS, MME
- **magma/cn/session** – Core session creation and management components
- **magma/cn/userplane –** Core data path control components
- **magma/cn/subscriber –** Core components to authorize subscribers in a network
- **magma/cn/nms –** Network management system
- **magma/cn/orc8r –** Orchestration logic
- **magma/ci –** Scripts for CI pipelines
- **magma/third-party –** Custom patches for third-party software, e.g., OVS, LibFluid
- **magma/docs –** Documentation for how to develop, deploy, and use OCN.

Regarding documentation, it must remain separate from the remainder of the code so a public representation (such as a website) can be automatically generated.

4.2 Software Contribution Documentation Requirements

OCN software development will follow a meritocratic and transparent governance process similar to that of other open source projects (e.g., CNCF and OpenStack), in which anyone can contribute. However, those contributions must be reviewed by trusted individuals before being included in the actual codebase.

We need to ensure that the following documents exist to facilitate this process:

Contribution guidelines – Describes the process all contributors must follow to contribute code. This includes any registration, authorization, and certification (e.g., Developer Certificate of Origin), as well as instructions for configuring and accessing any code repositories and CI systems. It must also describe any process requirements, e.g., submitting a proposed feature t o a governing committee, if applicable.

Coding guidelines – Provides enough information for a programmer to find what they need and code it according to community guidelines. This might include software architecture maps that explain where to find various functionality, plus style guides and test coverage requirements.

Review guidelines – Describes what to look for in contributed code, and how to provide such reviews. It should also include information for contributors who would like to become maintainers.

Testing guidelines – Describes how to initiate and access the build-test phase of the CI/CD process, as well as community expectations for the level of testing for unit testing, functional testing, integration testing, and so on. It should also describe expectations for "working" code, along with processes that must be followed should the code base fall below the expected level.

4.3 Software Contribution Governance

By definition the creation of open source software is a collaborative process. It involves multiple persons having differing levels of access and accountability.

4.3.1 Contributors

Anyone can become a Contributor by submitting code to the project and having it accepted through the review process. A Contributor has had code accepted, or merged, within the last 12 months. They have access to propose and review code, but the code must be merged into the OCN repository by Maintainers. Contributors are also eligible to vote in the Technical Oversight Committee elections.

4.3.2 Maintainers

A Maintainer has the ability to merge code into the project. Maintainers are active Contributors and project participants. To become a Maintainer, a Contributor must be nominated by an existing Maintainer and approved by a simple majority of established Maintainers (with no objections). Within the project, Maintainers of subcomponents may decide to have additional requirements for the review and acceptance of code in their repos.

4.3.3 Code Owners

A Code Owner is typically responsible for a particular area in the repo and must approve any contributions to it before they can be merged.

4.3.4 Technical Oversight Committee

The Technical Oversight Committee (TOC) is comprised of five members responsible for architectural decisions and making final decisions if Maintainers cannot come to an agreement. The initial OCN members will be appointed to 12-month terms with a goal of moving to an elected TOC membership within 24 months. Once elections begin, they will be staggered so that only a portion of the TOC seats becomes open during each election cycle.

There are no term limits, but no more than two of the five seats can be filled by any one organization so as to encourage diversity. The TOC will meet regularly in an open forum with times and locations to be published in community channels.

The exact size and model for the TOC may evolve over time based on the needs and growth of the project, but the governing body will always be committed to openness, diversity, and the principle that technical decisions are made by technical Contributors.

Any person may hold multiple roles simultaneously, e.g., Maintainer + TOC member.

Chapter 5 – Continuous Integration and Testing

The CI process automatically begins after code is approved by the Maintainer of a OCN component and is merged. CI amounts to the workflow engine (in our case Jenkins and Argo) taking newly built code through a series of steps (outlined in section 3); it ensures that the newly updated component functions within the overall system free of any problems. Test series are specific to the component, are programmatically defined, and are commonly referred to as a *CI pipeline*. Each step in the pipeline is a *CI job*. CI process output is an artifact—a container, a VM image, or a binary—that can be deployed using a deployment script into either a staging or production environment.

