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


Lead partner: IBM
Authors: David Breitgand [IBM], Avi Weit [IBM], David Griffin [UCL], Rizou Stamatia [SiLO], Francesco Iadanza [ENG], Gino Garozzo [NXW], Francesca Moscatelli [NXW]

Delivery date (DoA): 31/08/2018
Actual delivery date: 31/08/2018
Dissemination level: Public
Version number: 1.0  Status: Final

Grant Agreement N°: 761699
Project Acronym: 5G-MEDIA
Project Title: Programmable edge-to-cloud virtualisation 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: 1 June, 2017
Duration: 30 months

Project co-funded by the European Commission under the Horizon 2020 Programme.
### Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Who</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Aug 30</td>
<td>David Breitgand [IBM]</td>
<td>• Submission ready</td>
</tr>
<tr>
<td>0.9</td>
<td>Aug 29</td>
<td>David Breitgand, Avi Weit [IBM]</td>
<td>• The draft incorporating SILO and UCL reviews</td>
</tr>
<tr>
<td>0.8</td>
<td>Aug 20</td>
<td>David Breitgand</td>
<td>• The draft incorporating the NXW and IRT reviews</td>
</tr>
</tbody>
</table>
| 0.7      | Aug 7    | David Breitgand, Avi Weit [IBM] | • Second complete draft (except Section 3.5 and Section 4.6) – contributions required from ENG and UCL  
|          |          |                              | • Release for interim review by 5G-MEDIA consortium members                |
| 0.6      | Aug 6    | David Breitgand, Avi Weit [IBM] | • Second complete draft (except Section 3.5 and Section 4.6) – contributions required from ENG and UCL  
|          |          |                              | • Internal review by IBM                                                  |
| 0.5      | Aug 2    | David Breitgand, Avi Weit [IBM] | First complete draft (except Section 3.5 and Section 4.6) – contributions required from ENG and UCL |
| 0.4      | Jul 25   | David Breitgand, Avi Weit [IBM] |                                                                            |
| 0.3      | Jul 15, 2018 | David Breitgand, Avi Weit [IBM] | Third Draft                                                               |
| 0.2      | Jul 10, 2018 | David Breitgand [IBM]         | Second Draft                                                              |
| 0.1      | Jul 01, 2018 | David Breitgand [IBM]         | First Version                                                             |
### 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>09/08/2018</td>
<td>Madeleine Keltsch, Gordana Macher [IRT]</td>
<td></td>
</tr>
<tr>
<td>Reviewer</td>
<td>07/08/2018</td>
<td>Gino Garozzo [NXW]</td>
<td></td>
</tr>
<tr>
<td>Reviewer</td>
<td>21/08/2018</td>
<td>Francesco Iadanza [ENG]</td>
<td></td>
</tr>
<tr>
<td>Reviewer</td>
<td>27/08/2018</td>
<td>Stamatia Rizou [SiLO]</td>
<td></td>
</tr>
<tr>
<td>Reviewer</td>
<td>28/08/2018</td>
<td>David Griffin [UCL]</td>
<td></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 consists of the following organisations:

<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 – INEGNERIA 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>HELLINIC 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 GMB</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** ........................................................................................................... 11

**1. INTRODUCTION** .................................................................................................................. 13

**2. RELATED OPEN SOURCE PROJECTS** .................................................................................. 17
    2.1. **APACHE OPENWHISK** .................................................................................................................. 17
    2.2. **CONTAINER ORCHESTRATOR ENGINE (COE)** ............................................................................. 20
        2.2.1. **Background** .................................................................................................................................... 21
        2.2.2. **Kubernetes** .................................................................................................................................... 23
    2.3. **MONITORING FRAMEWORK** ........................................................................................................ 25
        2.3.1. **cAdvisor** ....................................................................................................................................... 25
        2.3.2. **Prometheus** ................................................................................................................................... 25
    2.4. **LOGGING SUPPORT WITH ELK STACK ON K8S** ......................................................................... 26

**3. FaaS ARCHITECTURE IN 5G-MEDIA** .................................................................................... 29
    3.1. **SINGLE FaaS VIM** ......................................................................................................................... 33
    3.1.1. **All-in-one FaaS VIM Deployment** ............................................................................................... 33
    3.1.2. **FaaS VIM Production Deployment** .............................................................................................. 36
    3.2. **FEDERATED FaaS ARCHITECTURE** ............................................................................................. 38
    3.3. **MULTIPLE FaaS VIMs AND VNF CHAINING ACROSS FaaS AND NON-FaaS VIMs** ................. 38
    3.4. **ETSI FLOWS** .................................................................................................................................. 39
    3.5. **PRE-ONBOARDING** ....................................................................................................................... 40
    3.6. **MONITORING** ............................................................................................................................... 41
    3.7. **INTEGRATION WITH MAPE** .......................................................................................................... 43
    3.8. **MULTITENANCY** ............................................................................................................................. 44

**4. REFERENCE IMPLEMENTATION OF THE FaaS ARCHITECTURE** ........................................... 46
    4.1. **FaaS VNF DAY 0 CONFIGURATION SERVICE** ............................................................................ 46
    4.2. **FaaS VNF DAY 1 CONFIGURATION SERVICE** ............................................................................. 47
    4.3. **OPENWHISK TO K8S ACTIONS OFFLOADING SERVICE** ............................................................ 48
    4.4. **OPENWHISK SUPPORT FOR GPUs AND AFFINITY/ANTI-AFFINITY ON K8S** ..................... 48
        4.4.1. **5G-MEDIA Docker Base Image with NVIDIA GPU Support** .................................................. 48
        4.4.2. **Placement Annotations** .......................................................................................................... 49
    4.5. **OPENWHISK SUPPORT FOR PROJECT DEPLOYMENT AND MIGRATION** ............................. 52

**ANNEX A: INSTALLATION AND GETTING STARTED** ................................................................. 53
    4.6. **KUBERNETES CLUSTER** ................................................................................................................. 53
    4.7. **OFFLOAD-SERVICE** ...................................................................................................................... 53
List of Figures

Figure 1: Programming Models Spectrum .................................................................13
Figure 2: 5G-MEDIA Architecture ..............................................................................16
Figure 3: Apache OpenWhisk Architecture ...............................................................18
Figure 4: Lean OpenWhisk Architecture ...................................................................20
Figure 5: Kubernetes (K8s) Architecture .....................................................................23
Figure 6: Prometheus on K8s Architecture (source: https://prometheus.io/docs/introduction/overview/) .........................................................26
Figure 7: ELK Stack Schematics (source: https://hackernoon.com/deployment-of-full-scale-elk-stack-to-kubernetes-6f38f6c57c55) .................................................................27
Figure 8: Integration of the ELK Stack for FaaS VIM in 5G-MEDIA .........................28
Figure 9: ETSI MANO Compatible Reference Architecture of a FaaS VIM ................29
Figure 10: Reference Implementation of FaaS VIM .......................................................30
Figure 11: High Level Architecture of FaaS (OpenWhisk)/COE (K8s) Integration ....31
Figure 12: All-in-One OSM/FaaS VIM Deployment ...................................................34
Figure 13: Basic FaaS VNF Instantiation Flow ............................................................36
Figure 14: FaaS VIM Deployment on Open Stack .......................................................37
Figure 15: Generalization of Production Deployment to Multiple VIMs and Data Centres ....38
Figure 16: High Level Monitoring Architecture ..........................................................41
Figure 17: Example Grafana Dashboard with Per-Action Resource Consumption ....42
Figure 18: OSM R4 Monitoring Architecture (source: https://osm.etsi.org/wikipub/index.php/OSM_Performance_Management) ......................................................43
Figure 19: Day O FaaS Configuration Service ............................................................46
List of Tables

Table 1: CPU utilization of OpenWhisk actions on K8s ................................................................. 42
Table 2: Dockerfile of base image for blackbox actions exploiting NVIDIA GPU .................... 49
## Definitions and acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>Authentication, Authorization, Accounting</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>BM</td>
<td>Bare Metal Machine</td>
</tr>
<tr>
<td>cAdvisor</td>
<td>Container Advisor</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>CNO</td>
<td>Cognitive Network Optimizer</td>
</tr>
<tr>
<td>COE</td>
<td>Container Orchestration Engine</td>
</tr>
<tr>
<td>CPU:</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CUDA:</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DevOps</td>
<td>Development and Operations</td>
</tr>
<tr>
<td>DNS:</td>
<td>Domain Naming System</td>
</tr>
<tr>
<td>ELK</td>
<td>Elastic Search, Logstash, Kibana</td>
</tr>
<tr>
<td>FaaS</td>
<td>Function-as-a-Service</td>
</tr>
<tr>
<td>Go</td>
<td>Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure-as-a-Service</td>
</tr>
<tr>
<td>IAM</td>
<td>Identity and Access Management</td>
</tr>
<tr>
<td>ICP</td>
<td>IBM Cloud Private</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>JSON</td>
<td>Java Script Object Notation</td>
</tr>
<tr>
<td>K8s</td>
<td>Kubernetes</td>
</tr>
<tr>
<td>LXC</td>
<td>Linux Containers</td>
</tr>
<tr>
<td>MAPE</td>
<td>Monitor, Analyse, Plan, Execute</td>
</tr>
<tr>
<td>NFVI:</td>
<td>Network Functions Virtualization Infrastructure</td>
</tr>
<tr>
<td>NS</td>
<td>Network Service</td>
</tr>
</tbody>
</table>
NSD  Network Service Descriptor
NVIDIA:  A leader in visual computing technology and the inventor of GPU
OSM  Open Source Mano
PaaS  Platform-as-a-Service
QoE:  Quality of Experience
QoS:  Quality of Service
RBAC  Role-Based Access Control
REST:  Representational State Transfer
RO:  Resource Orchestrator
SaaS  Software-as-a-Service
SDK  Software Development Kit
SDN  Software Defined Network
SSH:  Secure Shell
SVP  Service Virtualization Platform
TSDB  Time Series Data Base
UC  Use Case
URL:  Uniform Resource Locator
vCPU:  Virtual CPU
VDU:  Virtual Deployment Unit
VIM  Virtualized Infrastructure Manager
VM  Virtual Machine
VNF  Virtual Network Function
VNFD  Virtual Network Function Descriptor
VNFR  Virtual Network Function Record
YAML  YAML Ain't Markup Language
Executive summary

A cloud-native transformation is happening in the Network Functions Virtualization (NFV) orchestration field and its impact on the telecom ecosystem is expected to be profound\(^1\). At the core of this transformation, there is a micro-services architecture with container (such as Docker) and container orchestrator (such as Kubernetes) technologies powering up the microservices approach.

