ETSI GR NFV-EVE 025 V6.1.1 (2026-03)

GROUP REPORT

Network Functions Virtualisation (NFV) Release 6; Evolution and Ecosystem; Report on Serverless and other application virtualisation forms in NFV

Disclaimer

The present document has been produced and approved by the Network Functions Virtualisation (NFV) ETSI Industry Specification Group (ISG) and represents the views of those members who participated in this ISG.
It does not necessarily represent the views of the entire ETSI membership.

Reference
DGR/NFV-EVE025

Keywords
cloud, cloud-native, MANO, NFV, virtualisation

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI IPR online database.

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™**, **LTE™** and **5G™** logo are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This Group Report (GR) has been produced by ETSI Industry Specification Group (ISG) Network Functions Virtualisation (NFV).

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The present document provides the background of serverless computing model and other application virtualisation forms (e.g. Unikernel, MicroVM, Wasm and eBPF®), describes the relevant use cases to the NFV framework, investigates how the NFV framework (including the relevant functional entities, functions, and interfaces) can manage deployments based (or relying) on serverless computing model and other application virtualisation forms. It proposes several solutions and recommendations to be considered in the normative phase.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents may be useful in implementing an ETSI deliverable or add to the reader's understanding, but are not required for conformance to the present document.

[i.1] ETSI GR NFV 003: "Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV".

[i.2] Kata-containers documentation on Github.

[i.3] Unikernels documentation.

[i.4] WebAssembly documentation.

[i.5] Runwasi documentation on Github.

[i.6] Corewave documentation on Github.

[i.7] eBPF® documentation.

[i.8] The LLVM Compiler Infrastructure.

[i.9] eBPF Helper functions.

[i.10] ETSI GR NFV-EVE 019 (V5.1.1): "Network Functions Virtualisation (NFV) Release 5; Architectural Framework; Report on VNF generic OAM functions".

[i.11] Firecracker documentation on Github.

[i.12] Cloud Hypervisor documentation on Github.

[i.13] KVM documentation.

[i.14] wasi-sockets documentation.

[i.15] W3C® Candidate Recommendation Draft, 23 March 2026: "WebAssembly Core Specification".

[i.16] wasi-nn documentation.

[i.17] ETSI GS NFV-SOL 016 (V5.1.3): "Network Functions Virtualisation (NFV) Release 5; Protocols and Data Models; NFV-MANO procedures specification".

[i.18] eBPF Maps.

[i.19] OpenFaaS documentation.

[i.20] ETSI GS NFV-IFA 049 (V5.2.1): "Network Functions Virtualisation (NFV) Release 5; Architectural Framework; VNF generic OAM functions and other PaaS Services specification".

[i.21] ETSI GS NFV-IFA 008 (V5.2.1): "Network Functions Virtualisation (NFV) Release 5; Management and Orchestration; Ve-Vnfm reference point - Interface and Information Model Specification".

[i.22] eBPF registers and stack space.

NOTE: Information about eBPF stack is available at: https://www.kernel.org/doc/html/latest/bpf/bpf_design_QA.html#id24.

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the terms given in ETSI GR NFV 003 [i.1] and the following apply:

NOTE: A term defined in the present document takes precedence over the definition of the same term, if any, in ETSI GR NFV 003 [i.1].

runtime resource pool: kind of resource pool, that comprises multiple compute nodes equipped with the installed serverless function execution environment (either the resources are virtualised or physical)

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the abbreviations given in ETSI GR NFV 003 [i.1] and the following apply:

eBPF	extended Berkeley Packet Filter
FaaS	Function as a Service
FRS	FaaS Runtime Software
FRSR	FaaS Runtime Software Repository
JIT	Just In Time
KVM	Kernel-based Virtual Machine
MicroVM	Micro Virtual Machine
VMM	Virtual Machine Monitor
WASI	WebAssembly System Interface
Wasm	Webassembly
XaaS	X as a Service

4 Overview and background

4.1 Overview

Conventional deployment methods based on virtual machines and containers still face challenges such as resource management complexity, startup latency, and security considerations. The present document report explores new computing paradigms like serverless, and a series of cutting-edge virtualisation technologies (e.g. Unikernel, MicroVM, Wasm, and eBPF®), providing more possibilities for the flexible selection of the virtualisation technology and the paradigm (e.g. serverless, PaaS, etc.) that are suitable to support different use cases.

