**Programmable edge-to-cloud virtualization fabric for the 5G Media industry**

### D5.4 - Packaging and Integration Tools evaluation and setup

**Work Package:** WP5 – 5G-MEDIA APIs and SDK Tools  
**Lead partner:** ENG  
**Author(s):** Francesco Iadanza[ENG], Francesca Moscatelli [NXW], Stamatia Rizou [SILO], Panagiotis Athanasoulis [SILO], Alex Doumanoglou [Certh], Avi Weit [IBM], David Breitgand [IBM], Igor Fritzsch [BIT], Refik Fatih Ustok [NET], Ugur Acar [NET], Selcuk Keskin [NET]  
**Delivery date (DoA):** November 30th, 2019  
**Actual delivery date:** November 29th, 2019  
**Dissemination level:** Public  
**Version number:** 1.0  
**Status:** Final

### Project details

- **Grant Agreement N°:** 761699  
- **Project Acronym:** 5G-MEDIA  
- **Project Title:** Programmable edge-to-cloud virtualization fabric for the 5G Media industry  
- **Instrument:** IA  
- **Call identifier:** H2020-ICT-2016-2  
- **Topic:** ICT-08-2017, 5G PPP Convergent Technologies, Strand 2: Flexible network applications  
- **Start date of the project:** June 1st, 2017  
- **Duration:** 30 months
Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Who</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Mar. 14th, 2018</td>
<td>ENG</td>
<td>TOC</td>
</tr>
<tr>
<td>0.2</td>
<td>Jun. 10th, 2019</td>
<td>ENG</td>
<td>updated TOC</td>
</tr>
<tr>
<td>0.3</td>
<td>Aug. 30th, 2019</td>
<td>ENG</td>
<td>First draft</td>
</tr>
<tr>
<td>0.4</td>
<td>Oct. 15th, 2019</td>
<td>ENG</td>
<td>Second draft</td>
</tr>
<tr>
<td>0.5</td>
<td>Nov. 19th, 2019</td>
<td>ENG</td>
<td>Consolidated release</td>
</tr>
</tbody>
</table>

Quality Control

<table>
<thead>
<tr>
<th>Role</th>
<th>Date</th>
<th>Who</th>
<th>Approved/Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewer</td>
<td>Nov 15th, 2019</td>
<td>Stamatia Rizou, Panagiotis Athanasoulis (SILO)</td>
<td>Comments, corrections, suggestions on structure</td>
</tr>
<tr>
<td>Reviewer</td>
<td>Nov 18th, 2019</td>
<td>David Breitgand (IBM)</td>
<td>Comments, corrections, rephrasing, typos, style, questions and suggestions for improvement</td>
</tr>
<tr>
<td>Reviewer</td>
<td>Nov 25th, 2019</td>
<td>Stamatia Rizou, Panagiotis Athanasoulis (SILO)</td>
<td>Comments and corrections</td>
</tr>
</tbody>
</table>
Disclaimer

This document may contain material that is copyright of certain 5G-MEDIA project beneficiaries, and may not be reproduced or copied without permission. The commercial use of any information contained in this document may require a license from the proprietor of that information. The 5G-MEDIA project is part of the European Community’s Horizon 2020 Program for research and development and is as such funded by the European Commission. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability with respect to this document, which is merely representing the authors’ view.

The 5G-MEDIA Consortium is the following:

<table>
<thead>
<tr>
<th>Participant number</th>
<th>Participant organisation name</th>
<th>Short name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>ENGINEERING – INGEGNERIA INFORMATICA SPA</td>
<td>ENG</td>
<td>Italy</td>
</tr>
<tr>
<td>02</td>
<td>IBM ISRAEL - SCIENCE AND TECHNOLOGY LTD</td>
<td>IBM</td>
<td>Israel</td>
</tr>
<tr>
<td>03</td>
<td>SINGULARLOGIC ANONYMI ETAIREIA PLIROFORIAKON SYSTIMATON KAI EFARMOGON PLIROFORIKIS</td>
<td>SILO</td>
<td>Greece</td>
</tr>
<tr>
<td>04</td>
<td>Hellenic Telecommunications Organization S.A. - OTE AE (Organismos TilepiKOINION Tis Ellados OTE AE)</td>
<td>OTE</td>
<td>Greece</td>
</tr>
<tr>
<td>05</td>
<td>CORPORACION DE RADIO Y TELEVISION ESPANOLA SA</td>
<td>RTVE</td>
<td>Spain</td>
</tr>
<tr>
<td>06</td>
<td>UNIVERSITY COLLEGE LONDON</td>
<td>UCL</td>
<td>United Kingdom</td>
</tr>
<tr>
<td>07</td>
<td>TELEFONICA INVESTIGACION Y DESARROLLO SA</td>
<td>TID</td>
<td>Spain</td>
</tr>
<tr>
<td>08</td>
<td>UNIVERSIDAD POLITECNICA DE MADRID</td>
<td>UPM</td>
<td>Spain</td>
</tr>
<tr>
<td>09</td>
<td>INSTITUT FUER RUNDFUNKTECHNIK GMBH</td>
<td>IRT</td>
<td>Germany</td>
</tr>
<tr>
<td>10</td>
<td>NEXTWORKS</td>
<td>NXW</td>
<td>Italy</td>
</tr>
<tr>
<td>11</td>
<td>ETHNIKO KENTRO EREVNAS KAI TECHNOLOGIKIS ANAPTYXIS</td>
<td>CERTH</td>
<td>Greece</td>
</tr>
<tr>
<td>12</td>
<td>NETAS TELEKOMUNIKASYON ANONIM SIRKETI</td>
<td>NET</td>
<td>Turkey</td>
</tr>
<tr>
<td>13</td>
<td>INTERINNOV SAS</td>
<td>IINV</td>
<td>France</td>
</tr>
<tr>
<td>14</td>
<td>BITTUBES GMBH</td>
<td>BIT</td>
<td>Germany</td>
</tr>
<tr>
<td>15</td>
<td>NATIONAL CENTER FOR SCIENTIFIC RESEARCH - DEMOKRITOS</td>
<td>NCSRD</td>
<td>Greece</td>
</tr>
</tbody>
</table>
# Table of Contents

Executive summary ........................................... 8

1. Tools to package VNF images: ISO, containers, unikernels ................. 9
   1.1. The different packaging options available to the developer .......... 9
   1.2. Plain ISO ...................................................................... 12
   1.3. Unikernel ..................................................................... 12
   1.4. Docker container .......................................................... 21
       Docker-compose .................................................................. 22
       RancherOS ..................................................................... 22
       GPU-enabled VNFs ......................................................... 24
   1.5. LXC container ............................................................. 26

2. Tools to validate and onboard VNFs/NSs descriptors ......................... 27

3. Tools to support emulation and testing ............................................. 29
   3.1. Serverless vs Regular .................................................... 29
   3.2. Unikernels ................................................................... 29

4. Tools to enable monitoring ............................................................. 31

Conclusions ................................................................. 37

References ...................................................................... 38
List of Figures

Figure 1 - OSv unikernel C/C++ generic application architecture .............................................................. 14
Figure 2 - 5G-MEDIA SDK UI for creating an unikernel repository in Gitlab ............................................. 16
Figure 3 - 5G-MEDIA SDK UI for building the Jenkins task in an existing Gitlab repository ..................... 17
Figure 4 - CI/CD custom services for the runtime based unikernel image building ................................. 17
Figure 5 - OSv file structure for the runtime-based unikernel images ......................................................... 20
Figure 6 - CI/CD custom services and end to end interaction with the SDK .............................................. 20
Figure 7 - Jenkins configuration example for Docker container images building ................................... 22
Figure 8 - RancherOS high level architecture (credits: RancherOS) .......................................................... 23
Figure 9 - File structure for the Docker based image building ................................................................. 23
Figure 10 - File structure for the LXC based image building ................................................................. 26
Figure 11 - high level validation diagram ................................................................................................. 28
## Definitions and acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CI/CD</td>
<td>Continuous Integration/Continuous deployment</td>
</tr>
<tr>
<td>CSAR</td>
<td>Cloud Service Archive</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>CNCF</td>
<td>Cloud Native Computing Foundation</td>
</tr>
<tr>
<td>Faas</td>
<td>Function as a Service</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IM</td>
<td>Information Model</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>MANO</td>
<td>Management and Orchestration</td>
</tr>
<tr>
<td>NFV</td>
<td>Network Function Virtualisation</td>
</tr>
<tr>
<td>NS</td>
<td>Network Service</td>
</tr>
<tr>
<td>NSD</td>
<td>Network Service Descriptor</td>
</tr>
<tr>
<td>OSM</td>
<td>Open Source Mano</td>
</tr>
<tr>
<td>PIE</td>
<td>Position Independent Executable</td>
</tr>
<tr>
<td>PoP</td>
<td>Point of Presence</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface for Unix</td>
</tr>
<tr>
<td>QoE</td>
<td>Quality of Experience</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SO</td>
<td>Shared Object</td>
</tr>
<tr>
<td>SVP</td>
<td>Service Virtualization Platform</td>
</tr>
<tr>
<td>TOSCA</td>
<td>Topology and Orchestration Specification for Cloud Applications</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>VDU</td>
<td>Virtual Deployment Unit</td>
</tr>
<tr>
<td>VIM</td>
<td>Virtual Infrastructure Manager</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VNF</td>
<td>Virtual Network Function</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>VNFD</td>
<td>Virtual Network Function Descriptor</td>
</tr>
<tr>
<td>YAML</td>
<td>Yet Another Markup Language</td>
</tr>
<tr>
<td>YANG</td>
<td>Yet Another Next Generation</td>
</tr>
</tbody>
</table>
Executive summary