The 5G-MEDIA project not only aims at leveraging a classic cloud-native approach, but attempts to make a next big step beyond the current state of art by pioneering Functions-as-a-Service (FaaS), also known as “serverless”, for VNF orchestration for media intensive applications in 5G networks\(^2,3\). This first of a kind approach is expected to develop a platform that will (a) reduce the total cost of ownership for different players in the NFV ecosystem in the 5G media industry, (b) speed up time to market, (c) simplify developer’s experience, and (d) simplify platform operation. Taken together, these factors will provide significant competitive advantages to the platform users and operators.

This document describes a first release of the FaaS framework in the 5G-MEDIA platform. The currently implemented features include:

- Integration with OSM R3 by implementing a novel FaaS VIM Plugin;
- Support for GPU;
- Support for advanced placement policies;
- Support for network communication (not offered by FaaS frameworks out of the box);
- Support for monitoring and powerful visualization using dashboards;
- Support for chaining FaaS and non-FaaS VNFs within a single data centre;
- Cloud and virtualization technology independence;
- Day 0, 1 configuration services.
- Low footprint all-in-one environment (leveraged in SDK, see D5.1: “5G-MEDIA Programming Tools”)

We also briefly discuss the architecture blueprints and outline challenges for:

- Next generation monitoring (with the advent of OSM R4);
- Application logging;
- Federated FaaS architecture;
- Integration of FaaS VIM with Cognitive Network Optimizer (CNO) and Monitor, Analyse, Plan, Execute (MAPE) management loop;
- Chaining of VNFs across different data centres;
- Multitenancy.

\(^3\)[https://zenodo.org/record/1299197#.W2gAWVUzYcw]
The first release prototype uses the following best-of-breed technologies: Docker (container technology), Kubernetes (container orchestration), Prometheus (monitoring), Grafana (visualization), ELK stack (log analysis), Apache OpenWhisk (FaaS framework).

Some important results obtained during our work on the first release have been contributed to the open source community with some of them being already merged and other being in the process of merging\(^4,5,6,7\). Furthermore, some of the important merged contributions, such as wskdeploy export feature were introduced to the community (and accepted by it) with an explicit motivation of the needs of the 5G-MEDIA project\(^8\).

The rest of this deliverable is organized as follows. Section 1 introduces the FaaS concepts and explicitly outlines the design tenets for integrating FaaS with 5G-MEDIA. Section 2 reviews the relevant technological and exploitation aspects of the open source projects that have been selected for the first release prototype implementation. Section 3 details the reference architecture of the FaaS framework in the 5G-MEDIA platform. Section 4 discusses the reference implementation of the architecture presented in Section 3. Annex A covers the installation and provides a getting started tutorial. Section 5 concludes and provides the outlook into the next release.

\(^4\) https://github.com/apache/incubator-openwhisk/pull/2984
\(^5\) https://github.com/apache/incubator-openwhisk/pull/3525
\(^6\) https://github.com/apache/incubator-openwhisk/pull/3886
\(^7\) https://medium.com/openwhisk/lean-openwhisk-open-source-faas-for-edge-computing-fb823c6bbb9b
\(^8\) https://github.com/apache/incubator-openwhisk-wskdeploy/blob/master/docs/export.md
1. Introduction

The serverless programming model, also known as Function-as-a-Service (FaaS) has gained considerable momentum since its introduction in 2014\(^9\). The term serverless is somewhat misleading. The term does not imply that no servers are involved in running an application. Rather it hints at a level of abstraction, which allows to abstract the servers and focus exclusively on the application code. Figure 1 positions FaaS on the spectrum of programming models. FaaS can be viewed as a specialized Platform-as-a-Service (PaaS) taking care of all deployment and run-time issues and relieving the developer from any concerns related to server provisioning and maintenance.

![Figure 1: Programming Models Spectrum](image)

There are several main principles pertaining to FaaS:

- A unit of execution is a function written in a high-level programming language;
- A function is executed in response to an event (which also can be an HTTP call);
- Rules and triggers can be defined to bind functions and events together, so FaaS is an intrinsically event driven programming model;
- A customer is charged only for the resources used during the function execution (at a very fine granularity: typically, being 100 ms);
- Functions are transparently auto-scaled and instantaneously elastic: the load balancer is built into a platform and new functions are started in response to events as needed (within some system limits, such as maximum number of simultaneous invocations per user and invocations/sec per user);
- Functions are ephemeral (i.e., stateless), even though FaaS extensions, such as AWS Step Functions\(^{10}\), IBM Composer\(^ {11}\), and Azure Durable Functions\(^ {12}\) support automated functions state checkpointing;

---

\(^9\) “Amazon introduces Lambda, Containers at AWS re:Invent - SD Times”, SD Times 2014-11-14

\(^ {10}\) [https://aws.amazon.com/step-functions/](https://aws.amazon.com/step-functions/)


Functions can be chained with the output of one action being the input of another;
Functions can be orchestrated to execute in complex application topologies;
There is no server administration or provisioning.

A simple FaaS development cycle is as follows. A developer develops a function using her favorite text editor. Next a function is created (i.e., registered in the platform database) using a CLI or a Web based GUI. After being created, a function receives a name, by which it can be bound to rules and triggers and get invoked in response to events represented by the triggers. The reader is referred to IBM Cloud Functions tutorial\textsuperscript{13} and Apache OpenWhisk community resources\textsuperscript{14} for detailed step by step examples of serverless programming with OpenWhisk.

The FaaS programming model offers several advantages:

- Increased productivity: developers focus only on the application code;
- Increased cost-efficiency: if an application workload is inconsistent and time utilization of the workload is below a certain point (dependent on a charging scheme), on-demand functions invocation is more cost-efficient than continuous provisioning of resources (e.g. virtual servers);
- Ease of service mashup and extensibility: functions are a very convenient mechanism for gluing various services together fast and extending the service fabric with the new services.

In the 5G-MEDIA platform, we offer a developer the benefits of the FaaS programming model in a way which is compatible with ETSI MANO and without vendor lock-in. A specific FaaS framework, Apache OpenWhisk, used for the reference implementation can be easily replaced by other FaaS frameworks in the future.

5G-MEDIA developers can use FaaS to develop VNFs that will be invoked on-demand at run time (with low invocation latency), triggered by application-level or platform-level events, and for which providing and maintaining resources continuously would be inefficient. We have identified scenarios in each of the three project use cases that exemplify how FaaS can be applied for the efficient and dynamic instantiation of VNFs implementing media application features.

The requirements underpinning the FaaS framework integration with the 5G-MEDIA platform can be summarized as follows:

- FaaS VNF lifecycle management (on-boarding, instantiation, monitoring, and deletion) should be fully aligned with that of the non-FaaS VNFs and facilitated through the same MANO stack;
- Since FaaS frameworks rapidly proliferate, the architecture should provide for extensibility, i.e., it should be easy to incorporate new FaaS frameworks;

\textsuperscript{13} IBM Cloud Functions Overview, \url{https://console.bluemix.net/docs/openwhisk/index.html#index}

\textsuperscript{14} Apache OpenWhisk Community Resources, \url{https://github.com/apache/incubator-openwhisk-external-resources}
- FaaS VNFs should be discoverable and accessible over the public network similarly to non-FaaS VNFs;
- FaaS VNFs should be able to access services over the public network (including other VNFs);
- FaaS and non-FaaS VNFs can be chained (both in a single DC and across DCs);
- FaaS VNFs should be able to exploit special hardware, such as GPUs;
- It should be possible to influence FaaS placement through collocation and anti-collocation constraints;
- FaaS framework deployment and operation should be independent of the underlying cloud virtualization technology (i.e., it should be possible to deploy it on OpenStack, OpenNebula, VMWare, IBM Cloud, AWS, etc.);
- Minimal changes to the ETSI MANO workflows.

Figure 2 captures the overall 5G-MEDIA architecture that implements the requirements above. FaaS integration with the rest of the platform is realized through a plugin abstraction. In this approach, FaaS is regarded as an alternative type of resources virtualization technology that operates at the level of a function rather than that of a VM or a container. The rationale for this is that even though the physical resources are hidden from the developers, in any FaaS programming model implementation, a pool of resources (i.e., bare metal servers or VMs) exists to execute the functions, e.g., in the form of containers that are being transparently created in response to action invocation requests. A function can, therefore, be regarded as a form of resource virtualization and a Virtualized Infrastructure Manager (VIM) can be developed to manage this abstraction. We term this specialized VIM, a FaaS VIM. This approach provides for an elegant and seamless integration of the cloud-native serverless programming model with the more traditional VM based one within the same platform.

Although not explicitly shown in Figure 2, FaaS is integrated with the 5G-MEDIA SDK to enable FaaS VNF development, validation, emulation, profiling, and deployment to the Service Virtualization Platform (SVP). See deliverable D5.1 “5G-MEDIA Programming Tools”.
Figure 2: 5G-MEDIA Architecture
2. Related Open Source Projects

In this section we describe the open source projects, which are relevant for our FaaS integration architecture. We describe Apache OpenWhisk in D2.3: “5G-MEDIA Platform Architecture”. In this document we avoid repetition and provide only a bare minimum review of OpenWhisk to ensure that this deliverable is self-contained.

2.1. Apache OpenWhisk

Apache OpenWhisk is an extensible serverless computing platform that supports functions (also known as “actions”) that can be written in multiple programming languages including Node.js, Python, Swift, Java, and PHP. Also, OpenWhisk supports native binaries. With a native binary, any executable that is compatible with a standard OpenWhisk container may run as a serverless function. These functions are termed blackbox actions. Blackbox actions derive their container image from the base OpenWhisk container that includes some basic management services allowing the OpenWhisk framework to interact with the action.

In OpenWhisk, functions run within managed Docker containers. The platform’s architecture supports Docker images for each language runtime. Using this approach, IBM Cloud Functions, which is powered by OpenWhisk, offers Node v8 with SDKs that is specific to the IBM Cloud portfolio, including Watson services and Cloud Object Storage and Python Docker container image with packages popular for numerical analysis and machine learning.

The polyglot nature of the OpenWhisk actions is extremely useful for action chaining (known as “composition”). This feature is especially useful for the 5G-MEDIA platform, where FaaS VNFs from the catalog can be chained together to form novel Network Services (NS).