4.2 Serverless and Function as a Service

4.2.1 What is Serverless computing

Serverless computing is a cloud computing model in which the cloud provider allocates cloud infrastructure resources on demand to support the execution of applications. Provisioning, management and maintenance aspects of infrastructure resources required by applications are performed by the cloud providers and abstracted from the application provider, enabling them thus to avoid complex infrastructure management and to focus solely on application development. Serverless has become widely adopted, and many public cloud service providers are offering their own serverless services.

The serverless computing model exhibits a set of characteristics that are broadly acknowledged across the industry:

- Serverless computing model allows application providers to deploy and run their applications without caring for the allocation, management, operation, and maintenance of the infrastructure resources.
- There is no need for upfront provisioning since the function instances can be scaled dynamically and quickly.
- Application providers are charged based on the infrastructure resources used by their applications during the actual running time.

4.2.2 Introduction to FaaS

4.2.2.1 What is FaaS

Function as a Service (FaaS) is a platform providing serverless architecture deployment, orchestration, and management, allowing serverless application providers to develop, run, and manage applications in the form of functions (e.g. components with an input and an output) without the complexity of building and maintaining the underlying cloud infrastructure.

FaaS can be considered as a typical implementation of serverless computing model which is based on an event-driven computing framework. In FaaS a serverless application can be realized as a set of serverless functions.

Under FaaS, if there is no available instance of the function (e.g. no instance or existing instances are overloaded), one or more instances of the serverless function will be deployed to process incoming requests. These instances are ephemeral; when there are no incoming requests, the cloud provider will terminate the instances (depending on implementation, the instances can be terminated immediately or after a certain period).

Microservices define the architecture, guiding the decomposition of large applications into multiple independent services. FaaS provides an ideal platform for event-driven, short-lived tasks within such services. FaaS simplifies the deployment and operation of stateless microservices with short-lifetimes, and serves as a powerful tool for building microservices architectures.

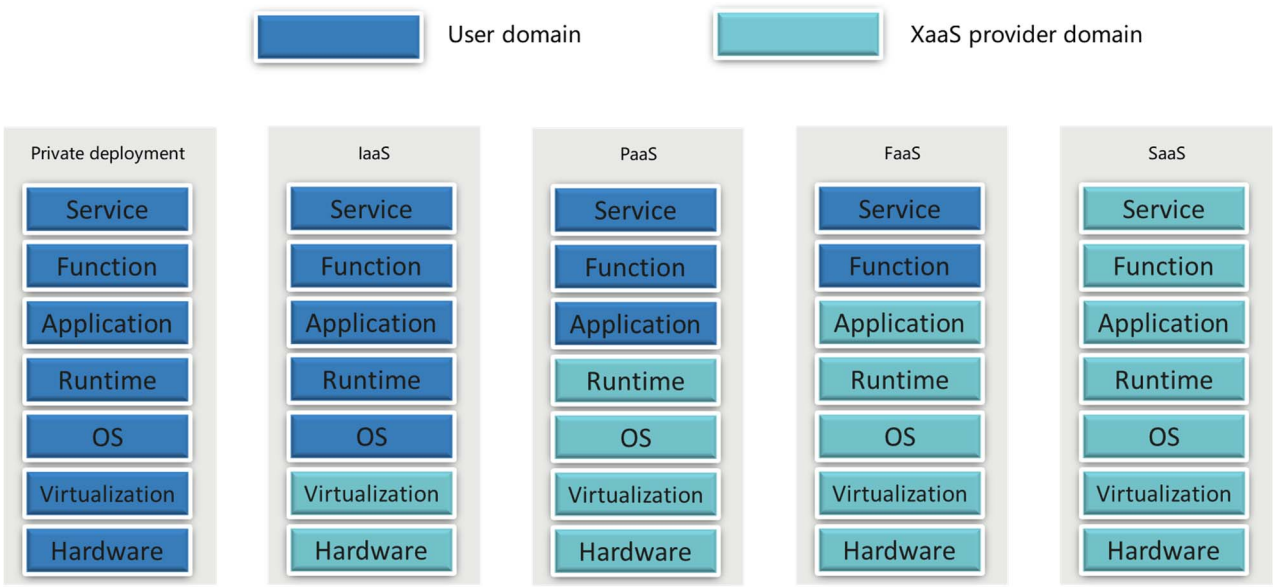

Figure 4.2.2.1-1: XaaS Comparison

In cloud computing, XaaS stands for "Everything as a Service". Figure 4.2.2.1-1 shows the comparison between several XaaS models by means of entities that belong to the user domain and those belonging to the XaaS provider domain. From left to right, the responsibilities of XaaS providers increase gradually, allowing users to focus less on the underlying infrastructure and environment, and more on how to satisfy their customers' requirements. However, as the responsibilities of XaaS providers increases, users are subject to more restrictions.