The main goal of this deliverable is the description of the packaging and integration tools provided to the 5G-MEDIA project, complementing the SDK Tools and APIs, to collect all the necessary artefacts and generate the software packages to be uploaded on the 5G-MEDIA Catalogue.

Section 1 depicts the tools used to package the VNF images for each of options available, such as ISO, containers, unikernels. A specific section is dedicated to GPU-based serverless containers.

Section 2 describes the main tools used to validate and onboard the VNFs using the SDK and Section 3 depicts the main tools used for emulation and testing. Section 4 describes the tools to allow the resource monitoring in the VNFs.
1. Tools to package VNF images: ISO, containers, unikernels

The VNF/NS images packaging options available in the 5G-MEDIA project span different formats and cover different scenarios.

For each scenario, we present the actual packaging technology used, the benefits for the project and some emerging options as a result of a first exploration phase that could be extended in the future.

1.1. The different packaging options available to the developer

Plain ISO have been so far the most commonly used packaging format in NFV and represent the easiest way to reuse existing ISO available with sole the addition of specific configuration; they are commonly supported by the VIMs and also easy to test in local environment like VirtualBox\(^1\) or local VIM like DevStack\(^2\). As described below though, this is not always the optimal solution.

A recent (May 2019) presentation by at the 14th Annual IEEE/ACM IT Professional Conference\(^3\) clearly shown the benefits of unikernel packaging technology and the differences with plain ISOs and containers that are briefly reported to highlight the benefits brought to the NFV domain.

An ordinary Linux system image comprises not only the operating system kernel, but also a set of applications and services that might not be used by a specific application that is hosted by a VM. Consider a firewall as an application example.

Although the benefits vs disadvantages of unikernels are still being debated in the art\(^4\), in the context of NFV, smaller images have faster boot time contributing to faster response. Even though such images are stripped down to the bare minimum of application and services, they still include much more than what is actually necessary.

Another important aspect is security. Flaws in unused software can be exploited to jeopardize the system. To mitigate these problems, operating systems use “protection rings”, a mechanism to protect data and functionality from faults and malicious behavior using layers arranged in a hierarchy from most privileged (usually numbered zero, dedicated to the kernel) to least privileged (where user applications run). Access between adjacent rings is regulated to avoid misuse [Protection Rings].

Even with this approach though, flaws in the code can be used to have access to privileged layers. Some estimations [Code defects] expect between 15 to 50 defects per 1000 lines of code (LOC), not necessary all flaws of course, but still to be considered as an unmatched requirement. Just Linux kernel is estimated to be around 22 millions of LOC, the rest of operating system applications in an ordinary Linux system is from 10 to 20 times bigger (Red Hat Enterprise Linux is 420 millions of LOC).

To complete the scenario, code traces shows that typical applications use around 1% of the kernel features [Unikernels].

---

\(^1\) [https://www.virtualbox.org/wiki/Downloads](https://www.virtualbox.org/wiki/Downloads)
\(^2\) [https://docs.openstack.org/devstack/latest/](https://docs.openstack.org/devstack/latest/)
\(^3\) [http://princetonacm.acm.org/tcfpro/Brad_Whitehead_SuperContainers.pdf](http://princetonacm.acm.org/tcfpro/Brad_Whitehead_SuperContainers.pdf)
\(^4\) [https://www.joyent.com/blog/unikernels-are-unfit-for-production](https://www.joyent.com/blog/unikernels-are-unfit-for-production)
These empiric studies raise a question whether it is possible to package just the needed features in an image, keeping it smaller and safer.

An approach used by Linux application is static linking, where only the needed libraries are composed together and become immutable, so the option explored by the different unikernel packaging technologies has been to reuse this approach to include also the necessary kernel code.

In Linux operating systems, the C Standard Library\(^5\) converts standard function calls to kernel system calls and adopts a standard POSIX interface. As the common operating system functions, drivers, and protocols are written as a library of functions, instead of having the C Library passing a call to the kernel, the library could be extended and include the instructions that are actually executed by the call itself and avoid the a context switch across the user-kernel boundary. Eventually, linking together the application with this extended C Library produces what is called a “unikernel”.

Here, removing unnecessary code even from the kernel can make the difference and reduce the surface of attack: for example, the Venom vulnerability\(^6\) based on a virtual floppy disk controller flaw that was used in 2015 to compromise both physical and virtual machines.

The approach promoted by unikernels to include in binary files just the actual needed code is definitely appealing, but unikernel cannot be considered as the best solution for each scenario, as they bring some major limitations that should be taken into consideration.

The first difficulty is to debug a running instance\(^7\), that can be only partially mitigated in case of runtimes like Java, Python, NodeJS configured to expose specific application layer API such as Jolokia\(^8\) or JDWP\(^9\). The limitation about the lack of memory management like ballooning\(^10\) still exists, but it also depends on the hypervisor used that could not support it\(^11\). Some unikernel technologies employ larger kernels than needed\(^12\) and include again useless code, but the most relevant is that existing applications needs to be ported\(^13\). Moreover, they need to run in just one process, making it hard to handle advanced services like Apache Traffic server (which has three\(^14\)) unless they are repackaged in multiple unikernel images with the addition of a custom orchestration layer.

**Containers** constitute an interesting alternative to unikernel and Virtual Machines. Containers share the same kernel, so they start in milliseconds as the kernel is already running, but they are weakly isolated from the host processes and the other containers. In some popular container management systems, like Docker, a container manager must run as root and this can lead to a security breach\(^15\),

---


\(^6\) [https://venom.crowdstrike.com/](https://venom.crowdstrike.com/)

\(^7\) [https://www.joyent.com/blog/unikernels-are-unfit-for-production](https://www.joyent.com/blog/unikernels-are-unfit-for-production)

\(^8\) [https://jolokia.org/](https://jolokia.org/)

\(^9\) [https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/introclientissues005.html](https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/introclientissues005.html)

\(^10\) [https://searchservervirtualization.techtarget.com/definition/memory-ballooning](https://searchservervirtualization.techtarget.com/definition/memory-ballooning)


\(^12\) [https://github.com/cetic/unikernels#choice-of-unikernel-solution](https://github.com/cetic/unikernels#choice-of-unikernel-solution)

\(^13\) [https://github.com/cloudius-systems/osv/wiki/Porting-native-applications-to-OSv](https://github.com/cloudius-systems/osv/wiki/Porting-native-applications-to-OSv)


\(^15\) Docker security, [https://docs.docker.com/engine/security/security/](https://docs.docker.com/engine/security/security/)
although in more recent frameworks, some of the security problems inherent to Docker are being mitigated\textsuperscript{16}.

On the other hand, containers do not suffer unikernel limitations and are much easier to package, they support an incremental approach that reduces the final image size. They require additional configuration of the hypervisor to be seamlessly supported\textsuperscript{17}. In the context of NFV, there is not yet mature support for containers based NFVI. By far and large, a leader in the container orchestration domain is CNCF’s Kubernetes. Presently, Kubernetes is not supported out of the box by OSM (even though it is supported by some other orchestrators, such as ONAP\textsuperscript{18}).

**Docker containers** are used in 5G-MEDIA project for MAPE and for VNF testing, as they are the packaging format accepted by the VIM emulator included in OSM, vim-emu\textsuperscript{19} (formerly known as “son-emu”). There are plans\textsuperscript{20} to extend such tool to accept also plain VM ISO, but this option is still not available at the time of writing this document.