As a result, VNF developers can use the most suitable language for the task while building their complex functions and the investment in development is preserved as the NS are being updated and further developed.

The core of the Apache OpenWhisk programming model is similar to other serverless frameworks. Apache OpenWhisk is an inherently event-driven framework. OpenWhisk allows to define events (“triggers”), and “rules” that connect the triggers to actions. Also, OpenWhisk supports sequential chaining of actions (“sequences”) with one action’s output being another action’s input. OpenWhisk Composer allows to combine actions into arbitrary complex topologies with a stateful flow of execution, which is automatically supported by the compositions.

---

15 https://openwhisk.apache.org/
16 https://medium.com/openwhisk/composing-functions-into-applications-70d3200d0fac
Figure 3 captures the main components of the Apache OpenWhisk architecture. At the core of the OpenWhisk project are two components: Controller and Invoker. The Controller performs load balancing among the invokers to distribute load evenly and to maximize the chances of hitting an Invoker node having a pre-warmed container for the action. The Controller publishes action invocation requests through the Kafka topics, to which Invokers subscribe.

One of the following situations might occur when an action invocation request hits an Invoker node:

- There is no container to run an action and a new container should be created (this is termed a “cold start”. A cold start might incur an invocation latency of 500-700ms).
- A pre-warmed non-initialized container exists. A pre-warmed non-initialized container is a container for a given run-time that has not yet been initialized with the specific action code. The base image of OpenWhisk Docker containers includes an HTTP service that serves two resources: /init and /run. The Invoker performs a REST call on the /init resource to “inject” the action code into the right place in the file system of the base image. Then the Invoker performs an HTTP POST request to the /run resource of the Docker container and the action starts executing. Initialization of a pre-warmed container is very fast and adds only a few tens of milliseconds.
- A pre-warmed initialized container exists for the action. This happens when a previous invocation of the action terminates. OpenWhisk does not terminate the container in
anticipation that another invocation of the same action from the same user may arrive. If this indeed happens, the action invocation is almost instantaneous.

OpenWhisk is a multi-tenant system. It supports user isolation at the level of name spaces and requires an API key to interact with resources. OpenWhisk can be deployed on Kubernetes\(^\text{17}\). This makes OpenWhisk independent of the underlying cloud technology and achieves vendor neutrality. Deployment on Kubernetes allows to leverage:

- Support for networking;
- Actions discoverability as services;
- Placement policies;
- Virtualization technology neutrality

OpenWhisk allows programming artefacts to be grouped into packages, which can be shared. Furthermore, OpenWhisk development tools, such as wskdeploy, allow projects to be created that define multiple artefacts using a manifest and deploy and manage projects as units rather than dealing with multiple independent artefacts.

The project has developed a version of Apache OpenWhisk with a small footprint that we term Lean OpenWhisk\(^\text{18}\). Lean OpenWhisk is used as part of the 5G-MEDIA SDK as described in D5.1: “5G-MEDIA Programming Tools”. 5G-MEDIA has proposed a pull request to the Apache OpenWhisk community\(^\text{19}\). The pull request is under review and Lean OpenWhisk is expected to be merged with the core of the Apache OpenWhisk project. Figure 4 depicts the Lean OpenWhisk architecture.

\(^{17}\) [https://medium.com/openwhisk/deploying-openwhisk-on-kubernetes-3f55f781fbab](https://medium.com/openwhisk/deploying-openwhisk-on-kubernetes-3f55f781fbab)


\(^{19}\) [https://github.com/apache/incubator-openwhisk/pull/3886](https://github.com/apache/incubator-openwhisk/pull/3886)
The project is developing a federated OpenWhisk architecture, in which multiple independent OpenWhisk instances can be installed in the Internet (e.g., in the central Cloud and at the Edge) and OpenWhisk projects can be developed at a central designated location and deployed across the OpenWhisk instances, so that the artefacts will be kept synchronized according to an eventual consistency model, despite intermittent connectivity and failures. A prototype of the federated OpenWhisk architecture has been developed internally at IBM as part of its work on the Watson IoT Platform. IBM will adapt and integrate this work with the 5G-MEDIA architecture as described in D2.3: “5G-MEDIA Platform Architecture”. The federated architecture is essential in the second year of the project for all three use cases, allowing them to leverage FaaS in a distributed fashion.

2.2. Container Orchestrator Engine (COE)

Similar to virtual machines (VMs), containers provide isolation, except this is done by the OS and at the process level. Each container is a process or group of processes that run in isolation. Typical containers explicitly run only a single process. Although containers have been known about for many years, they only became a mainstream virtualization technology with the advent of Docker, which provided a convenient API for packaging an application once and running it everywhere. “Containerization” refers to packaging an application with all its dependencies, so that it can be deployed in different environments and pulling the necessary changes in an incremental fashion as the application undergoes changes. This is the “build once, run everywhere” mode of operation that is being promoted by Docker.

In practice, it is rarely the case that containers are deployed in a stand-alone manner. An application might need many containers and they should be grouped in a meaningful way and operated as a single entity. Furthermore, containers require many resources to run such as...
volumes, networks, and secrets that are required to connect to databases, interact with back ends, and with other applications. Furthermore, containers of the applications might need to be discoverable from the outside, replication for high availability might be required, scheduling of resources in a cluster is another task that should be efficiently performed. Scheduling placement policies that expose specific resources, such as GPUs might be needed, as well as placement restrictions enforcement, such as node and container affinity and anti-affinity. The Container Orchestrator Engine (COE) is tasked with these challenges, relieving application developers and operators from this work and, therefore, allowing to deploy and operate applications at very large scale.

There exist several COEs. The most popular are Mesos, Swarm, and Kubernetes (K8s) with the latter becoming a de-facto COE standard.

In 5G-MEDIA we use K8s to achieve virtualization technology neutrality for the FaaS framework as explained in Section 3. In the following subsection we provide a background and highlight the architecture and the main features of K8s.

### 2.2.1. Background

Kubernetes is a cluster management system that aims at optimizing the utilization of the cluster resources, such as CPU, memory, storage, and network through efficiently scheduling containers across managed nodes. K8s can run anywhere either on bare metal or in any cloud provider infrastructure. K8s is cloud agnostic and focuses on deploying and scheduling containers inside the infrastructure instead of directly utilizing host (referred as “nodes”).

The main features of the K8s platform include:

- Support for a POD abstraction for grouping containers: a POD is the smallest unit that can be deployed and managed by Kubernetes. It is a group of containers that share the same network space (i.e. same IP address and port space). A POD is assigned a network interface (eth0) and is the unit of scheduling in K8s;
- Support for declarative “desired state” of a deployment;
- Support for self-healing towards the desired state;
- Auto-scaling and load balancing;
- Exposition of a POD as a service accessible from the outside;
- Support for overlay container networking within the same POD and among PODs running on the same and different hosts;
- DNS management;
- Detailed resources monitoring and logging;
- Rolling updates with rollback.

Some facts that make K8s particularly important in the industry are as follows.
• It was developed by Google based on its Borg scheduler\textsuperscript{20}. Hence Google brings their expertise, which includes more than ten years of running containerized services at large scale. This is Google’s third-generation platform with many core contributors to the first platform also being contributors to Kubernetes;

• K8s was donated to CNCF (run by the Linux Foundation) with many companies now actively contributing to K8s development and Google accounting to less than half of the contributions;

• Multiple leading cloud vendors, including IBM, Google, and Microsoft, use K8s for their managed container service cloud offerings at large scale;

• Built using best-of-breed open source components (e.g., etcd by CoreOS\textsuperscript{21})

• K8s natively works with Docker; however, it is not dependent on Docker. For example, it will add support for the competing rkt container format\textsuperscript{22};

• Support for competing Swarm COE is considerably down;

• Another viable alternative, Mesos, is in its early adoption stage.

In light of the above, selecting K8s as COE for 5G-MEDIA FaaS platform will maximize the chances of project results sustainability throughout the project duration and beyond. Selecting K8s as COE also strengthens the IBM’s exploitation plan. IBM’s commercial offering called IBM Cloud Private (ICP), is based on K8s, which allows a strong alignment of the results of the 5G-MEDIA project with the IBM roadmap for ICP. Recently, IBM entered a strategic partnership with RedHat on accelerating each other’s flagship K8s based offerings and aligning between the IBM ICP the RedHat OpenShift\textsuperscript{23}.

\textsuperscript{20} https://ai.google/research/pubs/pub43438
\textsuperscript{21} https://coreos.com/etcd/
\textsuperscript{22} https://coreos.com/rkt/
\textsuperscript{23} http://www.nihaobar.com/2018/05/08/ibm-red-hat-expand-partnership-cloud/
2.2.2. Kubernetes

Figure 5: Kubernetes (K8s) Architecture

Figure 5 depicts the main components of the K8s architecture. We will only provide a brief overview of the features, which are intimately related to the FaaS framework architecture described in the rest of this document. Detailed information on K8s can be found at the K8s official page\textsuperscript{24}.

K8s POD is a collection of containers. This is a deployment unit in K8s that is assigned a single IP address. Inside the POD, a special Pause container handles networking by managing a network namespace, IP address and ports. The IP address is used by all containers in the POD. As explained in Section 3, we implement VNFs as PODs comprising three containers (the container running the VNF itself, the Day 1 configuration service, and a “caller” container that orchestrates the VNF running within a POD.

A K8s cluster comprises a master node and a set of worker nodes (referred simply as nodes). The K8s master includes:

- **API server**: users interact with the API server using REST calls or kubectl CLI. The API server is contacted for every operation related to K8s “resources” (which are, in fact, HTTP objects), such as PODs, name spaces, services, etc. For example, to create a POD, a user first defines a yaml file manifest describing the POD and interacts with the API server via either kubectl or REST to communicate this definition to K8s master. The API server stores the desired state of the cluster (i.e., all the declarative definitions) in

\textsuperscript{24} \url{https://kubernetes.io/docs/home/?path=users&persona=app-developer&level=foundational}
etcd\textsuperscript{25}. Our Offload Service briefly referred to in Section 3 and described in more detail in Subsection 4.3, interacts with the API server to define OpenWhisk actions as K8s PODs and request their execution on K8s.