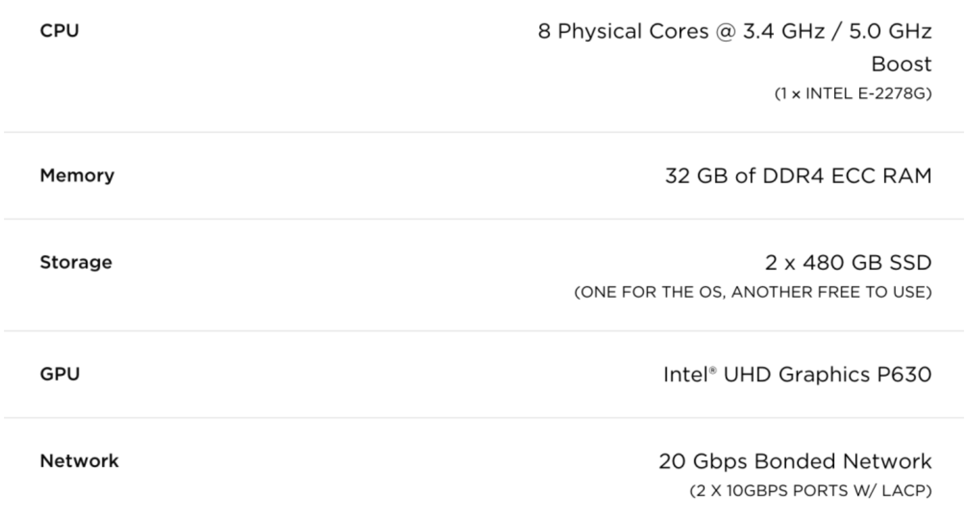
5.1 CI Infrastructure Resource Pools

Running CI jobs requires compute resources—either physical servers or VMs—that can execute CI tasks; these include building a binary from source code, building a VM or container, or sending traffic to a VM and logging the behavior. Such compute resources can come from a variety of infrastructure pools. They can be physical servers by TIP Labs, physical servers provided by one of the OCN partner organizations (e.g., OpenAirInterface Software Alliance or OpenStack Foundation), or virtualized resources in the public cloud.

Most CI jobs don't permanently consume resources. Instead, the resources are temporarily provisioned during any given CI job. This makes it possible to use infrastructure pools having low uptime SLA (e.g., AWS EC2 Spot Instances or servers with limited availability), as long as the logic for dynamically selecting the appropriate CI infrastructure pool is coded into the CI pipeline.

During initial OCN development stages, we expect that fewer than 30 CI jobs will run per day. Because of that, initial CI resource requirements aren't very high. We plan to use the following two infrastructure pools to run OCN CI jobs triggered in the public GitHub repository:

For the first OCN seed code that will go into the public GitHub repository as a CI Job, we'll start with a single, bare metal instance having the following specifications.



CPU	8 Physical Cores @ 3.4 GHz / 5.0 GHz Boost (1 x INTEL E-2278G)
Memory	32 GB of DDR4 ECC RAM
Storage	2 x 480 GB SSD (ONE FOR THE OS, ANOTHER FREE TO USE)
GPU	Intel® UHD Graphics P630
Network	20 Gbps Bonded Network (2 X 10GBPS PORTS W/ LACP)

Figure 10 – Initial OCN bare metal compute resource

It'll use a TIP Labs K8s cluster, the architecture of which is described in the OCN cloud-native infrastructure requirements.

As the number of components and contributing developers grows, OCN will likely add additional CI infrastructure resource pools, provided by third-party partners, to the mix. We'll also look into adding logic for dynamic infrastructure resource selection into CI, in addition to engines for executing parallel CI jobs having complex dependency logic (e.g., Prow, Zuul).

5.2 Platform components

Described in the previous section, CI process output will be a number of deployable artifacts such as containers or VMs. To make it faster and easier to build new versions and deploy them more predictably, components are generally split into platform level and application level.

For example, OCN microservices often require several components that can be built and maintained separately from application code, such as operating systems with particular kernel modules, databases of specific versions, or application servers. We refer to those artifacts as *platform components*. Separating platform- and application-level components makes it easier to run CI jobs; instead of building a new kernel module with databases and other components from scratch, one can take an existing, recently built image and only run a build job for the component that was changed.

This approach also helps make it simpler to maintain compatibility with various platforms. For instance, UPF functions, or Magma access gateway (AGW), can run on x86, Arm, and MIPs. Having prebuilt kernel images for these three platforms, with all required dependencies baked in, simplifies testing code changes to UPF or AGW against three disparate hardware platforms.