**5G-MEDIA serverless framework** uses Docker containers via Apache OpenWhisk\textsuperscript{21}. In case of standard runtime environments (Python, Node.js, Java, and several other), there is no packaging involved. The code in the target language is automatically injected into the preconfigured container template. Moreover, the runtime system tries to minimize an impact of cold starts and preferentially schedules new function to a prewarmed container. In case of a language that is not supported out of the box (e.g., C/C++), a container image is created using the regular Docker methodology and an OpenWhisk serverless function is created using this image.

In 5G-MEDIA, serverless VNFs are offloaded by Apache OpenWhisk to be executed at Kubernetes. This way 5G-MEDIA benefits from the best of the two approaches: a mature serverless programming model of OpenWhisk coupled with the maturity of Kubernetes for low level container orchestration.

A choice between regular and serverless functions usage depends on the expected actual time utilization of the VNFs. The cost/benefits tradeoff is optimized by a special component described in the D4.2.

**LXC/LXD containers** are also used due to better performances compared to KVM\textsuperscript{22} although they suffer the same limitation as Docker containers of not being supported by OSM yet.

To complete the description, **additional emerging packaging technologies** have been briefly evaluated as they are pursuing the easiness of the container packaging together with the security provided by unikernels. It is worth to mention KataContainers\textsuperscript{23}, Firecracker\textsuperscript{24} and gVisor\textsuperscript{25}.

\textsuperscript{16} Daniel J. Walsh, “Podman: A more secure way to run containers”, https://opensource.com/article/18/10/podman-more-secure-way-run-containers
\textsuperscript{17} https://www.openstack.org/containers/leveraging-containers-and-openstack/
\textsuperscript{18} ONAP: K8s based Cloud region support (Continue from R4), https://wiki.onap.org/pages/viewpage.action?pageId=60889650
\textsuperscript{19} https://osm.etsi.org/wikipub/index.php/VIM_emulator
\textsuperscript{20} https://github.com/sonata-nfv/son-emu/issues/273
\textsuperscript{21} https://openwhisk.apache.org/
\textsuperscript{22} https://ubuntu.com/blog/lxd-crushes-kvm-in-density-and-speed
\textsuperscript{23} https://katacontainers.io/
KataContainers relies on an OCI compliant container runtime based on QEMU that provides the same security isolation of a VM; Firecracker provides a faster cloud-native alternative to QEMU and OSv support to Firecracker has been announced in April 2019. gVisor does not support pass-through and so GPUs. They are all quite recent (2018), some have networking limitations or require custom network configuration, but they all point to the direction to sandbox containers and improve security.

1.2. Plain ISO

Plain ISO in 5G-MEDIA are distributed through the central image repository through NGINX. The CI/CD services with Gitlab and Jenkins can manage the recompilation from sources starting from publicly available make file from the opensource community. Some may include cloud-init for the so called “day-0” configuration, done when the image is started the first time and that is supported by OSM.

The main advantage of this approach is the usage of the well-known and tested make files from the community, such as those based on Packer. A major disadvantage could be the size of the image. The image size inflicts a higher booting time and inclusion of not strictly necessary libraries and services that makes the VM more vulnerable to security attacks. This is mitigated by unikernels, as described below.

1.3. Unikernel

Unikernel is a relatively new concept introduced in 2013 [Unikernel cloud]. Among many different unikernel packaging technologies available, the choice made in the first release period to integrate OSv using Mikelangelo Capstan tools has been confirmed for three main reasons: support to different hypervisors, ability to minimize the disruption to existing applications implementing POSIX-

---

24 https://firecracker-microvm.github.io/
25 https://github.com/google/gvisor
26 http://blog.osv.io/
27 https://unit42.paloaltonetworks.com/making-containers-more-isolated-an-overview-of-sandboxed-container-technologies/
28 https://github.com/kata-containers/documentation/blob/master/Limitations.md#networking-limitations
29 https://medium.com/@Pawlrus/aws-firecracker-configure-host-guest-networking-b08b90d4f48d
30 https://cloudinit.readthedocs.io/en/latest/
31 https://www.packer.io/
32 https://github.com/cetic/unikernels#state-of-the-art
33 http://osv.io/
34 https://github.com/mikelangelo-project/capstan-packages
35 https://github.com/cetic/unikernels#comparing-solutions
like interfaces\textsuperscript{36} and providing Linux ABI compatibility\textsuperscript{37}, some optimizations tailored for the NFV domain\textsuperscript{38}. Emerging alternatives like Unikraft\textsuperscript{39} seem promising and could be explored later on.

The packaging of runtime-based images (Java, Python, NodeJS) turned out to be straightforward as based on the sheer addition of application files (like .class) on top of the pre available images from OSv.

Compared with the first release period, one first result has been the testing of the packaging of C/C++ native unikernel images starting from the official list available in the OSv project repository\textsuperscript{40}.

In the end, some packaging of C/C++ required an additional effort to solve some incompatibilities in the case of ffmpeg\textsuperscript{41} and x265\textsuperscript{42}, where some deep investigation from the OSv community was necessary\textsuperscript{43}, making it hard for not insiders.

Other C/C++ native unikernel images packaging worked out of the box (for example, nginx, haproxy), with the sheer addition of a generic launcher component to allow dynamic reconfiguration that enables multiple instances of a specific function, the configuration and the output retrieval using a REST API.

The architecture of OSv images for such launcher is reported in the Figure 1, which includes the “libhttpserver-api” lib with REST API for file management and app launching\textsuperscript{44}, the specific functions compiled as shared object (SO) or position-independent executables (PIE), the basic file system services.

\textsuperscript{36} \url{https://github.com/cloudius-systems/osv/wiki/POSIX-API-support}
\textsuperscript{37} \url{https://github.com/cloudius-systems/osv/wiki/OSv-Linux-ABI-Compatibility}
\textsuperscript{38} \url{http://osv.io/nfv}
\textsuperscript{39} \url{https://github.com/sysml/ucc-unikraft/wiki}
\textsuperscript{40} \url{https://github.com/cloudius-systems/osv-apps}
\textsuperscript{41} \url{https://ffmpeg.org/}
\textsuperscript{42} \url{http://x265.org/}
\textsuperscript{43} \url{https://groups.google.com/forum/m#!topic/osv-dev/ZAjvi_vEhnA}
\textsuperscript{44} \url{https://github.com/cloudius-systems/osv/wiki/Using-OSv-REST-API}
As a result, images are packaged to start with the following parameters:

- `bootcmd: "--redirect=/log.txt /libhttpserver-api.so --config-file=/etc/httpserver.conf"`

Then it is possible to launch generic commands with:

- `curl -X PUT 'http://IP:PORT/app?new_program=true&command=<URLENCODED COMMAND>'`

For example, in case of ffmpeg+x265 the following command (to be URL encoded) manages remote source/remote destination transcoding:


Similarly, for haproxy it will be:

- `/haproxy.so -f haproxy.cfg`

In both cases, REST calls are available to upload/get configuration or log files:


This approach has been replicated with other C/C++ native unikernel images like Nginx.

Another major advancement about unikernel has been to complete integration of the packaging services within the CI/CD managed by the SDK. As a consequence, the developers can easily have access to unikernel images abstracting on the entire compilation process that, especially with native images based on C and C++, often requires advanced knowledge about the source code compilation and the tools to produce the unikernel images.

To facilitate the entire development process, two workflows are available to the developer:

- the packaging and instantiation of unikernel images in the local development environment for a first check of the functionalities before enabling the collaborative development and use the CI/CD services;
- the packaging and availability of unikernel images on a common repository automatically done by the CI/CD services when new code is pushed on the source code repository by the developers.

The **packaging in the local environment** is provided by a dedicated Docker images prepared to build runtime-based unikernels (Java 8 OpenJDK provided by Zulu\(^{45}\) or by AdoptOpenJDK\(^{46}\), Python 2.7, Node JS 8.11): the developers just has to start it, copy there the files (".class", ".py", ".js" etc.), launch the builder and then the embedded QEMU to have the service running.

Such Docker images are based on similar tools available from the opensource community\(^{47} \ 48\) but unified, simplified to manage the runtime-based unikernel images and with some examples.

The examples describe how to create a simple server based on Java SpringBoot, Python or NodeJS starting from sample projects on Gitlab\(^{49} \ 50 \ 51\). Specific shell scripts are provided to build and launch locally with QEMU the unikernel image for the tests: the only information required from the developer is the name of the package to produce and the name of the main file to be launched.

It is here reported the steps for Java; those for Python and NodeJS are very similar.