- **Scheduler**: the scheduler’s responsibility is to place PODs (created via API server by users as per yaml manifest definition) to available nodes based on the resources requirements and placement policies specified in the yaml definition of the pod. The yaml definition, may, for example, require that the scheduler places PODs on the nodes if the labels of the PODs match those of the nodes. Also, the POD scheduling definitions might include policies related to affinity and anti-affinity of PODs. For example, a scheduler might be required not to place a POD on a node, where another POD with the same label is already running. We use nodes and POD labeling mechanisms to properly schedule VNFs (implemented as OpenWhisk actions offloaded to K8s) on the nodes with GPUs and to ensure VNF anti-affinity as required by UC1.

- **Controller Manager**: performs operations on the resources based on the cluster state and executes necessary operations to bring the current state of a containerized application to the declared desired state (the desired state is described in the yaml manifest). The Controller Manager comprises multiple specialized controllers. For example, there is a Node Controller that monitors availability of the nodes, a Replication Controller that ensures that there is always the specified number of PODs replicas running, etc. Also, the Controller Manager can be extended with Custom Controllers.

- **etcd**: all configuration information about the K8s cluster desired state is stored in the etcd reliable key/value store. These key/value pairs contain all the data about the nodes comprising the cluster and the PODs that should be run.

- **Add-ons**: there are multiple add-ons, such as Cluster DNS add-on, RBAC for role-based access, etc.

A K8s node comprises:

- **Docker**: each node runs a Docker Daemon. The Docker Daemon pulls container images as needed from some configured repository (e.g., Docker Hub), if a POD that is scheduled to run on a node that does not have a container(s) image(s) locally.

- **Kubelet**: a kubelet is an agent that runs on every node linking it to the API server of the master. Kubelet checks the health of containers running in the PODs of a node periodically and triggers management actions via the API server if the current state of the node deviates from the desired one. Kubelet retrieves container metrics from \textit{cAdvisor} (Container Advisor): a native monitoring component of K8s, aggregates them and exposes the metrics through the Kubelet Summary API for external components, such as Prometheus\textsuperscript{26} to consume.

\textsuperscript{25} https://github.com/coreos/etcd
\textsuperscript{26} https://prometheus.io/
• **cAdvisor**: runs at every node. By default, it collects and keeps 2 minutes worth of container metrics for each POD running locally on a node. cAdvisor exposes a REST API through which metrics can be consumed\(^{27}\).

• **kube-proxy**: kube-proxy runs in each node and watches the API server for changes made to PODs and services definition to maintain the desired state. It makes PODs available as services and distributes load among the PODs. Kube-proxy uses iptable rules to forward requests to the relevant containers comprising a service. For example, a NodePort service can be accessed from any node, because kube-proxy sets up the iptables rules correctly to ensure cluster wide connectivity. In our reference implementation, we use NodePort service type to expose PODs (i.e., VNFs) as services.

• **Flannel**: Flannel is an overlay network providing container networking at the POD to POD level within a single cluster. It is explained in greater detail in Subsections 3.1.1 and 3.1.2.

### 2.3. Monitoring Framework

In this section we describe the building blocks of the monitoring infrastructure of K8s that we use in the FaaS VIM integration with the 5G-MEDIA platform (see Section 3.6).

#### 2.3.1. cAdvisor

cAdvisor provides metrics on resource usage and performance of containers run in K8s. cAdvisor is a daemon that runs on every K8s node and collects, aggregates, processes, and exports data about running containers for all PODs running on the node. cAdvisor natively supports Docker and has support for other types of containers, e.g., LXC, aiming at supporting any container technology across the board. cAdvisor is a host level technology, not a cluster level technology. While cAdvisor aggregates statistics per container groups and at the machine level, it might prove to be a too low-level interface to monitor a large cluster and too much work might be required to get meaningful insights from the raw metrics provided by cAdvisor. Also, it does not store data for long periods of time. For these reasons, cAdvisor is usually used in conjunction with higher level monitoring frameworks, such as Prometheus.

#### 2.3.2. Prometheus

Prometheus is an open source system monitoring and alerting toolkit that has joined CNCF in 2016 as the second project after K8s. The architecture of Prometheus is shown in Figure 6.

---

\(^{27}\) [https://github.com/google/cadvisor/blob/master/docs/api.md](https://github.com/google/cadvisor/blob/master/docs/api.md)
Prometheus collects metrics from cAdvisor, via Pushgateway or from the instrumented jobs. It stores the collected datapoints in its Time Series Data Base (TSDB). There is an HTTP server that visualization tools such as Grafana\(^28\) can use to provide powerful visualization dashboards. Also, the monitoring data can be pulled by a custom API client and exported to an external consumer (e.g., to the MAPE loop via the Kafka bus in the context of the 5G-MEDIA architecture\(^29\)). Section 3.6 discusses details of our current monitoring solution for FaaS and provides an outlook per migration to OSM R4 as per its experimental monitoring functionality.

2.4. Logging support with ELK Stack on K8s

Elasticsearch, Logstash, and Kibana (ELK Stack) are the mainstream tools for log aggregation and analysis at scale. Logging is an essential part of any DevOps cycle. VNFs executing as containers produce logs to stdout and stderr. Since containers are ephemeral, logs should be retrieved from them for further processing before containers terminate. While collecting logs from a Docker container is possible through the Docker engine (`docker logs [OPTIONS] <container>`), the native functionality provided by Docker does not scale. A container can crash, a POD can be evicted, and a node can fail. In these cases, the native logging functionality

\(^{28}\) https://grafana.com/

\(^{29}\) See 5G-MEDIA D3.3 “Specification of the 5G-MEDIA QoS Control and Management Tools” for the details of the overall monitoring framework designed by the project.
is insufficient. Therefore, a cluster level logging solution with a separate storage and lifecycle, which is not dependent on the nodes, PODs, or containers is required.

Kubernetes does not provide a native logging solution. Rather, several existing logging stacks can be integrated with K8s.

![ELK Stack Schematics](https://hackernoon.com/deployment-of-full-scale-elk-stack-to-kubernetes-6f38f6c557c55)

Figure 7 captures a typical schema of deploying ELK stack to K8s. We are particularly interested in the container logs. All container logs are stored as files in /var/log/containers directory at each node in the K8s cluster. A Filebeat agent running as a DaemonSet is configured per node. A DaemonSet is a K8s workload object. A DaemonSet represents a group of replicated PODs with one POD per node policy. As nodes are added to the K8s cluster, DaemonSet automatically adds PODs to the new nodes. DaemonSet is a particularly useful K8s construct for deploying ongoing background tasks that need to run on the nodes and which do not require constant user attention and intervention (e.g., collectd daemon or Filebeat daemon). The Filebeat daemon is a replacement of an older logstash forwarder daemon. However, the basic functionality is the same. The Filebeat agent collects the files from /var/log/containers and pushes it to Logstash. Elasticsearch allows to query the files in Logstash. This architecture can be extended across the K8s clusters. Kibana dashboards are used to visualise the Elasticsearch queries.

In the current release of the FaaS VIM framework, we didn’t implement the cluster-wide logging solution.

OpenWhisk automatically stores application logs in CouchDB. The application logs can be retrieved from CouchDB using OpenWhisk CLI. This is a reasonable solution for a prototype or

---

30 [https://www.elastic.co/products/beats/filebeat](https://www.elastic.co/products/beats/filebeat)
and SDK. However, when production at scale is considered, this solution does not scale, because CouchDB will quickly swell with the application logs. To avoid this, a cluster-wide log forwarding solution is required.

IBM Cloud Functions commercial offering uses ELK for the cluster-wide logging. Also, ELK is part of the K8s IBM ICP commercial offering. As part of IBM’s exploitation in 5G-MEDIA, we will use the IBM existing solution for log analysis of OpenWhisk actions. This is also in full alignment with the rest of the architecture and will be useful for exploiting the project results contributing to the cloud native NFV orchestration transformation.

![Figure 8: Integration of the ELK Stack for FaaS VIM in 5G-MEDIA](image)

Figure 8 shows a planned integration between the ELK stack for the FaaS VIM and the rest of the 5G-MEDIA SVP. A user instantiates a VNFD via OSM. The FaaS VIM Plugin invokes an OpenWhisk action that implements the VNF. OpenWhisk offloads the action to K8s as POD. The Filebeat agent is configured as DaemonSet in K8s, collects the log data and pushes it to the ELK stack. The user analyses the log data using Kibana.
3. **FaaS Architecture in 5G-MEDIA**

Figure 9 shows an ETSI MANO compatible FaaS VIM reference architecture. The FaaS VIM supports the standard VNFM-VI, OR-VI, and NF-VI interfaces on the northbound. On the southbound, FaaS VIM is specific to a FaaS Framework being used for implementation. The FaaS framework can be deployed on top of some underlying PaaS that, in turn, can use IaaS virtualization technology or directly run on the bare metal machines.

![Figure 9: ETSI MANO Compatible Reference Architecture of a FaaS VIM](image)

The rationale of the architecture shown in Figure 9 is to allow compatibility with the MANO standard without tight coupling the FaaS paradigm to some specific implementation or deployment options.
In Figure 10, the reference implementation (i.e., the software architecture) of the FaaS VIM in 5G-MEDIA is depicted. OSM is used as VNF MANO Resource Orchestrator (Release 3 was used for actual implementation as we go to press). A FaaS VIM plugin is implemented with Apache OpenWhisk being used as the FaaS framework. To ensure virtualization technology neutrality and portability across different use cases and future environments, Kubernetes is used as an industry de facto standard container orchestrator. Kubernetes can be either deployed on the bare metal servers or on top of some virtualization technology. We use Open Stack for the first prototype.

All management flows start at the OSM. It is required that the VNF image would be already uploaded to the VIM, before the VNF can be onboarded (i.e., a VNFD can be created in the catalog), instantiated, monitored, and terminated. The ETSI flows that are impacted by FaaS are discussed in Section 3.4.
Figure 11 zooms into the FaaS Framework (embodied by Apache OpenWhisk) and Container Orchestrator (realized via CNCF Kubernetes) layers of Figure 10.

To facilitate typical 5G-MEDIA use cases, where several FaaS VNFs might comprise a FaaS network service and should be pre-uploaded consistently to the FaaS VIM, we utilize a wskdeploy utility that helps to describe any part of the OpenWhisk programming model using a Manifest file written in YAML. The utility is used to deploy OpenWhisk assets, such as Packages, Actions, Triggers, and Rules. Moreover, Packages can reference other Packages to facilitate code reuse.

Despite its name, wskdeploy does not instantiate assets, but rather saves the metadata about them (including the source code in case of the regular, specific run-time-oriented actions) into the OpenWhisk database.