For OCN, we intend to use Packer to build all container and VM images, including those for platform components. Over time we might expand the list of separate platform components that will be maintained as part of the CI process; but for now we expect to start with the following that are required for CI development:

- x86 Kernel for UPF/AGW: Debian 4.9 with OVS 2.8 and Python 3.7
- DB for Magma Orchestrator: Postgres 9.6.11 + Helm charts for HA Deployment

5.3 Incorporating Third-Party Vendor Components

A key advantage of the open source approach to software development is the ability for the ecosystem of vendors having complimentary solutions to join and integrate with the project in a frictionless manner. A service provider core network always interfaces with a large number of third-party systems, including components such as:

- RAN
- Network management systems
- Network orchestration systems
- Subscriber management systems
- Billing & charging
- Third-party core networks

OCN CI/CD infrastructure must be designed such that it's easy for third-party vendors to add CI tests specific to their systems. All of CI/CD tooling and specific OCN tests will be developed in the open, and hosted in an open source GitHub repository (refer to section 4.1 for GitHub repository structure specifics).

Third-party vendors may propose to add non-gating integration tests into the existing CI workflow for OCN, following the same process as adding a new bug report or feature. Detailed documentation for adding third-party, non-gating CI tests into the OCN release process should be provided (along with other documentation and guides described in section 4.2).

It's already required for some OCN seed code components coming from Magma (specifically Magma Access Gateway) to test compatibility with the SPGateway version being hosted in the OpenAirInterface GitLab code repository and maintained independently from remaining OCN components. Introducing non-gating integration tests for Magma AGW to test against OAI SPGateway can be a first example of third-party CI hook implementation within the OCN development process.

5.4 Tagging Strategy

Upon completion of a successful CI run, the newly updated software artifact (platform or application level) is stored in an artifact repository (we've selected JFrog Artifactory, per section 3). For stored artifacts to be easily consumable for subsequent deployments, it's essential that all are supplemented with relevant tags indicating when each artifact was built, as well as how well-tested and ready for deployment it is. To that effect we'll apply a consistent tagging strategy to all artifacts, with them generally being split into two categories:

- **CI Artifact** – A VM, image, container, or binary that has successfully completed a CI run, but isn't a release candidate. For example, one could be an intermediary build that hasn't been sufficiently tested to be deployed as part of the complete system.
- **Release Artifact** – A VM, image, or container that can be used as part of a deployment, either to staging or production environments.

All artifacts will have CI Tags applied to them, the format of which will be: **ci-<commit_hash>**. Commit hash is an auto-generated Git object used to easily derive essential information about the commit, such as date, time, author, previous commit, and so on.

Release tags will use semantic versioning for all prereleases and releases. For example:

- 1.0.0-rc1 for release candidates
- 1.0.0 for major version releases
- 1.0.1 for minor version releases

These versions will be used for Git release branches, docker images, and binary releases, if available. A detailed description of semantic versioning methodology is described here. In short, the approach adheres to the following set of principles:

Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes
2. MINOR version when you add functionality in a backward-compatible manner
3. PATCH version when you make backward-compatible bug fixes

Additional labels for prerelease and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

5.5 Seed Components & Workflows

To bring together the concepts described in previous chapters, we'll use an example of running a CI process for one of the OCN seed code components – converged-mme.

1. New code is committed to a component of the converged MME repo.
2. Code is reviewed by a number of contributors from various organizations, such as OpenAirInterface and Facebook Connectivity. The pull request must be approved by at least two reviewers.
3. The converged-mme repository maintainer (equipped with Write access as a minimum) accepts the code and merges the pull request; this automatically engages the Jenkins pipeline.
4. Jenkins (later to be replaced by Argo) kicks off CI jobs to build and provision six VMs on a Packet.net bare metal server (described in *Section 5.1 CI Infrastructure Resource Pools*). Three are test VMs and other three contain various converged-mme builds, representing disparate deployment configurations (see the following diagram)
    a. VM Image with Converged MME and Magma Service (mobilityD and sessionD)
    b. VM Image with Converged MME, OAI-HSS and GTP SPGW
    c. VM Image with Converged MME, OAI-HSS and CUPS SPGW
    d. VM with S1AP Tester software, which emulates an eNB S1 interface
    e. VM with DS Tester Software
    f. VM that generates artificial networking traffic
5. Upon completion of a successful run, the final CI stage tags the artifacts (in accordance with the tagging strategy laid out in section 5.3) and saves the images to the Artifactory.