FaaS is responsible for "stateless computing logic," while backend services handle "stateful backend capabilities". Some common backend services include databases, identity authentication, and message notifications. These are typically provided by a backend service provider to simplify the development of front-end applications. In FaaS, developers do not need to manage these back-end services themselves.

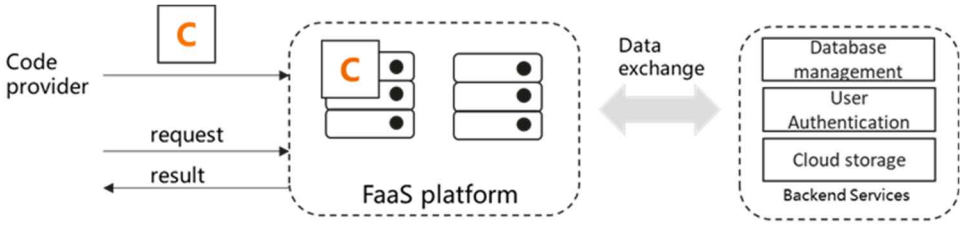

Figure 4.2.2.1-2: FaaS Architecture

Figure 4.2.2.1-2 depicts a typical architecture consisting of FaaS and backend services that is widely used in IT domain to support the implementation of Web service, Real-time file processing, etc. Taking e-commerce applications as an example, users log in and use backend services for authentication. When a user initiates an order request, a FaaS function can be triggered to process inventory deduction. The backend service associated with the database refreshes the order and inventory data.

4.2.2.2 Advantages and challenges of FaaS

FaaS has several advantages:

- No concerns regarding the management of infrastructure resources: compared with the traditional method of provisioning infrastructure resources to users, FaaS enables the infrastructure management to be transparent to the serverless application providers. The latter only need to provide code packages to the FaaS platform and set runtime and triggering events for the functions. This frees serverless application providers from the task of provisioning the needed resources, server management, load balancing, and disaster recovery, etc.

- Fast scaling: FaaS can quickly scale the service instances based on load. Compared with the scaling of traditional cloud services, FaaS can reduce the number of application instances to zero if there are no incoming requests. No resources are consumed when there is no load. In case of high load, if needed, FaaS can scale up/out the service instances in hundreds of milliseconds.

- Pay As You Use: IaaS users are still charged for the infrastructure, even if their applications receive no traffic. In FaaS architecture, serverless application providers are only charged based on what they actually use, which is more cost-effective.

FaaS also presents some challenges:

- Cold start: refers to the initial latency or delay experienced when a function is invoked after a period of inactivity. FaaS allows the number of application instances to be reduced to zero in case there is no application load. Instance creation is triggered by events. As a result, the number of cold starts increases, and the cold start usually takes hundreds of milliseconds to several seconds. The cold start time could be impacted by various factors, e.g. programming language, runtime, memory, code size.

- Vendor lock-in: in FaaS environment, vendor lock-in can occur due to lack of de facto standards and standardization activities. Although open-source solutions like OpenFaaS [i.19] exist, the functions provided by serverless application providers invoke the APIs provided by vendors (most commonly public cloud service providers) to interact with underlying infrastructure resources. Therefore, FaaS is more coupled with vendor implementation specific environments, making it difficult to migrate or deploy functions from one implementation environment to another.

4.2.2.3 FaaS runtime

FaaS runtime refers to the set of software and/or artefacts needed for the execution of user-provided code for FaaS deployments. The specific runtime dependencies are determined by the programming language and the way the code is compiled. In the general case, for interpreted languages such as Python, JavaScript, and Ruby, the code execution needs an interpreter to be installed on the target host beforehand. During execution, the interpreter reads and executes the code line by line. For languages or technologies like Java, Wasm, users typically provide pre-compiled bytecode. In the preparation phase, appropriate runtime software needs to be installed on the target host. In some cases, the runtime software can compile the bytecode into machine code to achieve portability (e.g. for Java or Wasm). One of the advantages of FaaS computing model is that it reduces the complexity for users in particular with regards to the management and operation of virtualised resources. Users only need to provide the code related to the service logic, while the runtime environment is managed and configured by administrators.