Application developers use wskdeploy tool via the SDK to “pre-onboard” FaaS VNF “image” into the OpenWhisk database. As discussed in Section 4.5, wskdeploy allows to deploy a collection of the artefacts as a managed project, which can be exported to other FaaS VIMs (e.g., installed at the edge), deleted from the OpenWhisk database, and specify relationships to other managed projects to facilitate code reuse.

In case of black box actions, the “image” comprises two parts:

- The action metadata (saved in the OpenWhisk database)
- The action container image (stored in some container repository, e.g., Docker Hub)

https://github.com/apache/incubator-openwhisk-wskdeploy
In case of the regular actions, the “image” comprises just the metadata and the source code. A specific run-time Docker container image that will be used to execute this code is always pre-loaded.

OpenWhisk actions (implementing VNFs) execute as Docker containers orchestrated by CNCF K8s COE. To manage the lifecycle of these serverless VNFs on K8s several management actions are added by the FaaS VIM implementation. These actions are bundled as a special purpose pre-deployed action life cycle management package.

- **Offload Action**: this action is called by the FaaS VIM upon a VNFD instantiation. It interacts an Offload Service (an HTTP service) running on one of the K8s Nodes. The Offload Action passes the meta information about the VNF action that should be executed and its statically (Day 0) configured parameters to the Offload Service and the latter creates a POD definition for the action and uses kubectl client to start its execution.

- **Terminate Action**: this special action interacts with K8s master via kubectl to terminate a POD running this VNF (kubectl delete pods <pod> --grace-period=<grace>).

- **Status Action**: this is a special action that interacts with K8s master via kubectl to pull metrics about the pod (kubectl top pod POD_NAME -containers)

The actions of the Lifecycle Management Package are called transparently by the FaaS VIM (embodied as the OSM R3 plugin interfacing to OpenWhisk on the southbound). The association between the VNF name, the POD name, and the OpenWhisk action name is stored in VNFR.

The actual K8s POD definition creates three Docker containers per each VNF instance, as outlined by the deep blue dashed rectangles at the K8s layer in Figure 11. Each such rectangle represents a single VNF. For the sake of the VNF instance lifecycle management, the POD is treated as a single entity. The functionality of each container in the VNF POD is as follows.

- **VNF instance container**: implements the VNF logic itself. It is being interconnected with other FaaS VNF instances via the K8s Flannel network (shown in red) and is discoverable by VNFs outside this K8s cluster by the K8s Ingress resource.

- **Day 1 configuration server container**: this container implements an HTTP server that serves the VNF instance configuration parameters as REST objects listening on a specified port for the incoming REST calls.

- **Caller container**: this container is a reduced functionality of a regular Invoker. It interacts with the HTTP server of the 5G-MEDIA action container base image (derived from the OpenWhisk base image) and executes the REST calls on [http://127.0.0.1/init](http://127.0.0.1/init) and [http://127.0.0.1/run](http://127.0.0.1/run) resources to inject the action code (in case of regular actions) and run the VNF instance container, respectively.

The three containers share the same file space. Upon instantiation by the Caller, the VNF instance expects to find its configuration parameters in the file space to be set up by the Day 1 configuration server. The programming model pattern for FaaS VNF is to start with a busy loop waiting for configuration of the expected parameters. Note that if the VNF is not
configured fast enough, it might be terminated due to the system limits, which are enforced in the FaaS framework to terminate an action after its maximal lifetime.

Each K8s node is connected either natively or by bridging to the physical network that has access to the Internet. Each POD is bridged to the same network (possibly on a private segment). Finally, the Flannel network, which connects the VNF instance containers running within the PODs is bridged to POD network. The K8s can be deployed either on BMs or VMs and either in a cluster or an all in one mode. The former is used in production, the latter – in SDK.

In Subsection 3.1.1 and Subsection 3.1.2 we will examine the FaaS VIM architecture and highlight deployment approach for VMware (all in one) and OpenStack (cluster), respectively, in greater detail.

### 3.1. Single FaaS VIM

Our current prototypical implementation of the FaaS VIM caters for a use case, in which FaaS VNFs execute in the same Data Centre. A natural extension, utilizing K8s ingress resource and SDN is considered in the next section.

#### 3.1.1. All-in-one FaaS VIM Deployment

Figure 12 shows all-in-one deployment of the FaaS VIM (OpenWhisk/K8s-based) on a bare metal VMware ESXi host. We start with this deployment option for simplicity and then discuss how FaaS VIM is deployed for production in a Data Centre.
There are three networks:

- The “green” network is a physical external network that provides connectivity to the Internet;
- The “blue” network is an internal network that facilitates communication between VMs on which K8s master and nodes execute. This network is bridged to the green network to allow external connectivity of VNFs that execute as container pods in K8s worker nodes;
- The “red” network is an overlay provided by K8s out of the box via Flannel, a container network fabric, designed for K8s. This network facilitates intra-VNF network communication for FaaS VNF.

VNF instances run as container pods in K8s, as explained in the previous section, are exposed as services of type NodePort. K8s automatically assigns IP and port number to the new VNF instance. FaaS VIM polls K8s to obtain this information and puts it into VNFR for the newly created instance.

---

32 [https://github.com/coreos/flannel](https://github.com/coreos/flannel)
33 [https://console.bluemix.net/docs/containers/cs_nodeport.html#nodeport](https://console.bluemix.net/docs/containers/cs_nodeport.html#nodeport)
Figure 13 summarizes the basic instantiation flow of a VNFD, in which the instantiation request is initiated by OSM.

0. Define VNF instantiation parameters by issuing REST call to Day 0 FaaS Configuration Service
1. An OSM user requests a NS instantiation. OSM starts instantiating VNFDs one by one;
2. Create VNF instance method is invoked on the FaaS Plugin (implementing OSM VIM Plugin interface);
3. FaaS VIM Plugin extracts a fully qualified OpenWhisk action name that corresponds to the FaaS VNF implementation, which was pre-onboarded on OpenWhisk. The fully qualified name, which was stored in the VNFD:vdu:vm_image at the time of on-boarding of the NSD;
4. Using the VNFD index in the NSD, FaaS VIM Plugin issues a REST call to Day 0 FaaS Configuration Web service, which can be run anywhere as along as it has a publicly accessible URL that FaaS VIM Plugin can access (this URL is part of the FaaS VIM Plugin configuration parameters). Day 0 configuration implementation is described in Section 4.1;
5. Using the parameter key-value pairs for initial configuration, FaaS VIM Plugin issues a REST call to OpenWhisk to invoke the OpenWhisk action that embodies a VNF with the keys being action parameter names and values being these parameter values;
6. In fact, in Step 5, rather than calling the action implementing the VNF directly, FaaS VIM Plugin calls a special action that we term offload action, passing it the fully qualified names of the VNF action and its parameters obtained in Step 4. The offload action executes a REST call to an offload service, which is collocated with the K8s master node;
7. The offload service issues a REST call to OpenWhisk to obtain the code of the data. In case of a Blackbox action, an action container repository location is fetched;
8. Using the data from Step 7, a K8s POD definition is created and executed on K8s as a service with IP and port being automatically allocated by K8s;
9. VNF container now executes and its IP and port are known to the K8s master;
10. Offload service obtains the IP and port for the running action (i.e., VNF that was instantiated) from the K8s master;
11. OSM requests the VNF IP address to update the VNFR data structure of the instantiated VNFD;
12. FaaS VIM plugin requests an IP address by executing a special Status Action (see Figure 11)
13. The Status Action interacts with the Offload Service and the requests an IP address of the newly instantiated action;
14. The FaaS VIM plugin obtains the IP address and returns it to the OSM, which
15. Stores the address in the VNFR along with other metadata stored in this VNFR.

To the OSM the FaaS VNF instance appears to be just a regular VNF instance that can be managed using standard ETSI MANO flows via OSM. Other orchestration flows that are impacted by FaaS are discussed in Section 3.4.

This basic approach is generic, and it seamlessly extends to the more general deployments in a single and multiple data centres as explained in the next subsection and in Section 3.1.2.
3.1.2. FaaS VIM Production Deployment

In this subsection, we describe FaaS VIM deployment in production in a single data centre with Open Stack being the virtualization technology. The details of a deployment are presented in Figure 14.

Conceptually, Figure 14 is similar to Figure 12. There are four networks that should be considered:

- **Management Network**: depicted using the black color. This is a segment of the physical network that exists prior to the Open Stack installation. Open Stack Compute Nodes and the Open Stack Controller Node are connected to this network. It is not managed neither by the Open Stack admin nor by the tenants. The Management Network has access to the Internet.

- **Provider Network**: depicted using the green color. This is a virtual network provided by the Open Stack administrator and controlled by Open Stack’s Neutron. The provider network segment maps to the existing physical network (i.e., Management Network). The Provider Network is connected to the Internet via a gateway to the Management Network. The gateway is not managed by Open Stack.

- **Self-Service (AKA ‘tenant network’)**: depicted using the blue color. This is a private network segment completely controlled by Neutron. The tenant network is provided by the Open Stack administrator and is connected to the Provider Network and Management network via vRouter managed by Neutron.

- **Flannel network**: depicted using the red color. The Flannel network is managed by K8s. A new POD created on K8s is automatically attached to the Flannel network via its eth0 interface.
All Compute Node hosts and the Controller host have two network devices: ens32, ens33. The first network device, ens32 is assigned a static IP address on the Management Network. The second network device, ens33 device is used to interconnect the hosts via L2 switch with promiscuous mode enabled. The IP addresses on the green, blue, and red networks are dynamically allocated. The presented networking layout use standard Open Stack and K8s practices. For more details consult Open Stack networking documentation\(^{34}\) and K8s networking\(^{35}\).

![Diagram of FaaS VIM Deployment on Open Stack](image)

**Figure 14: FaaS VIM Deployment on Open Stack**

For simplicity of the presentation, we omit the OSM and the Day 0 configuration service from Figure 14. The OSM and the Day 0 configuration service can be deployed anywhere in the network (including outside of the data centre), as long as they are reachable by their public URLs. In this setting, the public URL of OpenWhisk is surfaced by NGINX (which is part of the OpenWhisk distro). The OSM FaaS Plugin uses this URL to interact with OpenWhisk on the southbound.