1) Launch the Docker instance and publish an external port (in this case 8000) for the service to be started:

```
docker run -p 8000:8000 -it docker5gmedia/osv_composer bash
```

2) Launch the shell script without parameters to get the help, where **testjava1** is the example package name

```
./create_template_java.sh
```

**Usage:** `create_template_java.sh <package_name> <jar_file_name>
```

**example:**

```
git clone https://production.eng.it/gitlab/5G-MEDIA/cicd_example_java.git
cd cicd_example_java
mvn package
cd target
PACKAGE_NAME=testjava1
/scripts/create_template_java.sh cicd_example_java-0.0.1-SNAPSHOT.jar
capstan package compose -s 512M
capstan run -p qemu --boot default -f "8000:8000"
```

3) Once the instructions are executed, the image is built and stored, then the service is responding on the port 8000

The last command actually wraps the QEMU tools (such as `qemu-system-x86_64`) to launch the image called **“testjava1.qemu”** stored locally from within the Docker environment for functional tests.

---


\(^{46}\) [https://adoptopenjdk.net/about.html](https://adoptopenjdk.net/about.html)

\(^{47}\) [https://github.com/wkozaczuk/docker_osv_builder](https://github.com/wkozaczuk/docker_osv_builder)

\(^{48}\) [https://github.com/wkozaczuk/docker_osv_runner](https://github.com/wkozaczuk/docker_osv_runner)

\(^{49}\) [https://github.com/5g-media/cicd_example_java.git](https://github.com/5g-media/cicd_example_java.git)

\(^{50}\) [https://github.com/5g-media/cicd_example_python.git](https://github.com/5g-media/cicd_example_python.git)

\(^{51}\) [https://github.com/5g-media/cicd_example_nodejs.git](https://github.com/5g-media/cicd_example_nodejs.git)
/root/.capstan/repository/testjava1/testjava1.qemu

Such image can also be copied outside the Docker instance and launched on a separate hypervisor like VirtualBox or OpenStack/DevStack for performance tests.

docker cp `docker ps | grep osv/composer | awk '{print $1}'`
:/root/.capstan/repository/testjava1/testjava1.qemu testjava1.qemu

As a result, the developer can completely abstract from the container packaging details and get runtime-based images using a preconfigured environment.

The packaging through CI/CD of runtime-based unikernels is triggered by the developer activities over the SDK. The SDK UI provides a dedicated interface to push code on Gitlab, then to use templates to define the name and type (Java, Python, Node) of the unikernel project that invokes a custom CI/CD service that configures (and runs) the Jenkins building task that ends up producing the unikernel image in the central image repository (See Figure 2 for creating the repository and Figure 3 for the building the Jenkins task).

![Figure 2 - 5G-MEDIA SDK UI for creating an unikernel repository in Gitlab](image-url)
The custom CI/CD services for the packaging of runtime-based unikernels are designed to automatically configure Jenkins tasks hiding the Mikelangelo Capstan tools setup and allowing a seamless experience through the SDK. Such CI/CD service acts as a Jenkins proxy and relies on a pool of empty tasks that are configured on request. The API is shown in the Figure 4.

```
GET /availableTaskPoolNames getAvailableTaskPoolNames
GET /createTask/(taskName)/(projectName)/(projectType)/(userlist) createTask
GET /deleteTask/(taskName) deleteTask
GET /getlastBuildDetails/(taskName) getLastBuildDetails
GET /getlastBuildLog/(taskName) getlastBuildLog
GET /getlastBuildResult/(taskName) getLastbuildResult
GET /runTask/(taskName) runTask
GET /simulateTask/(taskName)/(projectName)/(projectType)/(userlist) simulateTask
GET /usedTaskPoolNames getUsedTaskPoolNames
GET /version version
```

The detailed specifications for the custom CI/CD are reported below. All methods will return HTTP code 200 in case of correct execution, HTTP code 50x code (and a textual message) if an error occurs.

- **createTask**: accepts the name of the task to be created, the name of the Gitlab project to link, the “projectType” to produce an unikernel image based on runtime (with possible values “java”,

---

**Figure 3 - 5G-MEDIA SDK UI for building the Jenkins task in an existing Gitlab repository**

**Figure 4 - CI/CD custom services for the runtime based unikernel image building**
“python”, “nodejs”), the comma-separate user list to be allowed to access the configured Jenkins task to check the console output and returns the full name of the created task. The CI/CD service will remove a task from the free pool, configure it and make it available to the SDK; the method returns the name of the created task;

deleteTask: accepts the name of the task to be deleted. The CI/CD service will clean up the task configuration and put back the task in the available pool;

getLastBuildDetails: accepts the name of the task to get the build information about. The CI/CD service will return the build status summary as a JSON file with the following data:

- building: [true|false];
- durationMS: [amount of time required for the latest build measured in ms];
- estimatedDurationMS: [estimated amount of time required for the build measured in ms];
- result: [“ABORTED”|“FAILURE”|“NOT_BUILT”|“SUCCESS”|“UNSTABLE”] as a wrapper of the Jenkins CI “Result” model;

getLastBuildLog: accepts the name of the task to get the log information about. The CI/CD returns the latest task build log in textual representation;

getLastBuildResult: accepts the name of the task to get the build status. The CI/CD service will return the build status:

result: [“ABORTED”|“FAILURE”|“NOT_BUILT”|“SUCCESS”|“UNSTABLE”] as a wrapper of the Jenkins CI “Result” model;

availableTaskPoolNames: returns a space separated name list of the empty pool tasks used to check the current availability from the empty pool;

usedTaskPoolNames: returns a space separated name list of the used tasks from the pool;

simulateTask: accepts the name of the task to be created, the name of the Gitlab project to link, the “projectType” to produce an unikernel image based on runtime (with possible values “java”, “python”, “nodejs”), the comma-separate user list to be allowed to access the configured Jenkins task to check the console output and returns the full name of the created task. The CI/CD service returns the XML Jenkins configuration of a task without allocating from the pool. This is used to check the configuration and allow the debug of CI/CD services;

version: returns the release number of the CI/CD service to allow the synchronization between the current CI/CD services and the expected release on the SDK.

To allow the correct runtime based unikernel image, the related Gitlab project referred by “projectName” requires some specific OSv files that depend on the “projectType” value.

For “projectType” valued “java”, the Gitlab project needs to have a “meta” folder with two files inside:

- “package.yaml” with content

---

52 https://javadoc.jenkins-ci.org/hudson/model/Result.html
53 https://javadoc.jenkins-ci.org/hudson/model/Result.html
name: <name of the Gitlab project>
title: <possible description of the project>
author: <reference to the author of the project>
created: <timestamp of the project in the format 2019-01-29T12:12:12Z>
require:
- openjdk8-zulu-full <this will include the OpenJDK from Zulu>

- “run.yaml” with content
  runtime: java
  config_set:
    main_conf:
      main: /<path to the jar file to be launched>
      args:

    config_set_default: main_conf

For “projectType” valued “python”, the Gitlab project needs to have a “meta” folder with two files inside:

- “package.yaml” with content
  name: <name of the Gitlab project>
title: <possible description of the project>
author: <reference to the author of the project>
created: <timestamp of the project in the format 2019-01-29T12:12:12Z>
require:
- osv.cli

- “run.yaml” with content
  runtime: python
  config_set:
    main_conf:
      main: /<path to the python file to be launched>
      args:

    config_set_default: main_conf

For “projectType” valued “nodejs”, the Gitlab project needs to have a “meta” folder with two files inside:

- “package.yaml” with content
  name: <name of the Gitlab project>
title: <possible description of the project>
author: <reference to the author of the project>
created: <timestamp of the project in the format 2019-01-29T12:12:12Z>
require:
- osv.cli

- “run.yaml” with content
  runtime: node
  config_set:
    main_conf:
      main: /<path to the javascript file to be launched>
      args:

    config_set_default: main_conf

The summarized file structure created by the SDK is reported in the Figure 5.
As a result, the actions taken by the SDK to setup the proper environment on the CI/CD services to build a unikernel image based on a runtime are:

1. SDK creation of the Gitlab project using the Gitlab CI API;
2. SDK invocation of the custom API to configure a Jenkins task for unikernel runtimes;
3. SDK creation of the required OSv files on the Gitlab project to allow the correct build;
4. Jenkins automatic check about changes on the Gitlab project to trigger the image building and its upload on the repository.

The end-to-end high level workflow is shown in the sequence diagram in Figure 6.
In the end, the developer is basically required to specify the name and type of the unikernel image, add the code, then the SDK and the custom CI/CD services will take care of the rest, with the automatic production of the images available on the project repository.

All the OSv unikernel images can be configured to include\(^{54}\) cloud-init for the “day-0” configuration just like the plain ISO.

The procedure requires to build the image without capstan tool starting from the local OSv development environment described in the section 5.2 and:

1. change the modules/cloud-init/cloud-init.yaml and have
   ```yaml
   include:
     - load-from-cloud
   ```
2. build the image (“my image”) with the addition of “cloud-init”
   ```sh
   ./scripts/build image=cloud-init,<my_image>
   ```
3. copy the image on the VIM (e.g. OpenStack) located in
   `/build/last/usr.img`
4. instantiate the image with
   ```sh
   #cloud-config
   files:
     /etc/config:
       - my_config
     goes here
   ```

   This can be to provide the “specific configuration file” shown in the Figure 1 and have the instance run the service at boot without first waiting to pass the haproxy.cfg and explicitly launch it.

1.4. **Docker container**

The Docker packaging regards both serverless (FaaS) and regular (not-FaaS) VNFs images, where the CI/CD services are configured to automatically trigger the builds whenever a change to the code is detected on the related Gitlab projects.

Each Gitlab project is expected to have a “Dockerfile”\(^{55}\) in the root configured to start the initial process and expose the correct ports. Once the image is built, it is automatically pushed to the 5G-MEDIA Docker repository at [https://hub.docker.com/u/docker5gmedia](https://hub.docker.com/u/docker5gmedia).

An example of CI/CD services (with Gitlab and Jenkins) configured to manage the building of a Docker container image is shown in the Figure 7 with the *build*, *tag* and *push* Docker commands.

---

\(^{54}\) [https://github.com/cloudius-systems/osv/wiki/cloud-init](https://github.com/cloudius-systems/osv/wiki/cloud-init)

\(^{55}\) [https://docs.docker.com/engine/reference/commandline/build/](https://docs.docker.com/engine/reference/commandline/build/)
As anticipated in the previous section, Docker containers are used mainly for testing purposes on vim-emu emulator; for production environment, Docker containers cannot be run as-is due to the lack of support to Docker/Kubernetes as a VIM in OSM.

**Docker-compose**

In the case of 5G-MEDIA MAPE component, Docker containers are packaged along with a docker-compose file to enable the deployment as a multi-container application, similarly to what happens with OSM. The main advantages of this approach is to be able to define additional configuration about shared volumes for data, ports, inter container connectivity and policy for scaling that allows to deploy a more effective cluster of container and manage their status as a whole. From the packaging perspective, this only requires a “docker-compose.yaml” file to be added to the Gitlab project.

**RancherOS**

A packaging option has been integrated in the project to build either a Docker container or a plain ISO starting from a container image and avoid an additional build step using a different build file; such option is based on RancherOS and allows to quickly create a VM image based on an existing Docker

---

56 https://docs.docker.com/compose/
58 https://rancher.com/
container to perform first tests. CI/CD services support generic scripts (e.g. based on Packer\(^{59}\)) to allow the custom building of images whenever this option cannot be used.

RancherOS follows a similar approach compared to unikernel, it uses a small code base that provides a reduced attack surface and improved security.

In fact, RancherOS is a minimalistic OS that includes the strictly needed amount of software to run Docker and most of system services are packaged in a “system” container that is responsible for the management of the other containers; this minimizes the security patches and maintenance. The high level architecture is shown in the Figure 8.

![Figure 8 - RancherOS high level architecture (credits: RancherOS)](image)

As a result, developers can use the CI/CD services to produce Docker containers for testing and have ready also the ISO on the central repository using just one build file (the Dockerfile) without the need to have a separate build file for plain ISO.

The custom CI/CD API are the same as unikernels, with “projectType” valued either “docker” (to produce a container) or “dockerVM” (to produce an ISO) and the expected Dockerfile file in the Gitlab project root folder used for the image build. The easy file structure created on Gitlab by the SDK for Docker images is reported in the Figure 9.

![Figure 9 - File structure for the Docker based image building](image)

A final note about the usage of Docker containers in FaaS: Apache OpenWhisk automatically takes care of the deployment on the functions so the only CI/CD service involved is just Gitlab as source code repository for collaborative development.

---

\(^{59}\) [www.packer.io/](https://www.packer.io/)
**GPU-enabled VNFs**

The 5G-MEDIA platform developed a first of a kind support for serverless functions that can use GPUs via extending the Apache OpenWhisk serverless framework. This is first of a kind capability for serverless computing.

With these extensions, serverless VNFs can utilize GPUs, when Kubernetes compute nodes with proper hardware are available. All FaaS VNFs that utilize GPUs are bundled as docker images that are based on the ‘docker5gmedia/5gmedia-base-gpu’ image which is uploaded on the standard Docker Hub repository. This base image is in turn based on an Ubuntu Linux image provided by NVIDIA, the well-known GPU manufacturer.

Differently from the other Docker images used in the 5G-MEDIA project that start from vanilla Ubuntu or Debian, the base Docker image for GPU based containers (“5gmedia-base-gpu”) extends from NVIDIA (“nvidia/cuda:8.0-runtime-ubuntu16.04”) as can be seen from the first line below.

Such NVIDIA base image, in turn, allows to leverage on the host GPUs and includes the CUDA drivers and libraries used for that purpose; CUDA architecture is described in the section 4.4 of the Deliverable 3.1 [5G-MEDIA D3.1] and in the Deliverable 3.2 [5G-MEDIA D3.2] for FaaS.

Below we provide the contents of the 5gmedia-base-gpu Dockerfile for completeness:

```dockerfile
FROM nvidia/cuda:8.0--runtime-ubuntu16.04

# Upgrade and install basic Python dependencies for this black-box action
RUN apt-get update &
RUN apt-get install -y bash

# vim
RUN apt-get install vim

# curl
RUN apt-get install curl

# gcc
RUN apt-get install gcc

# libc-dev
RUN apt-get install libc-dev

# python-pip
RUN apt-get install python-pip

# Install the required Python packages
RUN pip install --upgrade pip
RUN pip install --upgrade pip
RUN pip install setuptools
RUN pip install six

# Install the required Python packages
RUN pip install gevent==1.2.1
RUN pip install flask==0.12
RUN pip install requests==2.13.0

# Do not modify -- this is the internal opemwhisk invoker service port
ENV FLASK_PROXY_PORT 8080

RUN mkdir -p /actionProxy
ADD actionproxy.py /actionProxy/

RUN mkdir -p /action
ADD stub.sh /action/exec

RUN chmod +x /action/exec

# for the configuration service to push parameters
```


RUN mkdir -p /conf
# base cli to be used by the app for retrieving single parameter
ADD get-conf /
RUN chmod +x /get-conf

CMD ["/bin/bash", "-c", "cd actionProxy && python -u actionproxy.py"]

When creating the OpenWhisk action, a special annotation is required which tells the Kubernetes scheduler to deploy the image on a GPU-capable node. The annotation is provided through a json file which is given as a command line argument to the OpenWhisk’s cli\(^\text{62}\) that is used to create the action. An example of the command line execution of wsk to create a gpu-enabled action:

```
openwhisk/bin/wsk -i action create /guest/5g-media/vtranscoder_2_8_4 -A gpu-annotations.json --docker docker5gmedia/transcoder_2_8_4
```

In the above command line example, the argument following “action create” is the name of the action. The name of the action is fully qualified (i.e. comprises namespace, package name and the action itself). The annotations are given through the “-A” flag and the docker image which this action is linked to, is specified after the “--docker” flag.

The contents of the gpu-annotations.json file are:

```
{
  "placement": {
    "invoker-selector": {
      "processor": "gpu"
    }
  }
}
```

In the annotation json file the field “processor” can have two values: either “cpu” or “gpu” indicating the hardware requirements of the VNF.

In addition, annotation file can be added with the field “action-antiaffinity” that acts as a boolean flag to denote whether more than one instance of a VNF action can co-exist on a same Kubernetes node. If set to True, only one VNF instance will be deployed on same node.

GPU-capable nodes should be prepared in advance, before joining them to Kubernetes cluster. They need to be installed with Nvidia-cuda driver\(^\text{63}\) and nvidia-docker\(^\text{64}\) which is a special docker runtime that can expose GPU to container.

Finally, after joining\(^\text{65}\) them to the cluster a special Kubernetes device plugin (nvidia plugin\(^\text{66}\)) should be installed. Kubernetes uses that plugin to identify and manage the GPU-capable nodes.

---

\(^{62}\) [https://github.com/apache/incubator-openwhisk-cli](https://github.com/apache/incubator-openwhisk-cli)


\(^{64}\) [https://github.com/NVIDIA/nvidia-docker](https://github.com/NVIDIA/nvidia-docker)


When a GPU VNF is instantiated, the action POD is created with `nvidia.com/gpu: 1`. This instructs Kubernetes scheduler to place it on a GPU-capable node and expose a whole GPU to it\(^67\).

The final step to compose the NS follows the SDK pipeline described in the Deliverable D5.1 [5G-MEDIA D5.1] which is the same, regardless of the GPU requirements of the VNF.

### 1.5. LXC container

Support to LXC containers in 5G-MEDIA project has been investigated due to the claimed better performances compared to KVM.

From the perspective of packaging, the CI/CD services with Jenkins support the creation of containers images for LXC based on “distrobuilder” tool\(^68\), which results are stored in the central image repository as LXC images.

The CI/CD services manage the building of plain Linux OS images (based on the supported list\(^69\)) using custom build files (in \texttt{yaml} format) that include the additional libraries needed.

To have LXC containers running in a VIM requires support to LXD supervisor either through an extension of OpenNebula VIM plugin for OSM to manage LXDoNe\(^70\), or the equivalent for OpenStack VIM plugin to manage its specific configuration for LXD\(^71\), both not available in the actual 5G-MEDIA testbeds. Nevertheless, support to LXC packaging is provided by CI/CD services in case the testbed configuration will change in the future.

The custom CI/CD API are the same as unikernels, with “projectType” valued “lxc” and the expected \texttt{yaml} file in the Gitlab project root folder used for the image build. The file structure created by the SDK on Gitlab is similar to the Docker based images shown before, but with a \texttt{yaml} file (see the Figure 10).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure10.png}
\caption{File structure for the LXC based image building}
\end{figure}

\(^{67}\) Currently, Kubernetes does not support GPU sharing and NVIDIA only support sharing of GPUs only on certain (high end) GPU models.


\(^{69}\) [https://github.com/lxc/distrobuilder/tree/master/doc/examples](https://github.com/lxc/distrobuilder/tree/master/doc/examples)

\(^{70}\) [https://github.com/OpenNebula/addon-lxdone](https://github.com/OpenNebula/addon-lxdone)

\(^{71}\) [https://docs.openstack.org/charm-guide/latest/openstack-on-lxd.html](https://docs.openstack.org/charm-guide/latest/openstack-on-lxd.html)
2. **Tools to validate and onboard VNFs/NS descriptors**

This section contains the specifications for the tools used for the packaging, validation and onboarding of serverless (FaaS) and regular (non-FaaS) VNF/NS descriptors using the SDK.

2.1. **Information Model Specifications**

The specification of the descriptors for Network Services (NSs) and Virtual Network Functions (VNFs) in 5G-MEDIA has the objective of aligning with the latest standards available in ETSI NFV [ETSI2019]. The first phase involves the specification of language independent information models that identify the content of the descriptors through a structured tree of inter-correlated and inter-dependent information elements. The second phase translates these information models into data models based on specific languages and package formats, for example, the TOSCA YAML Cloud Service Archive (CSAR) [TOSCA] or the YANG language [YANG]. Developing the network service descriptors (NSD) and virtual network function descriptors (VNFD) based on TOSCA models is actually quite common in existing NFV frameworks; for example, Cloudify [CLOUDIFY], OpenStack Tacker [OSTACKER], OpenBaton [OBATON] and ONAP [ONAP] support TOSCA-based descriptors. Therefore, TOSCA has been also selected for the reference format in 5G-MEDIA descriptors. More details on TOSCA models can be found on D4.1 5G-MEDIA Catalogue APIs and Network Apps.

While the descriptors for FaaS services follow the same TOSCA template as of the ones for non-FaaS services, in FaaS services there is no VNF “image” in the regular sense (see D3.2 for details). A single VDU in the serverless case is an OpenWhisk action that can either be based on a standard Docker container template (and in this case the “image” is the code of the action) or a “black box” action (in which case the image is the Docker container image specific to this action). In addition, there might be some other artifacts pertaining to the VNF image definition, such as triggers, rules, and packages binding all the artifacts together.

2.2. **Packaging of the VNFs descriptors based on CSAR format**

ETSI presented a common VNF package format, based on the TOSCA Cloud Service Archive (CSAR) standard. A CSAR file is, mainly, a .zip file with a well-defined structure. The whole VNF package, or the manifest file inside it, is digitally signed. Moreover, each of the artefacts contained on the package could be signed. In addition, asymmetric or symmetric encryption may be added in order to improve security. Briefly, a VNF package in CSAR format consists of the following:

- **Metadata information**: Information about the name of the package.
- **VNF Descriptor**: A VNF Descriptor is the main TOSCA definitions YAML file in which metadata for package onboarding and VNF management is presented.
- **Manifest file**: It provides the package integrity and authenticity. It contains some metadata about the VNF.
- **Change history file**: As the title says, it is a human-readable text file that logs all the changes.
- **Testing files directory**: The provider includes other files on this directory, containing more necessary information, such as the description of the tests.
- **Licensing information directory:** A license term for the whole VNF as well as other license terms for other artifacts.
- **Certificate files:** The files with extension .cert which aim to add security to the VNF package.
- **Additional files:** These could include vendor specific files such as logo.

The SDK Validator Web UI is the main interface for the developers to write the network service descriptors in YAML format and validate them against the TOSCA schema (a schema based on OSM information model is also available to use). Furthermore, developers can also package the VNFs in CSAR format after successfully validating them. CSAR packaging tool is simply a JavaScript application utilizing jarchivelib [JARCH] which is an open-source archiving and compressing library that provides a thin and easy-to-use API layer. After packaging, one can onboard network services and virtual network functions to the private catalogue just simply pressing a button on the Validator Web UI. 5G-MEDIA Validator Web UI has been developed based on the architecture given in [D5.1 5G-MEDIA SDK Programming Tools] in order to check if given descriptors are compliant with 5G-MEDIA platform.

The Figure 11 shows the high level validation diagram.

![Figure 11 - high level validation diagram](image)

The validator first converts the input file into a JSON object and then validates the parsed JSON object against a schema. The validator returns success or failure result, also specifying a reason and an error code. The validator uses both the TOSCA and the OSM IM as schema definitions to validate NSD and VNFD files. An opensource tool, named Tv4 [TV4], has been used to validate the JSON formatted VNFD and NSD files against the input schema file. More details are provided in the [D5.1 5G-MEDIA SDK Programming Tools].

The information model of the serverless (FaaS) and regular (non-FaaS) services is the same as of the TOSCA based model that FaaS services follow. The main difference with non-FaaS services is that there are VNF “images” in the regular sense but just the name of the action to be executed.
3. **Tools to support emulation and testing**

This section contains the specification of the tools to enable the development, testing and emulation of VNF in a local environment, that comprehends FaaS, plain ISO and unikernel images to allow the developers to quickly test their features before uploading them on the VIMs.

3.1. **Serverless vs Regular**

The 5G-MEDIA Emulator facilitates local prototyping and testing of NSs in realistic end-to-end multi-PoP scenarios in multi-vim environments supporting both FaaS and non-FaaS development. The emulation platform allows the execution of real network functions packaged as Docker containers in emulated network topologies running locally on the developer's machine. The emulation platform not only offers Openstack-like APIs for each emulated PoP but also provides OpenWhisk APIs via Lean OW and Minikube for FaaS development. Furthermore, the FaaS emulator is not fundamentally different from the SVP, which uses a full clustered installation of Apache OpenWhisk and of a Kubernetes (k8s) cluster. A FaaS VNF that is to be emulated, should be pre-onboarded into Lean OW in a regular way using a wskdeploy tool. This tool allows developers to define the OpenWhisk action that implements this VNF. As a next step, a VNFD should be defined for the VNF and onboarded to the private catalogue via 5G-MEDIA Validator Web UI. Again, this process is not different from that of the regular onboarding.

5G-MEDIA Emulator also integrates with management and orchestration (MANO) system, which is responsible to deploy and manage the NSs tested in the emulated environment. In order to install the emulation environment that only supports non-FaaS emulation, thus includes vim-emu (openstack based), the recommended specs are 2 vCPUs, 8 GB RAM, 40GB disk and a single interface with Internet access. In order to support both FaaS and non-FaaS development in a local environment, one needs the complete 5G-MEDIA emulation environment including both vim-emu and faas-vim (a.k.a. kubernetes vim, openwhisk based). In this case, the recommended specs are 2 vCPU, 12 GB RAM, 80GB disk and a single interface with Internet access.

More information about the internal design of faas-vim can be found in deliverable [D3.2 - Specification of the 5G-MEDIA Serverless Computing Framework]. Please also refer to this deliverable (Sections 4.1 and 4.2) for the details of configuring a VNF at the time of instantiation (Day 0) and dynamic configuration when the VNF starts executing (Day 1), respectively.

3.2. **Unikernels**

The vim-emu tool used by OSM is based on Docker containers images, so it cannot be used to emulate directly unikernel VNF images which, in the end, are plain ISO images just smaller in size. As described in its backlog\(^{72}\), a new feature is going to be added to vim-emu, the support to Containernet 2.0 that will finally enable to use Docker containers and KVM VMs (that means also unikernels) together in

---

\(^{72}\) [https://github.com/sonata-nfv/son-emu/issues/273](https://github.com/sonata-nfv/son-emu/issues/273)
emulated networks. This feature should be available soon in OSM\textsuperscript{73}, although not yet at the time of writing this document.

On the other hand, unikernel images are smaller by definition, so they can be very easily instantiated and tested on all in one VIMs supporting KVM\textsuperscript{74}. Several options are available, for example based on OpenStack (DevStack\textsuperscript{75}) or OpenNebula (MiniOne\textsuperscript{76}), both can be installed either on bare metal or VirtualBox with few easy commands\textsuperscript{77}, but they have relevant requirements: 2 vCPU, 8GB RAM and 60 GB disk for DevStack\textsuperscript{78} (but it runs with 2 vCPU, 2.5GB RAM, 8GB disk\textsuperscript{79}), 2 vCPU, 2GB RAM and 20 GB disk for MiniOne\textsuperscript{80} and need to be added to the requirements from the previous section (for example those for OSM\textsuperscript{81}) in case of an end-to-end test. So, depending on the resource availability on the developer laptop, these tools can be installed locally or be provided on request on the 5G-MEDIA testbeds.

For the unikernels, Docker and QEMU\textsuperscript{82} play an important role for the first development and testing phase in a local environment, before the code is shared on Gitlab for collaborative development and leverages the CI/CD services for automatic image building described in the previous section.

The developer is provided two Docker images publicly available on DockerHub (docker5gmedia/osv_composer and docker5gmedia/osv_runner) that comprehend: a preconfigured OSv development environment, the OSv runtime unikernel images for Java (with javac compiler), NodeJS and Python, the Mikelangelo capstan tool to build the images and the QEMU environment to launch and instantiate them locally. The Docker images can also be run from scratch and customized starting from the Dockerfile available on Gitlab\textsuperscript{83} and configured itself with CI/CD to automatically publishing on DockerHub every time a change to the code is detected.

These two containers come with some easy step-by-step examples shown during the 5G-MEDIA coding event in September 2019\textsuperscript{84}.

\textsuperscript{73}https://osm.etsi.org/gitweb/?p=osm/vim-emu.git;a=commit;h=5e0efe50e974e1271d423abf575fe1b9b9a97b49
\textsuperscript{74}https://www.linux-kvm.org/page/Main_Page
\textsuperscript{75}https://docs.openstack.org/sahara/latest/contributor/devstack.html
\textsuperscript{76}https://github.com/OpenNebula/minione/tree/v5.6.0
\textsuperscript{77}https://docs.openstack.org/devstack/latest/ and https://opennebula.org/minione-install-opennebula-with-a-single-command/
\textsuperscript{78}https://docs.openstack.org/sahara/latest/contributor/devstack.html
\textsuperscript{79}http://ronaldbradford.com/blog/setting-up-ubuntu-on-virtualbox-for-devstack-2016-03-30/
\textsuperscript{80}https://opennebula.org/try-opennebula-lxd-with-minione-2/
\textsuperscript{81}https://osm.etsi.org/wikipub/index.php/OSM_Release_FIVE
\textsuperscript{82}https://www.qemu.org/
\textsuperscript{83}https://github.com/5g-media/osv_composer
\textsuperscript{84}http://www.5gmedia.eu/2019/09/10/5g-media-coding-event-etsit-upm-madrid/
4. Tools to enable monitoring

The approach used in 5G-MEDIA to enable VNF monitoring is to have 1) publishers for the NFVI/VIM that extract the **NFVI/VIM metrics**, then 2) publishers to be embedded into the VNF images to allow the extraction of **application level metrics**, both sending data to the 5G-MEDIA Kafka bus.

About the **NFVI/VIM monitoring**, the following publishers have been implemented:

- **OpenStack/Ceilometer publisher**: publishes the monitoring metrics per VDU (VM) from the OpenStack NFVI/VIM in the 5G-MEDIA Kafka bus
- **Kubernetes publisher**: publishes the monitoring metrics per VDU (container) from the Kubernetes/FaaS NFVI/VIM in the 5G-MEDIA Kafka bus
- **OpenNebula publisher**: publishes the monitoring metrics per VDU (VM) from the OpenNebula NFVI/VIM in the 5G-MEDIA Kafka bus

Each publisher interacts with one or more services per NFVI/VIM and feed the publish/subscribe mechanism of 5G-MEDIA platform (5G-MEDIA Kafka bus) with monitoring metrics of the deployed VNFs (in the infrastructure point of view). These publishers can be deployed individually as docker containers inside the NFVI or remotely if VPN tunneling is available.

The **OpenStack/Ceilometer publisher** operates as a multi-threaded UDP repeater. It uses as base image the “python:3.5.5-slim” one, available in the Dockerhub repository, and is available in the 5G-MEDIA gitlab repository. The **Dockerfile** file looks like:

```
FROM python:3.5.5-slim
RUN apt-get update && apt-get -y upgrade && \
    apt-get -y install apache2 python3 default-libmysqlclient-dev python3-dev python3-setuptools nano && \
    mkdir -p /opt/ceilPublisher
COPY ./ /opt/ceilPublisher
RUN pip3 install -r /opt/ceilPublisher/requirements.txt
ADD ./run.sh /opt/ceilPublisher/run.sh
RUN chmod 0755 /opt/ceilPublisher/run.sh
RUN ls -la /opt/ceilPublisher/*
CMD ["/opt/ceilPublisher/run.sh"]
```

It includes a configuration file called **publisher.conf** that includes the following variables:

- the port of the UDP server (default port is 10000),
- the IPv4 of the Kafka bus
- the port in which the Kafka bus listens to (default port is 9092), and
- the topic of the Kafka bus in which the metrics are published

The image can be built using the command

85 https://kafka.apache.org/
86 https://hub.docker.com/_/python
87 https://github.com/5g-media/openstack-kafka-publisher
$ sudo docker build -t ceilometer_kafka_publisher .

while it can be instantiated using the command

$ docker run -d --name ceilometer_kafka_publisher -p 10000:10000/udp --restart always ceilometer_kafka_publisher

The *Kubernetes publisher* uses as base image the “ubuntu:16-04” image, available in Dockerhub repository\(^{88}\), and is available in the 5G-MEDIA gitlab repository\(^{89}\). Its *Dockerfile* file looks like:

```bash
FROM ubuntu:16.04
MAINTAINER Athanasoulis Panagiotis
LABEL version="1.0"

ENV DEBUG=$DEBUG
ENV KAFKA_IP=$KAFKA_IP
ENV KAFKA_PORT=$KAFKA_PORT
ENV KAFKA_KUBERNETES_TOPIC=$KAFKA_KUBERNETES_TOPIC
ENV PROMETHEUS_HOST=$PROMETHEUS_HOST
ENV PROMETHEUS_PORT=$PROMETHEUS_PORT
ENV PROMETHEUS_POLLING_STEP=$PROMETHEUS_POLLING_STEP
ENV SCHEDULER_SECONDS=$SCHEDULER_SECONDS

RUN pwd
RUN apt-get update
RUN apt-get -y upgrade
RUN apt-get -y install python3-dev python3-setuptools python3-pip supervisor vim \\
    && rm -rf /var/lib/apt/lists/*

RUN mkdir /opt/k8s-prometheus-publisher
COPY k8s-prometheus-publisher /opt/k8s-prometheus-publisher
RUN ls -la /opt/k8s-prometheus-publisher

RUN pip3 install -r /opt/k8s-prometheus-publisher/requirements.txt \\
    && cp /opt/k8s-prometheus-publisher/deployment/k8s-prometheus-publisher.conf /etc/supervisor/conf.d/k8s-prometheus-publisher.conf \\
    && cp /etc/supervisor/supervisord.conf /etc/supervisor/supervisord.conf \\
    && chmod +x /opt/k8s-prometheus-publisher/deployment/run.sh