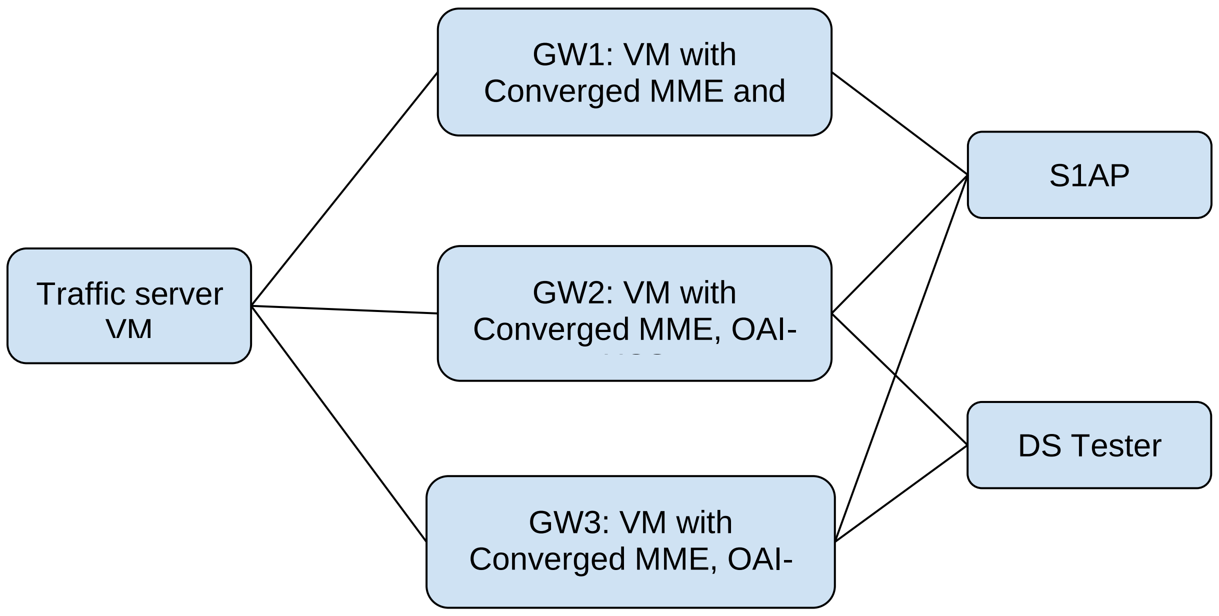


Figure 11 – Running a CI job for converged-mme

Chapter 6: Continuous Deployment

Continuous deployment is the final process stage, when by manual trigger the CD delivery workflow engine initiates a series of jobs; they combine the artifact tagged for release with deployments scripts (mostly Helm charts) to deploy the complete OCN or just those components that have changed into either a staging or an operator environment.

The goal of the CD stage is to automate the deployment process, making it possible for newly built, release-ready artifacts to be immediately tested in real life scenarios and quickly promoted to production. Automating deployment also provides immediate feedback to the development process, uncovering issues with the system that might have been missed by unit and functional testing during the CI process.

6.1 Integration with Business Process

Unlike the CI stage that is meant to test standalone system components purely as part of the software development process, CD frequently extends into staging and production environments. It requires much deeper interconnection with the business process of the entity that owns the production environment.

If a given operator/vendor is running an OCN production component, its DevOps group can't simply decide to update their environment and automatically push new artifacts into production. Prior to moving artifacts from the repository into the staging or production realm, a series of approvals and validation steps must occur. For example, artifacts might need to be scanned for compliance and receive a security department sign-off. Or they might need to go through a series of additional test steps performed by an end user operations team. So unlike CI, where code progress from initial commit to validated artifact is mainly done by engineers and DevOps, progressing through CD stages can require many cross-departmental approvals.

A separate tool that coordinates deployment workflow from the business process standpoint might need to be overlaid on top of lower level tooling responsible for pushing the artifacts. Generally, such a tool needs to be able to express the deployment workflow using business process model and notation (BPMN) and be easy to use for non-technical users. Examples include Viewflow, Camunda BPM, and Zeebe.