4.3 Overview on new virtualisation technologies

4.3.1 Introduction

Existing ETSI NFV specifications studied the usage of VMs and containers as the main virtualisation technologies for deploying network functions in the Cloud.

However, towards supporting a wide range of cloud-native deployments, new virtualisation technologies have become mainstream and are rapidly gaining traction in the field. Each of which has different characteristics, has its advantages, performance capabilities, and limitations, making it more suitable for some use cases and environments than others. In this perspective, investigating these technologies is crucial for moving beyond VMs and containers as the two de facto solutions for telco cloud applications deployment in the cloud. This shift allows for adoption of more appropriate technologies to the specific requirements and constraints of for instance, serverless applications, short-lived NFs, and those with computational and latency constraints. Therefore, it is essential to investigate their operation process and their performance (e.g. memory usage, CPU usage, booting time) to ensure their effective integration into the NFV framework.

The present clause 4.3 aims at providing an overview and a comparison of the following newer technologies:

- MicroVMs;

- Unikernels;

- WebAssembly (Wasm); and

- extended Berkeley Packet Filter (eBPF).

4.3.2 Unikernels

To deploy applications in the Cloud, container-based virtualisation has emerged as a form providing more flexible and smaller resource footprint. However, security is a major concern for the adoption of OS containers.

Unikernels-based virtualisation tackle the security issues that containers suffer from, while also being a lightweight virtualisation technology. Unikernels are single-purpose lightweight VMs where a target application is statically tied to a minimalistic OS that provides the exact needed libraries and dependencies [i.3]. Hence, unikernels enhance resource usage by providing a lean image size and, at the same time, reduce the attack surface by reducing the dependencies. Unlike containers, unikernels guarantee strong isolation and this is without having to go through full hardware virtualisation (as a traditional VM does). Another benefit of unikernels is their quasi-instant boot time. Combined, these factors make unikernels a potential virtualisation technology for short-lived processes (e.g. FaaS).

On the other hand, unikernels can introduce extra complexity for developers , as their debugging can be difficult, as all the utilities and tools that are typically supplied with the OS are in this case missing. Unikernels are also less flexible in comparison to containers or VMs since they are immutable images, removing or adding any functionality means rebuilding the executable image. Finally, unikernels consume more memory compared to containers because the kernel has to be duplicated in every unikernel image.

4.3.3 MicroVMs

4.3.3.1 Overview

Micro Virtual Machines (MicroVMs) are lightweight virtual machines. MicroVMs exclude unnecessary OS components, using only what is needed to run a workload, for instance, they avoid bulky drivers and services. Often, they run on specialized hypervisors ("microvisors") optimized for speed and minimalism. For example, they can be deployed using minimalistic Virtual Machine Monitor (VMM) such as Firecracker® [i.11] or Cloud Hypervisor® [i.12]. Those VMMs most often utilize Kernel-based Virtual Machine (KVM®) [i.13] which is integrated into the Linux® kernel and enables it to act as a type-1 hypervisor. KVM therefore enables microVMs to execute in near bare-metal performance by leveraging the kernel's scheduling and device management. Furthermore, the memory footprint and the attack surface of each microVMs are reduced, because unnecessary devices and guest functionality are excluded.

NOTE 1: Firecracker® is a registered trademark of Amazon Technologies, Inc.

NOTE 2: Firecracker, Cloud Hypervisor, and KVM are examples of suitable products or solutions available commercially or publicly. This information is given for the convenience of users of the present document and does not constitute an endorsement neither by ETSI, nor by the present document, of this product.

NOTE 3: Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

A microVMs implementation is focused on three key aspects:

- **Efficiency and scalability:** currently, in most of the existing implementations the memory overhead is reduced; the overhead per microVM does not exceed few MiB which enables dense deployments of microVMs.
- **Quick deployment:** kernel loading is accelerated and the user space can be initiated in less than 150 ms and the number of launched microVMs can go up to more than 100 microVM per second per host.
- **Security:** due to the minimal number of emulated devices, the attack surface of microVMs is reduced. Additionally, using KVM-based virtualisation can increase the security compared with traditional VMs and guarantees that workloads in a multi-tenant environment can be run securely on the same machine.