EXPOSE 3333
ENTRYPOINT ["/bin/sh"]
CMD ["-c", "/opt/k8s-prometheus-publisher/deployment/run.sh"]
```

The *kubernetes publisher* image can be built using the commands

$ cd $HOME

# download code from repository
$ mv kubernetes-prometheus-publisher/ k8s-prometheus-publisher/

---

\(^{88}\) [https://hub.docker.com/](https://hub.docker.com/) /ubuntu
\(^{89}\) [https://github.com/5g-media/kubernetes-prometheus-publisher](https://github.com/5g-media/kubernetes-prometheus-publisher)
After the instantiation of the container, the supervisor service is responsible to initialize the monitoring worker. The worker wakes up every X seconds (it depends on the variables in the container instantiation) using a scheduler package, collects the metrics from the Prometheus API (part of the Kubernetes cluster) and publishes them in the Kafka bus. During the instantiation, a set of environmental variables are set:

- **KAFKA_IP**: the IPv4 of the Kafka bus,
- **KAFKA_PORT**: the port in which the Kafka bus listens to (default port is 9092),
- **KAFKA_KUBERNETES_TOPIC**: the topic of the Kafka bus in which the metrics are published,
- **PROMETHEUS_HOST**: the IPv4 of the Prometheus API,
- **PROMETHEUS_PORT**: the port in which the Prometheus API listens to,
- **PROMETHEUS_POLLING_STEP**: the interval step based on which the metrics are extracted from the Prometheus,
- **SCHEDULER_SECONDS**: how frequently the publisher feeds with monitoring metrics the Kafka bus

The OpenNebula publisher uses as base image the “ubuntu:16-04” image, available in Dockerhub repository and is available in the 5G-MEDIA gitlab repository. Its Dockerfile file looks like:

```Dockerfile
FROM ubuntu:16.04
MAINTAINER Athanasoulis Panagiotis
```

90 [http://supervisord.org](http://supervisord.org)
91 [https://github.com/dbader/schedule](https://github.com/dbader/schedule)
92 [https://prometheus.io/](https://prometheus.io/)
93 [https://hub.docker.com/_/ubuntu](https://hub.docker.com/_/ubuntu)
94 [https://github.com/5g-media/opennebula-kafka-publisher](https://github.com/5g-media/opennebula-kafka-publisher)
The **OpenNebula publisher** image can be built using the commands

```
$ cd HOME
# download code from repository
$ find ./opennebula-kafka-publisher -type d -exec sudo chmod -R 755 {} \
$ find ./opennebula-kafka-publisher -type f -exec sudo chmod 664 {} \
$ chmod a+x ./opennebula-kafka-publisher/deployment/run.sh
$ cp ./opennebula-kafka-publisher/deployment/Dockerfile .
# build image
$ sudo docker build --no-cache -t opennebula-kafka-publisher .
$ source opennebula-kafka-publisher/deployment/clean.sh
```

while it can be instantiated as a docker container using the following command

```
$ sudo docker run -p 80:3333 --name opennebula-kafka-publisher --restart always \
  -e KAFKA_IP="217.172.11.173" \
  -e KAFKA_PORT="9092" \
  -e KAFKA_OPENNEBULA_TOPIC="nfvi.tid-onlife.opennebula" \
  -e KAFKA_TRAFFIC_MANAGER_TOPIC="trafficmanager.uc2.metrics" \
  -e OSM_IP="217.172.11.185"
```
After the instantiation of the container, the supervisor service\(^\text{95}\) is responsible to start the monitoring worker. The worker wakes up frequently (it depends on the variables in the container instantiation) using a scheduler package\(^\text{96}\), collects the metrics from the OpenNebula XML-RPC API\(^\text{97}\) and publishes them in the Kafka bus. During the instantiation, a set of environmental variables are set:

- KAFKA_IP: the IPv4 of the Kafka bus,
- KAFKA_PORT: the port in which the Kafka bus listens to (default port is 9092),
- KAFKA_OPENNEBULA_TOPIC: the topic of the Kafka bus in which the metrics are published,
- OSM_IP: the IPv4 of the OSM,
- OSM_USER: the username of the OSM user with administrative permissions,
- OSM_PWD: the password of the OSM user with administrative permissions,
- XML_RPC_SERVER: the IPv4 of the OpenNebula XML-RPC API,
- XML_RPC_SESSION: the credentials that is used for the creation of a session in the XML-RPC API,
- SCHEDULER_MINUTES: the granularity with which the publisher feeds with monitoring metrics the kafka bus, and
- NO_OSM_VM_IDS: The identifiers of the VMs that have not been deployed via OSM but there is need for monitoring metrics extraction.

About the **application level monitoring**, different agents have been implemented/integrated and packaged within the single VNFs in order to export application-related metrics, all based on *Influxdata Telegraf*\(^\text{98}\). Such agents offer a wide set of integrated plugins for capturing metrics and events from server applications, sensors, databases and systems.

As reported in D3.3 (Section 3.1.3.), *Telegraf* offers a wide set of native input and output plugins for collecting, processing and then exporting metrics and events. The plugins’ configuration is applied through the Telegraf configuration file, where different drivers can be activated and configured according to the application to be monitored and how this application is capable of exposing metrics.

The *Telegraf* agent is used in UC3’s VNFs and PNFs for realizing the application level monitoring. In particular, Telegraf is integrated and packaged in the “vCache” VNF and in the “UHD Streaming Server” PNF.

---

\(^{95}\) [http://supervisord.org](http://supervisord.org)

\(^{96}\) [https://github.com/dbader/schedule](https://github.com/dbader/schedule)

\(^{97}\) [http://docs.opennebula.org/5.8/integration/system_interfaces/api.html](http://docs.opennebula.org/5.8/integration/system_interfaces/api.html)

\(^{98}\) [https://www.influxdata.com/time-series-platform/telegraf/](https://www.influxdata.com/time-series-platform/telegraf/)
In the specific case of the vCache, Telegraf has been configured and packaged within the VNF image to be instantiated on demand. The baseline software of the “vCache”, Apache Traffic Server, exposes application metrics via REST APIs or CLI (e.g. cache hits, throughput, etc.), then the Telegraf “inputs.exec” plugin has been configured for getting metrics of interest for the optimization of the service; these metrics are then exported on a per service topic into the 5G-MEDIA infrastructure bus via Telegraf “outputs.kafka” plugin.

- Kafka-client "kafka-node"\(^{99}\): The native Kafka publish-subscribe client for Node.js offers the possibility to connect to a Kafka broker (Kafka-server) and start a Kafka producer (publish) and Kafka consumer (subscribe). Producers and Consumers can handle data or data-streams as messages which are pushed (publish) to or received (subscribe) from a Kafka topic.

In Node.js environments Kafka-node can be installed, added and initiated within an application with a constructor and connection-options. The minimal connection-parameter is Kafka-host (default: localhost:9092). Other parameters configure authentication (SASL), secure connection via SSL, and connection setup behaviour. With the established Connection between the Kafka-client and the Kafka-server, the applications can start Kafka-producers and Kafka-consumers. Each Kafka consumer should use its own Kafka-client.

A Kafka producer can send (publish) messages to a Kafka topic as a single message (data-payload) or data-stream (continuously data-payload). An important option for the producer is the Kafka topic. The Kafka node client can create a Kafka topic. Kafka producers are used in vCompression and vDetection VNFs. The vCompression VNF publishes metrics such as current cpu load, current memory usage, current frames per second (fps), and current bitrate to the Kafka-bus. The vDetection VNF publishes metadata from the face-detection or face-recognition. The metadata include detections (position and size) with the confidence-values (0% - 100%, 100% as best) and in face-recognition-mode additionally labels (possible name of the recognition) and distance-values (0 - infinite, 0 as best).

A Kafka consumer can receive (subscribe to) messages from a Kafka-topic as a single message (data-payload) or data-stream (continuously data-payload). An important option for the consumer is the Kafka topic which should be known. A Kafka consumer is used in vCompression. The vCompression subscribes to a Kafka topic and receives messages from the Kafka bus with parameters such as the bitrate-value. A message with a bitrate as payload will trigger a reconfiguration of the vCompression VNF.

In the specific cases of vCompression and vDetection VNFs, the Kafka producers and consumers have been configured and packaged within the VNF. The Node.js package "Kafka-node" is a reasonable choice for applications based on Node.js and JavaScript as FaaS in the OpenWhisk platform, as its functional logic (action) can be dynamically scheduled or triggered to run in associated events from external sources, e.g. the Kafka consumer is triggered by specific messages from a Kafka topic which can be published by other Kafka producers.

---

\(^{99}\) https://github.com/SOHU-Co/kafka-node
Conclusions

This deliverable describes the packaging tools included into the CI/CD services used by the SDK to allow the developers to have their VNF images ready abstracting over the packaging details and focusing only on the features they need. The document provides a comprehensive overview of the different packaging options for the developers varying from lightweight virtualization formats such as unikernels and containers up to most commonly used ISO images. For each of this option, the deliverable discusses the VNF lifecycle i.e., validation, onboarding, emulation and testing, and monitoring.
References

[5G-MEDIA use cases] http://www.5gmedia.eu/use-cases/


[Unikernels], Brad Whitehead, 14th Annual IEEE/ACM IT Professional Conference at TCF (2019)