For MVP implementation of CI/CD infrastructure, OCN won't implement a specific integration with BPMN management framework. Instead, the approach will be to integrate with the same tool as third-party operators as we move OCN to operator trials. we'll then consider adding BPMN as part of OCN CI/CD itself at a later stage.

6.2 Dealing with Diversity in CD Tooling

There is no such thing as a universal CD pipeline that could work for all use cases; its final stages will always be specific to the end user environment. Moreover, the CD tool ecosystem is vast. The closer you get to the deployment phase, the more the pipeline becomes intertwined with the end-user business process and, consequently, the more diversity in tools and workflows you encounter.

To that effect, MVP of OCN CI/CD should not be designed to replace the existing operator CD process, but rather integrate with it in a flexible way. For instance, an existing operator CD system might require that our CI/CD process deposit artifacts with a particular set of metadata that complies with their format—and to a designated location such as its AWS S3 account.

6.3 Deployment Topologies for Staging Environments

OCN consists of many microservers that can be deployed in various configurations and topologies depending on the use case. The initial, primary OCN use case is fixed wireless access. To ensure that we're able to efficiently test OCN for fixed wireless access (FWA), it's important that we do so against topologies specific to that use case. The two common, expected topologies for FWA are:

- **Centralized** – In this topology all OCN components (control and data plane) run on a single K8s cluster in one physical location. This topology might be commonly used for early trial stages or smaller scale deployments. Deploying and testing in this configuration forces us through a healthy exercise of testing a E2E K8s-driven deployment, where all OCN components are running on Kubernetes rather than bare metal.

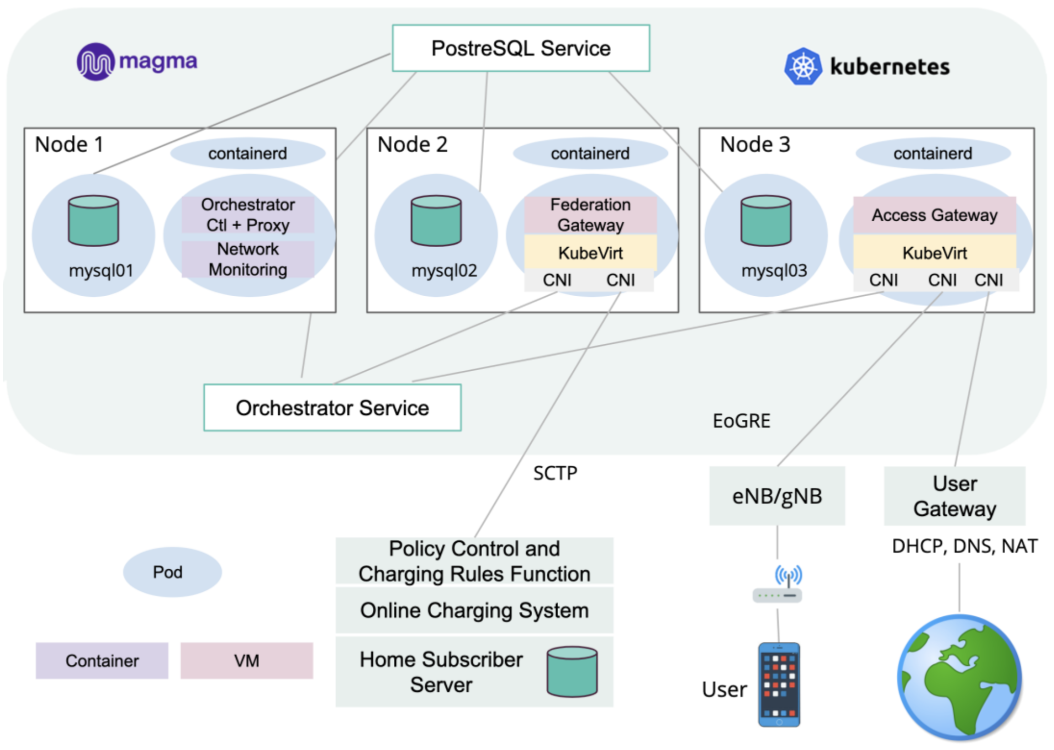


Figure 12 – Centralized topology

- **Distributed** – This is the most deployment topology (described in more detail in the OCN cloud-native infrastructure requirements) used in the fixed wireless access use case. Here the control plane components (such as Orc8r) are running in a centralized location on a K8s cluster (in the public cloud or operator datacenter), whereas the user plane component (AGW or UPF) runs on bare metal appliances (usually one appliance per cell site).