4.3.3.2 Usage of MicroVMs

- **Data pipeline as a service:** as the workload running on a VM is more isolated than when running on a container, using microVMs gives the user the possibility to run their own custom steps in isolation of the rest of the pipeline which prevents the risk of the latter being exposed.

- **Creating test environments:** because of their fast deployment and their relatively low requirements in terms of resources, microVMs can be used for creating testing environments.

4.3.3.3 Limitations of MicroVMs

Despite the advantages they provide, MicroVMs have several limitations. For example:

- They support a minimal device model.
- Due to their minimal design, microVMs lack many features of full VMs (e.g. full device support, GUI capabilities).
- MicroVMs can still be slower to start than containers.

4.3.3.4 Example of MicroVMs

This clause discusses kata-containers as an example of microVMs:

Security is a major disadvantage commonly associated to container-based virtualisation. The reason is that containers running on the same host share the same OS kernel, and in multi-tenant environment involving trusted/untrusted workloads, this can translate to security vulnerabilities.

Kata-containers offer a unified solution that combines the benefits of both VMs and containers. A kata-container encapsulates container's workloads in a microVMs; therefore, each container can boot as a lightweight VM with its own dedicated kernel [i.2]. By doing so, kata-containers not only offer the advantages of containers, but also prevent failures and malicious behaviours achieving thus security and isolation on a par with those of VMs.

Kata-containers present some challenges, for instance, kata-containers are wrapping containers' workloads in microVMs, hence they suffer from the overhead caused by the hypervisor which means that, unlike traditional containers, kata-containers keep the CPU busy, however, their usage of CPU is lower than that of traditional VMs. Additionally, the overhead imposed by the hypervisor makes kata-containers face a penalty in terms the service throughput which is lower than that of unikernels or traditional containers. Taken as a whole, kata-containers offer a good trade-off between security and efficiency.

4.3.4 WebAssembly

WebAssembly, or Wasm for short, is a binary instruction format for a stack-based virtual machine [i.4]. It was originally designed to provide an efficient and secure way to deploy and run applications on the Web. Because of its high performance, the mainstream browsers supports Wasm.

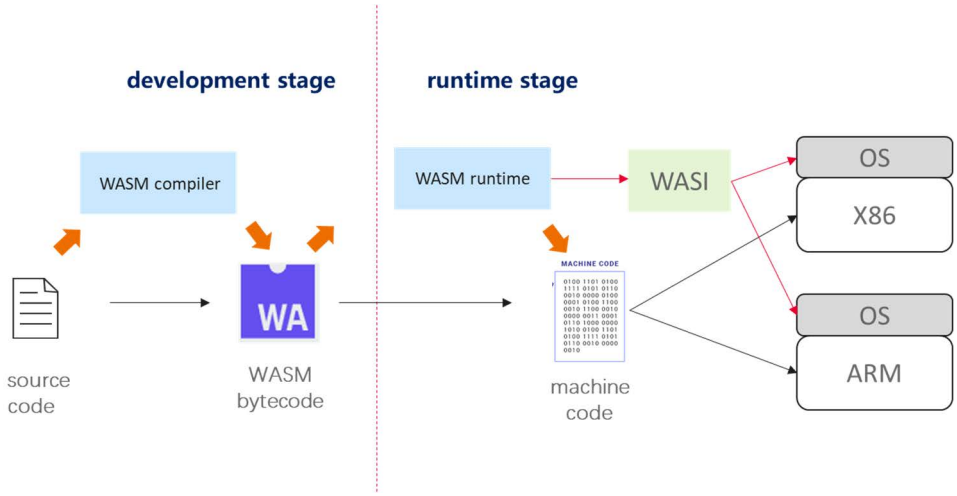

Figure 4.3.4-1: Wasm workflow

Figure 4.3.4-1 illustrates the Wasm workflow. Code written in different programming languages can be compiled into Wasm binary bytecode. At the target platform, the Wasm runtime (e.g. Wasmtime, WasmEdge, etc.) can load, parse, and instantiate the Wasm binary bytecode.