\(^{35}\) [https://kubernetes.io/docs/concepts/cluster-administration/networking/](https://kubernetes.io/docs/concepts/cluster-administration/networking/)
3.2. Federated FaaS Architecture

The federated architecture blueprint is described in Section 4.5.4 of Deliverable D2.3 “5G-MEDIA Platform Architecture”. In the next release of the FaaS framework we will implement this feature to address requirements of the three use cases.

3.3. Multiple FaaS VIMs and VNF Chaining across FaaS and Non-FaaS VIMs

Figure 15 generalizes production deployment of Figure 14 to multiple cloud technologies and multiple data centers, allowing chaining of FaaS and non-FaaS VNFs across multiple data centers and cloud technologies. The cloud neutrality approach in K8s allows to replicate our K8s FaaS framework to a number (e.g. three as per Figure 15) different environments to address all 5G-MEDIA use cases, without making any changes to the FaaS framework itself.

In each of the VIMs, a K8s POD is available for VNF FaaS, which exposes Flannel ports to let the various FaaS VNFs be connected to a private network for configuration and operations. VNFs are connected to both, the private network for operation and management purposes and the “external” public network to implement the inter-VIM data plane connectivity among functions.
As described in D3.1: “Initial Design of the 5G-MEDIA Operations and Configuration Platform” sec. 4.5, we plan to realize the interconnection among the various VIMs through an SDN WAN Infrastructure Manager (WIM), which acts as a specialized VIM attached to the NFVO to establish and operate connectivity of the external connection points from the different NFVI-PoPs. In a homogeneous multi-VIM environment, solutions may just exist or are under development for the multi-VIM interconnection, like for example the VPN as a Service solution to interconnect multiple OpenStack instances via IPSec VPN tunnels or BGP VPN Interconnection or Neutron-to-Neutron interconnections proposed for inclusion in OpenStack Rocky planned to be released in Q3-2018. Contrarily, the seamless interconnect of VIMs based on different technologies, and – in particular – between an OpenStack VIM with a FaaS VIM are still not available. At the time of writing this document, the team is still evaluating different options to interconnect OpenStack with OpenWhisk environments for the purposes of the 5G-MEDIA representative use cases, in order to integrate the standard VNF and FaaS VNF execution NFVIs and allow the proper VNF chaining. Future deliverables of WP3 will detail the final solution of choice and specify architecture details of infrastructure and required plugins in to the NFVO (ETSI OSM) to be used (or developed) to implement the multi-VIM networking across the SDN network while preserving the isolation of the different tenants in the various VIMs.

3.4. ETSI Flows

One of the design tenets of the FaaS framework for the 5G-MEDIA platform is minimal changes to the ETSI MANO flows. In fact, except for the instantiation flow described in Section 3.1.1, the on-boarding flow described in Section 4.5.4 of Deliverable D2.3 “5G-MEDIA Platform Architecture”, and the scale-up/down flow, there are no changes at all to the standard ETSI MANO flows. We now describe the peculiarities of the scale out/in and scale up/down flows in the FaaS framework.

One of the main advantages of the FaaS approach is the built-in support for scale out. Indeed, as shown in Figure 3, OpenWhisk has a built in Load Balancing mechanism. Therefore, when a new VNF instance should be started, it just amounts to invoking another instance of the OpenWhisk action (via the FaaS VIM Plugin) that implements this VNF. Moreover, since the OpenWhisk actions have a limited life time for execution, scale-in – which is a considerably more complex operation than scale-out – is supported out of the box as well.

However, scale-up/down flows are not supported in OpenWhisk. Once a container is started its resource allocation parameters are not changeable. Even though it is possible to implement this functionality via K8s API server, the merits of this approach are questionable given the limited life time of the action and it might also result in instability of deployment. Therefore, we are not planning to support scale-up/down flows in the FaaS VIM.

36 https://docs.openstack.org/neutron/latest/admin/vpnaas-scenario.html
37 https://docs.openstack.org/networking-bgpvpn/latest/user/overview.html
38 https://specs.openstack.org/openstack/neutron-specs/specs/rocky/neutron-inter.html
Similarly, to support flavours of VM based VNFs, OpenWhisk can support multiple flavours of actions (resulting in different sizes of containers upon instantiation). Memory allocation is one of the attributes of the metadata that one can define when pre-onboarding an OpenWhisk action to the FaaS VIM and then on-boarding it to the catalog.

3.5. Pre-onboarding

This phase deals with uploading all FaaS VNF images of a given FaaS network service, into the VIM. This is achieved by creating the necessary assets, such as packages, actions and docker images specification into the OpenWhisk database. We are going to use wskdeploy to describe these assets in a well-known yaml syntax and submit it against the OpenWhisk installation of the FaaS VIM by supplying wskdeploy with the API host and key token (e.g. the prebuilt ‘guest’ token).

Note that in the case of the SDK it would result in creating the assets in Lean OpenWhisk but this does not affect the semantic of the flow as it would be identical for both cases.

In the below example, we would like to pre-on board the FaaS ping and pong VNFs. Thus, we create a wskdeploy asset definition file that includes the package where FaaS actions are going to reside, along with the action names and their corresponding docker images:

```
packages:
  5g-media:
    version: 1.0
    actions:
      ping-action:
        docker: docker5gmedia/action-ping
      pong-action:
        docker: docker5gmedia/action-pong
```

We save it into `my_specification.yml` and invoke:

```
wskdeploy --apihost 172.17.0.1 --auth 23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123z03xZCLrMN6v28KK1dXYFpXIPkcc0Fqm12CdAsMgRU4VrNZ91yGVCGuMDGIwP --m my_specification.yml
```

Then, verify that the assets had been properly created by issuing wsk against the above OpenWhisk service (i.e. supply same --apihost and --auth):

```
wsk package list
wsk package get <package name>
wsk action list
wsk action get <action name>
```
3.6. Monitoring

Figure 16 shows the high-level monitoring architecture for FaaS in 5G-MEDIA platform. As one can readily observe, the FaaS monitoring is fully aligned with the monitoring architecture presented in Deliverable D2.3 “5G-MEDIA Platform Architecture”. In our current implementation, we do not connect the API client of Prometheus to the Kafka bus yet. It is planned for the coming implementation sprints. At the moment, we simply visualize the metrics using Grafana dashboard. Figure 17 shows a sample dashboard with the OpenWhisk action level resolution.

The OpenWhisk action level resolution is achieved as follows. The Offload Service interacts with the K8s API server via REST to label the VNF PODs with the OpenWhisk fully qualified action name at the time when the action is being offloaded for execution at K8s.
Figure 17: Example Grafana Dashboard with Per-Action Resource Consumption

The dashboard in Figure 17, is produced by a Prometheus query shown in Table 1. More complex queries generating virtually any visualization for OpenWhisk actions are possible.

Table 1: CPU utilization of OpenWhisk actions on K8s

```
sum(
    max(kube_pod_labels{label_ow_action!=""}) by (label_ow_action, pod)
    *
    on(pod)
    group_right(label_ow_action)
    label_replace(
        sum by (pod_name) (
            rate(container_cpu_usage_seconds_total{namespace="$namespace"}[5m])
        ), "pod", "$1", "pod_name", "(.+)")
    ) by (pod, label_ow_action)
```

In OSM R4 the experimental monitoring functionality is added as shown in Figure 18. We will explore this new architecture as we will migrate 5G-MEDIA from OSM R3 to OSM R4. As one can see, our current architecture is very much in-line with the OSM R4 one and migration should be relatively painless.

In this architecture, we can still use cAdvisor as the source of the metrics and even seamlessly reuse Prometheus as a continuous source of metrics for selected VDUs.
The IBM’s exploitation plan for monitoring for OpenWhisk in IBM ICP is well aligned with the current 5G-MEDIA monitoring architecture for FaaS. IBM ICP is a K8s cluster that has cAdvisor, Prometheus, and Grafana installed as part of the offering. Hence, the results developed in 5G-MEDIA for serverless VNF monitoring have a clear exploitation path in IBM ICP.

3.7. Integration with MAPE

The MAPE component is responsible for Monitoring, Analysis, Planning and Execution within the 5G-MEDIA Service Virtualisation Platform. It is aiming at monitoring the performance of the deployed media NSs and their constituent VNFs as well as the usage and availability of the infrastructure (computational, storage and network resources) for the deployed functions. The monitoring service collects metrics from the infrastructure and the NSs/VNFs. The Analysis, Planning and Execution services implement advanced data analytics and optimisation algorithms to manage the infrastructure and the deployment and scaling of the NSs and VNFs to ensure they are running as efficiently, delivering high levels of performance, and ensuring the end-users are receiving high quality media applications.

There are three aspects to the integration of FaaS with the MAPE component. Firstly, there is the monitoring of the FaaS environment, where the instantiatiation of FaaS actions can be logged, computational infrastructure utilisation can be monitored and reported together with any application-specific metrics made available by the FaaS actions.

The second aspect is related to the optimisation algorithms implemented by the Cognitive Network Optimiser as part of the Pre-process & Analysis and the Planning services of the MAPE component. The CNO algorithms will, in general, be determining the optimal edge-node locations and clusters within those nodes where VNFs should be deployed. Optimisation will
be undertaken over standard as well as FaaS-enabled VNFs. Selection between computational nodes for the execution of FaaS-VNFs is dependent upon the final design of the federation architecture as mentioned in section 3.2 and on the mechanisms for chaining across multiple FaaS VIMs as presented in section 3.3. The decision on when the set of actions forming the overall VNF should be chained across multiple VIMs and multiple sites is one that can be made by the FaaS controller by itself, or by the CNO, which has visibility of the full deployment environment for the NS. The options for the cooperation between the CNO as implemented in the Planning service of the MAPE component and the FaaS controller will be studied in detail in the second project year. The resulting architecture, which will show how optimisation decisions can be effectively made through the cooperation between algorithms and through the delegation of decisions through directives from the CNO to the FaaS controller, will be presented in deliverable D3.4.

The third possibility for the integration of FaaS and MAPE is in the deployment of the algorithms for the MAPE services themselves. The Pre-process & Analysis service undertakes data analysis, anomaly detection, resource usage and load forecasting and makes use of a range of machine learning techniques to implement these algorithms. The option of using a FaaS approach for the implementation of analysis actions that are deemed necessary when certain conditions present themselves will enable the analytics algorithms to scale on demand to the analysis needs of the system. The same is also true for the Planning and Execution services: small-scale optimisation functions can be spawned as parallel FaaS actions which will enable a scalable optimisation system for complex, distributed infrastructures. One downside to this is that it may result in local optima rather than globally optimal solutions. The benefits of using FaaS for implementing analytics and optimisation algorithms will be studied during the second project year and will be reported in deliverable D3.4.