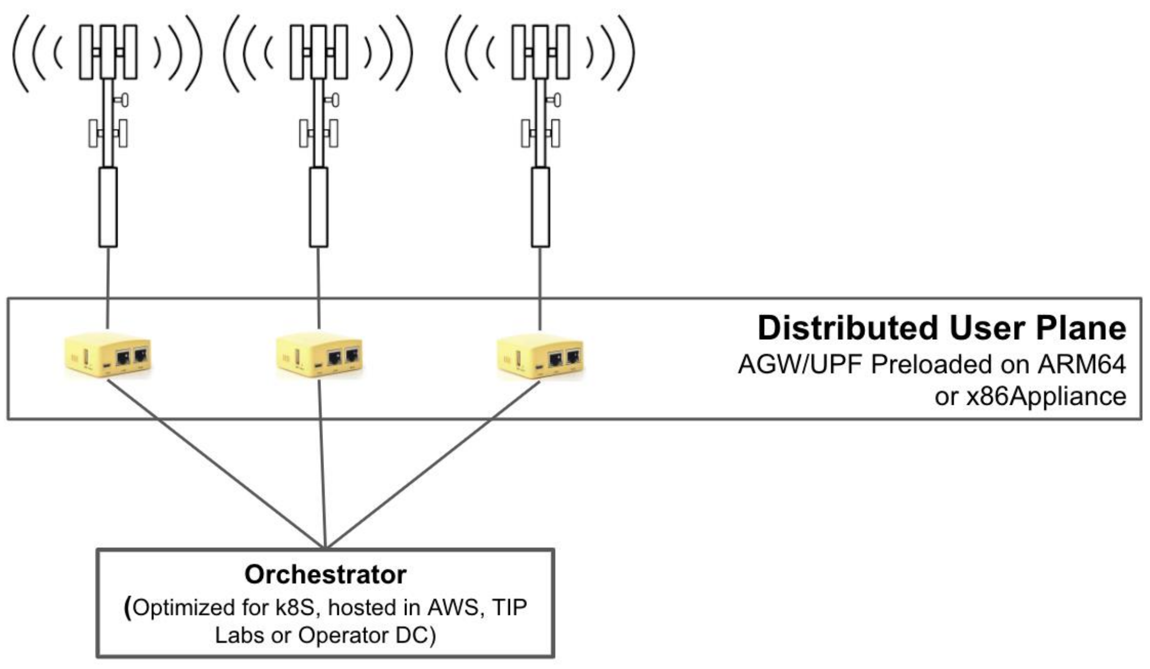

Distributed User Plane
AGW/UPF Preloaded on ARM64 or x86Appliance
Orchestrator
(Optimized for k8S, hosted in AWS, TIP Labs or Operator DC)

Figure 13 – Distributed topology

For the initial OCN version of CI/CD we'll start by supporting these two topologies and over time will add additional ones as we support more use cases and expand the scope.

Centralized topology will initially be tested by deploying all OCN code components to a TIP Labs K8s cluster, the architecture of which is described in the OCN Cloud-Native Infrastructure requirements.

Distributed topology will be tested by automating the deployment to a series of partner labs capable of replicating the distributed topology depicted in Figure 6.

6.4 OCN CNTT Conformance Testing

Per OCN cloud-native infrastructure requirements, the code for OCN microservices and deployment topologies aim to stay conformant with CNTT Reference Architecture 2 (RA 2). At least one of the continuous deployment stages must involve deploying OCN to a CNTT-compliant lab (to be deployed in TIP Labs) and running CNTT conformance tests. Completing this conformance test stage will result in one of two outcomes:

- Deployed OCN reference architecture and the K8s-based infrastructure pass the test to receive the CNTT certification badge
- Deployed OCN reference architecture fails the test and some changes to the code/ deployment topology are required to retain the CNTT certification badge