Whilst Wasm was designed for the web, its features such as its compact binary format, running in a sandboxed environment, etc., made it suitable for running applications in non-web environments too (e.g. edge computing, serverless computing, IoT devices, etc.). In more detail, the Wasm programs are organized into modules that constitute the distributable, loadable, and executable unit of code [i.15]. By default, these modules are intended to be sandboxed for security and portability, i.e. they execute in a restricted environment with limited access to resources, a requirement in web browsers to prevent malicious behaviour. However, in non-web environments, this can prove restrictive, as Wasm modules need access to system resources (e.g. files, networking, acceleration hardware, etc.).

To allow this access, WASI has been designed. WASI is a standard API, providing a POSIX-inspired set of functionalities, but tailored specifically to enable Wasm modules to interact with system resources (refer to figure 4.3.4-1). The modules use WASI calls to request resources, while the Wasm runtime manages them, translating the requests into system operations.

For a WASI-enabled module (i.e. compiled with instructions to call WASI APIs) to operate in a non-web environment, a runtime providing WASI APIs implementations needs to be installed on the host machine. The runtime does not always implement all the necessary standardized interfaces needed by the Wasm module to access some system resources or external functionalities. For instance, WASI itself does not feature direct support for GPU acceleration, in which case, extensions or plugins can be added to the runtime to enable these additional capabilities. Examples of extensions could be wasi-nn extension [i.16] to run AI/ML processes on GPU, or wasi-sockets extension [i.14] to grant the Wasm module rights to open sockets.

Therefore, information about the choice of the runtime to use to instantiate a Wasm module as well as (if necessary) the extensions and/or the plugins are required to be supplied with the Wasm module.

Wasm has the following characteristics:

- High portability: Wasm development tools can compile source code written in different programming languages into Wasm bytecode, which can be translated by Wasm runtime into machine code running on multiple operating systems (such as Linux, Windows, macOS, Android, etc.) and physical instruction sets (e.g. X86, ARM, RISC-V, etc.).
- High security: each Wasm instance has its own memory space. The Wasm sandbox can monitor and check all the memory access to prevent out-of-bounds memory access. Wasm sandbox isolation makes the code only interact with the outside world through explicitly declared import/export and WASI, providing isolation between the neighbouring Wasm applications running on the same platform.
- High performance: Wasm is a statically strongly typed low-level language. Wasm code can run at near-native speeds on different platforms. Wasm can achieve millisecond-level cold start time and low resource consumption.
- Small image size: comparing with OS container image, the Wasm image mainly consists of Wasm bytecode, without including dependencies and libraries, which makes the Wasm image smaller.

4.3.5 eBPF

4.3.5.1 Introduction

In the Linux OS, the kernel space has full access to hardware resources (memory, CPU, network, etc.) and only the most reliable codes are executed in the kernel. The user space, on the other hand, is where standard applications are running, and it is given restricted access to the hardware. For applications running in user space to access hardware resources, they need to specify their requirements using the kernel's "system calls" API. The system call interface is sufficient, however, in some cases developers might need more flexibility to implement new functionalities. extended Berkeley Packet Filter (eBPF) [i.7] emerged as a new mechanism that provides this flexibility by making it possible to load and execute bytecode in the kernel space. Linux Kernel Modules (LKMs) can also provide similar levels of flexibility, they can be loaded directly to the kernel; however, they are inherently hard to maintain since they need to be adapted to the changes that occur with each kernel revision, potentially breaking compatibility and causing the kernel to crash. Using eBPF programs reduce the risk of kernel crashes and incompatibility concerns since they are meant to be portable across kernel versions.

4.3.5.2 How eBPF works

eBPF allows sandbox programs to run within the OS. The program can be written and then compiled using tools such as LLVM [i.8]. Once loaded into the kernel, a verifier guarantees that the program can be executed safely, otherwise it is rejected. This verification is carried out by running the eBPF program in a virtual machine. A series of checks are then performed by the verifier. It ensures that the program completes its execution without any issues which would cause the kernel to crash. This is done by tracing the execution path an eBPF program might follow when running in the kernel. Following verification, the eBPF program will be translated to machine specific instruction set using Just-in-Time (JIT) compilation and attached to a hook point which, when triggered by specific events, results in the execution of the eBPF program. In fact, eBPF programs are event-driven, they are triggered by kernel hooks (system calls, network events, etc.). eBPF programs rely on maps (key-value pairs) to record the states as well as to communicate data to applications within the user space. eBPF programs use helper functions to store and fetch data in/from maps [i.18]. Helper functions serve many other various purposes, they are functions defined by the kernel and can be invoked by eBPF programs (for example, to get information about the current state of CPU, or information about the processes for which the program is invoked, etc.) [i.9]. Figure 4.3.5.2-1 represents eBPF workflow.