3.8. Multitenancy

The integration of OpenWhisk with the AAA service of 5G-MEDIA will be explored in the second year and will focus on two main goals: to manage the OpenWhisk authorization tokens that will be provided to the Catalogue instead of plain credentials and to configure remotely users and tenants to be assigned to Catalogue users, in case Catalogue and OpenWhisk instances will share the same security domain.

Both are meant to simplify the configuration of OpenWhisk from the perspective of a system user hiding the connection details while, on the other hand, minimizing the sharing of plain credentials.

The first goal will require the integration with OpenWhisk authorization service to retrieve temporary tokens for the Catalogue, possibly based on a shared protocol such as OpenID Connect\(^{39}\) to go beyond the simple usage of service credentials and manage different security domains.

\(^{39}\) http://openid.net/connect/
The second goal will require the integration with the OpenWhisk services to allow the remote configuration of users and tenants, with the possible support to fine grained authorization (e.g. based on users and resources attributes, the ABAC\(^{40}\) model) in case specific requirements will be elicited during the second year of the project.

Further details about the integration options with the AAA service of 5G-MEDIA are contained in the Deliverable D4.1 “5G-MEDIA Catalogue APIs and Network Apps” \(^{41}\).

\(^{40}\) [https://en.wikipedia.org/wiki/Attribute-based_access_control](https://en.wikipedia.org/wiki/Attribute-based_access_control)

\(^{41}\) [http://www.5gmedia.eu/outcomes/deliverables/](http://www.5gmedia.eu/outcomes/deliverables/)
4. Reference Implementation of the FaaS Architecture

In this section we describe details of the reference implementation of the first release of the FaaS VIM framework for the 5G-MEDIA platform corresponding to the reference architecture described in the previous section.

The first release of the reference implementation includes the following features and components:

- OSM R3 FaaS VIM Plugin for Apache OpenWhisk;
- OpenWhisk functions offload service for K8s with GPU affine placement support and advanced placement policies;
- Support for network communication among FaaS functions;
- Support for monitoring and powerful visualization of OpenWhisk actions realizing VNFS, using Grafana dashboards;
- Support for chaining FaaS and non-FaaS VNFs within a single data centre;
- Cloud and virtualization technology independence demonstrated for OpenStack and VMware ESXi;
- Day 0, 1 FaaS configuration services;
- Low footprint all-in-one environment based on Lean OpenWhisk (leveraged in SDK, see D5.1: 5G-MEDIA Programming Tools

Annex A provides the installation and getting started instructions for the first release of the 5G-MEDIA FaaS framework.

4.1. FaaS VNF Day 0 Configuration Service

Figure 19: Day O FaaS Configuration Service
Upon instantiation, a VNF might require parameters to allow its proper initialization. The Day 0 configuration parameters comprise two groups:

- **Action parameters**: key/val JSON format parameters that the FaaS VNF is instantiated with. These can be arbitrary custom parameters;
- **Server ports**: a list of VNF application ports that are needed for this VNF to get externally exposed as a service. As an example, consider in UC1 the Game Server and FaaS vTranscoders need to inter-connect.

For example, the following REST calls set FaaS VNF instantiation parameters for the two VNFs of a given NS named “star_balls”:

```bash
curl -POST -d '{"action_params": {"Name": "VNF_1", "Rate": "50"}}' http://config_service_ip:config_port/conf/star_balls/BlackBoxVNF/1

curl -POST -d '{"action_params": {"Name": "VNF_2", "Rate": "100"}}' http://config_service_ip:config_port/conf/star_balls/BlackBoxVNF/2
```

Figure 19 shows the architecture of the Day 0 configuration service. When NSD gets instantiated, OSM instantiates VNFDs comprising the NSD one by one. FaaS VIM queries the Day 0 configuration service for action parameters for the VNFDs according to their index (sequence of appearance in the NSD) under the given network service name, e.g., star_balls/BlackBoxVNF/1. OpenWhisk will then invoke the actions with these parameters (e.g., Name and Rate in the example above).

### 4.2. FaaS VNF Day 1 Configuration Service

We found Jujucharms\(^{42}\), which is a standard way of configuring VNFs (VM based) in OSM, to be slow in OSM R3 for the cloud native framework, such as OpenWhisk. Also, using Jujucharms in OSM R3 required to have support for SSH in the base image of the 5G-MEDIA black box action container (derived from the OpenWhisk one). Supporting multiple and fast proliferating SSH keys for short lived serverless VNF actions offsets the benefit of light weightiness and fast transparent execution accrued by using FaaS.

Hence, in the current implementation, we avoided using Jujucharms altogether, opting for the custom Day 1 FaaS Configuration service. However, we will revisit this issue in the OSM R4 and explore harmonization between Jujucharms and FaaS VIM.

In our current implementation, after the FaaS VNF gets instantiated and its POD receives an IP address, it can be dynamically configured by pushing configuration parameters to the Day 1 configuration service running in the same POD via a REST call. This is referred to as Day 1 configuration. For example, the below REST call injects “target_ip” parameter into the VNF whose ip address is 172.17.0.5 (obtained from OSM VNFR):

```bash
...
```

\(^{42}\) [https://jujucharms.com/](https://jujucharms.com/)
4.3. **OpenWhisk to K8s Actions Offloading Service**

The Offload Service is an HTTP service implemented in Python, running as a container within a POD on K8s. A user never interacts with the Offload Service directly. This is an internal utility used by the FaaS VIM. The Offload Service retrieves the metadata about an action being invoked through the FaaS VIM, by making a REST call to the OpenWhisk repository. The Offload Service is responsible for creating a yaml definition of a POD that will run an action and labelling it with the action name and placement related labels, which will be used for monitoring data visualization and placement of the action, respectively.

The offload service can be extended to other COEs.

4.4. **OpenWhisk Support for GPUs and Affinity/Antiaffinity on K8s**

It is expected that media intensive applications will make use of GPUs to provide a required QoE to the end users. As a characteristic example, consider UC1, which deals with the tele-immersive gaming. UC1 requires GPUs for the vTranscoder VNF operation.

The vTranscoder of UC1 is implemented in C, which is not one of the run time environments supported by OpenWhisk out of the box. Therefore, the vTranscoder implementation is a Blackbox action, which allows a Docker container running arbitrary user code to execute as OpenWhisk action.

4.4.1. **5G-MEDIA Docker Base Image with NVIDIA GPU Support**

To containerize vTranscoder so that it could run as an OpenWhisk Blackbox action while exploiting the NVIDIA libraries for GPUs, we created a new base image from which the vTranscoder container is created. Table 2 shows the Dockerfile of the new base image. It should be noted that the regular Blackbox action base image of OpenWhisk does not support GPUs and this is one of the innovations that the 5G-MEDIA project plans to contribute to the Apache OpenWhisk open source community.

It should be noted that a similar technique can be used to exploit the GPUs also in the run times, which are supported out of the box by Apache OpenWhisk, to allow the VNF developers to develop applications in languages, such as Java, Python, Node.js, Go, etc., without the need to explicitly containerize the VNF.

An ability to do this is an important feature of the serverless programming model and we will explore this integration option in the second year of the project.

The base image container is publicly available from the Docker Hub repository of 5G-MEDIA: [https://hub.docker.com/r/docker5gmedia/5gmedia-base-gpu/](https://hub.docker.com/r/docker5gmedia/5gmedia-base-gpu/)
Table 2: Dockerfile of base image for blackbox actions exploiting NVIDIA GPU

<table>
<thead>
<tr>
<th>FROM nvidia/cuda:8.0-runtime-ubuntu16.04</th>
</tr>
</thead>
<tbody>
<tr>
<td># Upgrade and install basic Python dependencies for this black-box action</td>
</tr>
<tr>
<td>RUN apt-get update &amp;&amp; apt-get install -y bash</td>
</tr>
<tr>
<td>bzip2</td>
</tr>
<tr>
<td>vim</td>
</tr>
<tr>
<td>curl</td>
</tr>
<tr>
<td>gcc</td>
</tr>
<tr>
<td>libc-dev</td>
</tr>
<tr>
<td>python-pip</td>
</tr>
<tr>
<td>RUN pip install --upgrade pip setuptools six</td>
</tr>
<tr>
<td>RUN pip install --no-cache-dir gevent==1.2.1 flask==0.12 requests==2.13.0</td>
</tr>
<tr>
<td># Do not modify - this is the internal openwhisk invoker service port</td>
</tr>
<tr>
<td>ENV FLASK_PROXY_PORT 8080</td>
</tr>
<tr>
<td>RUN mkdir -p /actionProxy</td>
</tr>
<tr>
<td>ADD actionproxy.py /actionProxy/</td>
</tr>
<tr>
<td>RUN mkdir -p /action</td>
</tr>
<tr>
<td>ADD stub.sh /action/exec</td>
</tr>
<tr>
<td>RUN chmod +x /action/exec</td>
</tr>
<tr>
<td># for the configuration service to push parameters</td>
</tr>
<tr>
<td>RUN mkdir -p /conf</td>
</tr>
<tr>
<td># base cli to be used by the app for retrieving single parameter</td>
</tr>
<tr>
<td>ADD get-conf /</td>
</tr>
<tr>
<td>RUN chmod +x /get-conf</td>
</tr>
<tr>
<td>CMD [&quot;/bin/bash&quot;, &quot;,-c&quot;, &quot;,cd actionProxy &amp;&amp; python -u actionproxy.py&quot;]</td>
</tr>
</tbody>
</table>

4.4.2. Placement Annotations

Having a base container image that can exploit GPUs for a Blackbox action is not enough in itself under all execution scenarios. If a whole GPU should be exposed to the action and there is more than one such action instance executing simultaneously in the platform, a scheduling policy that ensures affinity with GPUs and anti-affinity of actions should be enforced at run time.

In our UC1 scenario we have exactly this problem: in the simplest setting there are two vTranscoders, each of which requires an exclusive control of a GPU of a K8s node, on which it executes as a Docker container inside the K8s POD.

These requirements lead us to enhancing OpenWhisk with support for advanced placement policies for actions executing on K8s.
To that end, we use the OpenWhisk annotations mechanism. The OpenWhisk annotations are key/value pairs that can be added to any OpenWhisk asset (Action, Trigger, Rule, Package, etc.) to convey additional meta-data about the assets to the asset user (either a human one or a program). OpenWhisk annotations are optional. They are not interpreted by the OpenWhisk engine itself. Hence, they represent an ideal vehicle for experimentation and innovation without the need to make complex changes in the data model.