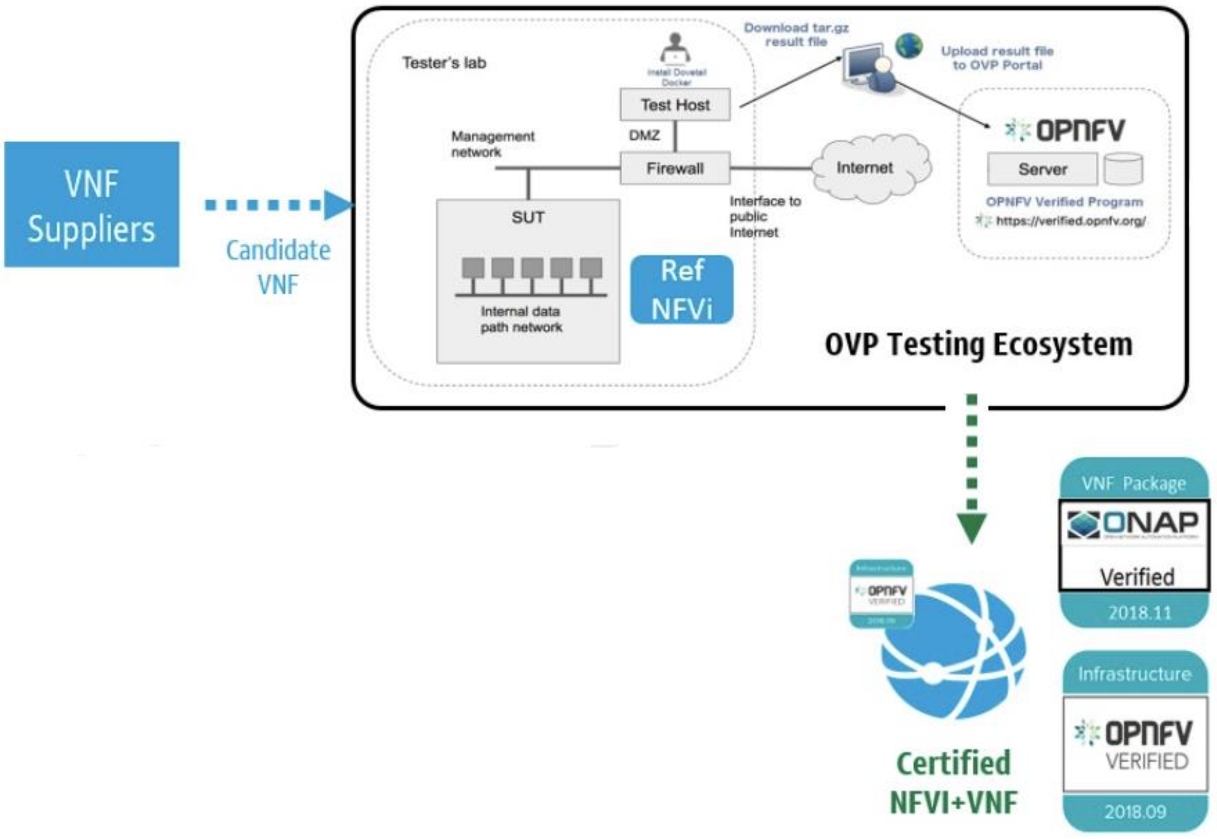


Figure 14 – Seeking CNTT certification through testing

Telecom Infra Project and CNTT will collaborate to define and build a testing harness that will continuously test all newly deployed OCN topologies in the TIP Labs staging environment.

6.5 Example Continuous Deployment Workflow

It's expected that over time the OCN CI/CD infrastructure will support multiple deployment workflows to multiple environments—some running in TIP Labs, others belonging to operator or vendor staging environments. To build on the workflow we began to describe in section 5.4 (building and integration testing of the converged-mme component), an example CD workflow could look as follows:

1. Argo Workflows fetches a recently built converged-mme container from Artifactory that has been previously saved there and tagged per CI workflow described in 5.4.
2. Argo combines the said artifact with the latest version of Helm charts. It executes a canary deployment strategy and creates a KubeVirt pod with the new access gateway (AGW) (based on latest converged-mme) in a CNTT-conformant TIP Labs cluster.
3. Argo kicks off a stage with the CNTT conformance-testing harness against a system version that uses the pod containing the newly deployed access gateway.
4. Upon passing the CNTT conformance testing, Argo continues to gradually increase the traffic load to the pod with the new AGW, phasing out the pod containing the prior version.