We use OpenWhisk annotations to describe the placement policies. When an action gets offloaded to K8s, the Offload Service translates these annotations to K8s scheduler hints that cause K8s master to invoke the action POD on a correct node (i.e., the one that meets the placement criteria).

The following policies are currently supported:

- **Node affinity** – invoke an action on a K8s Node that meets the specified labelling (in case of multiple labels, the labels are logically AND-ed);
- **POD anti-affinity** – do not invoke an action on a K8s Node, if the node already executes another instance of this action (i.e., an action annotated with same labels).

A placement annotation comprises two parts:

- "labels": a dictionary of key/value labels that the action is annotated with
- "placement": a dictionary with sub-keys that provide further details required to complete placement by K8s scheduler:
  - "node-selector": key/val labels to be matched against kubernetes nodes
  - "action-anti-affinity": true: do not place together two actions with the same labels. The default is false.

For example, the following action annotation:

```json
{
  "labels": {
    "processor-required": "gpu"},
  "placement": {
    "node-selector": {
      "processor": "gpu",
      "disk": "ssd"
    },
    "action-anti-affinity": "true"
  }
}
```

requests from OpenWhisk to invoke the action on a node that is labelled with both "processor":"gpu" AND "disk":"ssd". Since "action-anti-affinity": "true" This action should not be collocated with another action that is labelled with "processor-required": "gpu".

This approach generalizes to any container orchestrator engine. In the 5G-MEDIA implementation, we use K8s as COE, so the offload service translates the action placement
annotations to a POD yaml definition that is sent to kube-scheduler. In our example, the yaml definition created by the Offload Service would look as follows.

```yaml
apiVersion: v1
dkind: Pod
metadata:
  name: vtranscoder
  labels:
    processor-required: gpu
spec:
  nodeSelector:
    processor: gpu
disk: ssd
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: processor-required
                operator: In
                values:
                  - gpu
          topologyKey: kubernetes.io/hostname
  containers:
    - name: vtranscoder-GPU
      image: 5g-media/vtranscoder-GPU
```

For this definition to take effect in terms, the K8s nodes should be labeled appropriately, so that the node’s label would match the node-selector definition of the action placement annotation.

The is the responsibility of the operator to properly label the K8s nodes via (see `kubectl label nodes <node name> <label name> <label value>`)\(^{43}\)

\(^{43}\) [https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/)
4.5. **OpenWhisk Support for project deployment and migration**

Wskdeploy\(^44\) is a utility to help a developer to describe and deploy any part of the OpenWhisk programming model using a Manifest file written in YAML. It should be used to deploy OpenWhisk Packages, Actions, Triggers, Rules, Sequences, and Compositions in a single command.

```
wskdeploy export --projectname managed_project_name
```

allows to "export" a specified managed project into a local file system. Namely, a `managed_project_name.yml` Manifest file will be created automatically. This Manifest file can be used with wskdeploy to redeploy the managed project at a different OpenWhisk instance. If the managed project contains dependencies on other managed projects, then these projects will be exported automatically into their respective manifests.

The export feature has been developed by IBM during the 5G-MEDIA project. This functionality is at the core of the federated OpenWhisk architecture artefacts synchronization protocol, which will be provided in the next release to address distributed scenarios in the use cases.

Wskdeploy can be used in addition to the OpenWhisk CLI. In fact, this utility uses the OpenWhisk Go Client to create its HTTP REST calls for deploying and un-deploying projects\(^45\).

---

\(^{44}\) [https://github.com/apache/incubator-openwhisk-wskdeploy/blob/master/README.md](https://github.com/apache/incubator-openwhisk-wskdeploy/blob/master/README.md)

\(^{45}\) [https://github.com/apache/incubator-openwhisk-client-go](https://github.com/apache/incubator-openwhisk-client-go)
Annex A: Installation and Getting Started

Prerequisites: at least three Ubuntu 16.04 machines will need to be provisioned in the targeted OpenStack (e.g., Queen release). The machines should be connected to a provider network (e.g., 10.30.0.0/16) that can be externally reachable from the Internet. Kubernetes master, OpenWhisk and OSM R3 should be installed on one of the machines in an all-in-one installation. On the other two machines, K8s nodes should be installed. The Ubuntu machines can be VMs unless GPU support (e.g., UC1) is required. In this case, the control node, which runs OSM, K8s master and OpenWhisk can be a VM, and the other two machines should be BM servers to properly expose GPUs and to avoid nested virtualisation.

4.6. Kubernetes cluster

Comprises of three ubuntu 16.04 VMs: master and two nodes each configured with 2 vCPUs, 4 GB ram and 40 GB disk.

The simplest way to install Kubernetes would be to use kubeadm: https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm. Following that, Flannel network should be installed by applying this file on the master:

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/v0.10.0/Documentation/kube-flannel.yml
```

Finally, issue `kubectl join <vm ipaddress>` on the two nodes passing their provider network ip address. This will add them to the kubernetes cluster and automatically install the kubernetes software stack on them.

4.7. Offload-Service

The offload service is responsible to manage the offloaded action POD (deploying, status retrieval, deletion). It is a pythonic micro-service that listens on a predefined port and accepts requests from internal OpenWhisk actions that get invoked by the FaaS VIM.

This service is installed as a Kubernetes POD.

Before installing the service, RBAC roles (Role and RoleBinding types) should be applied so that it has permissions to manage the action PODs via kubectl APIs.

Also, the service should be externally available so that it can serve requests from outside the Kubernetes cluster (i.e. FaaS VIM plugin). For this, we define a yaml specification file that wraps it with a NodePort Service resource that causes Kubernetes to map the offload service application port to a one on the master node.

Finally, the offload service is being deployed using the yaml specification similar to the below:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ow-offloadserver
spec:
  replicas: 2
```
4.8. OpenWhisk

OpenWhisk all-in-one is installed on an ubuntu 16.04 VM configured with 2 vCPUs, 4 GB ram and 40 GB disk and is connected to the provider network.

Follow the below installation instructions


https://github.com/apache/incubator-openwhisk/blob/master/docs/cli.md

The built-in "guest" user authentication token is being used when invoking OpenWhisk APIs. The token is located under ansible/files/auth.guest

Three OpenWhisk actions (implemented in python) should be created. They will be invoked by the FaaS VIM plugin to offload and manage the actions. They will reside under K8s package:

1. /guest/k8s/offload – responsible to offload a given action on a given kubernetes cluster
2. /guest/k8s/action_get_pod – responsible to retrieve POD information for a given JobId on a given kubernetes cluster
3. /guest/k8s/action_delete_pod – responsible to delete all POD resources for a given JobId

4.9. Configuration Service (Day0 configuration)

FaaS configuration service is responsible to store FaaS VNF initialization parameters. It is a pythonic micro service deployed in OSM R3 host. FaaS VIM plugin queries those parameters during VNF instantiation via /conf/<ns name>/<vnf name>/<index> api

The following curl example sets instantiation parameters to the VNF whose name is "vtranscoder" located at the first index of nsd named "star_balls"
curl -H "Content-type: application/json" -POST -d '{"action_params": {"rate":"30","codec":"mpeg4"}}'
    http://osm_ip_address:5000/conf/star_balls/vtranscoder/1

Once star_balls nsd get instantiated by OSM, FaaS VIM on new_vminstance hook get called for every VNF (i.e. vtranscoder) that uses above curl to query instantiation parameters.

4.10. Configuration Service (Day1 configuration)

The configuration service is also responsible to dynamically configure FaaS VNF by injecting parameters into the VNF. For example, the below REST call injects “target_ip” parameter into VNF whose ip address is 172.17.0.5 (obtained from OSM VNFR):

curl -POST -d '{"value": {"param_name": "target_ip", "param_value": "172.17.0.10"}}' http://config_service_ip: config_port
    /conf/172.17.0.5

4.11. FaaS-VIM Plugin

This is a pythonic plugin that implements OSM R3 abstract VIM class. It is installed inside RO container: /usr/lib/python2.7/dist-packages/osm_ro/vimconn_faaS.py

The following OSM CLI is used to create a FaaS VIM instance of this plugin


The following arguments are supported:

- **name** | string
  
  Name of this VIM instance (e.g. openwhisk_k8s_vim)
  
  (mandatory)

- **auth_url** | string
  
  URL of OpenWhisk management API
  
  (e.g. https://:443, or for lean OW: http://:10001)
  
  Note: make sure to surround it with quotes
  
  (mandatory)
- **tenant** | string
  OpenWhisk tenant that will be used for this FaaS VIM (e.g. whisk.sys)
  (currently being ignored; will be supported in the next release)

- **account_type** | string
  VIM type. Should be set with: FaaS
  Required

- **offload-service-url** | string
  URL of OpenWhisk offload-service (e.g. http://172.16.0.251:31567)
  Note: make sure to surround it with quotes
  Required

- **configuration-service-url** | string
  URL of FaaS config service (e.g. http://10.205.110.1:5000)
  Note: make sure to surround it with quotes
  Required

- **offload-action** | string
  Fully qualified internal action name responsible to offload the action
  (e.g. /guest/k8s_pkg/offload)
  Required

- **auth_token** | string
  OpenWhisk API authentication token.
  Note: make sure to surround it with quotes
  Required
5. Conclusions

In the first release of the FaaS framework we focused on the basic functionality that will be validated in the context of UC1 initial pilot. The next release will address the extended UC1 scenarios (including QoS management) and encompass functionality required in UC2 sub-use case of mobile contribution and UC3 replay functionality enablement. Also, we will address additional topics related to efficient operation of the FaaS framework in 5G-MEDIA platform, as well as specific FaaS workloads. To this end, we will seek extending the FaaS framework first release along the following dimensions:

- Full integration of the FaaS framework with the MAPE loop;
- Full support for FaaS artefacts development, deployment, and synchronization in the federated architecture (to enable distributed Cloud/Edge scenarios);
- Full support for multi-VIM FaaS and non-FaaS VNF chaining;
- Full integration with the monitoring framework for 5G-MEDIA;
- Full support for log analysis at scale;
- Integration with the AAA service of 5G-MEDIA to support multitenancy.

We will continue on the path of identifying relevant open source contributions and align our development efforts with practical avenues for results exploitation.