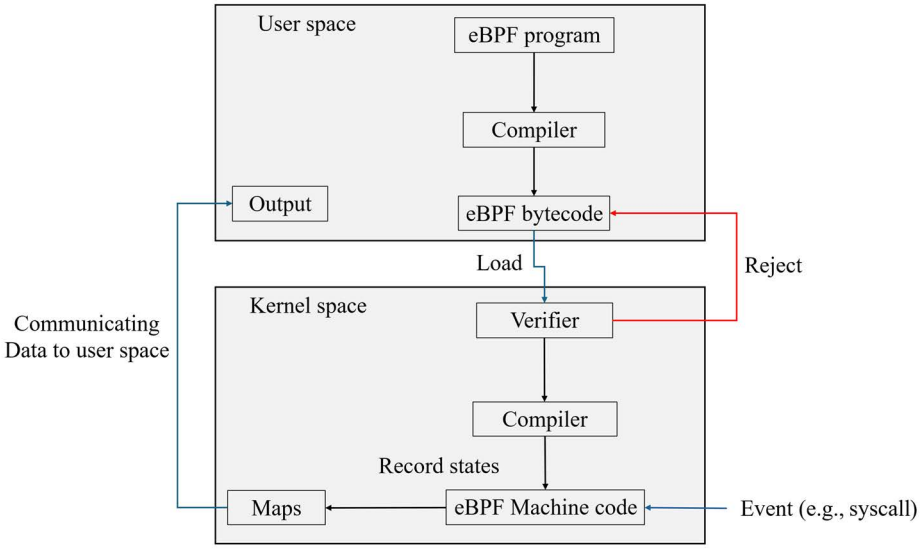

Figure 4.3.5.2-1: eBPF workflow

4.3.5.3 eBPF for VNF deployments in the telco Cloud

Numerous possible scenarios can be envisioned for the use of eBPF technology to facilitate VNF operations in the telco cloud. The following are some examples, and more tailored use cases applied to NFV can be envisioned:

- eBPF for network function related processing:

  Depending on the virtualisation technology in effect, transferring packets from the kernel space to the user space where the VNF resides entails a significant cost. eBPF can help overcome these issues by enabling the deployment of a part of the VNF in the kernel using an eBPF program that can be directly bounded to the network interface, receive packets and perform pre-processing. The packet can then be forwarded to the other part of the VNF for further processing.

- eBPF for observability:

  Kubernetes observability and security tools used the sidecar-proxy model prior to eBPF, where for each Pod a sidecar container is deployed to monitor the Pod (see clause 4.4.1 in ETSI GR NFV-EVE 019 [i.10] for a description of use cases related to the use of sidecar-proxies in NFV). A more efficient approach to monitor pods is to bundle observability functionalities needed by the Pods together in a single eBPF-based agent and run it in the kernel. This can prevent the consumption of additional resources induced by the use of sidecars.

4.3.5.4 eBPF limitations

Though eBPF enables programmability within the kernel, there are limits to its flexibility and capacity, so as to guarantee kernel stability and security. The size of an eBPF program is limited; programs need to meet system size requirements. Therefore, it is not possible to load eBPF programs of any desired size. Additionally, the program in question needs to be of finite complexity. The verifier will assess all potential execution paths and needs to finish the analysis within the specified complexity limit otherwise the program will be rejected [i.7]. Moreover, eBPF is designed for CPU execution but not for direct execution on GPUs.

4.3.6 Comparison of different virtualisation technologies

The integration of emerging virtualisation technologies into the NFV system aims at simplifying the latter by further facilitating the decoupling of software and hardware, thus increasing the portability of NFs. It also seeks to bring more modularity to NFV, so that NFs may be scaled and adapted easily as required, enabling the building of more robust networks.

The present clause provides a comparison of these technologies in terms of their architecture (refer to figure 4.3.6-1), their performance (e.g. security, portability, image size, booting time, etc.), their suitability to various telecom network use cases, and level of complexity for developers (refer to table 4.3.6-1).

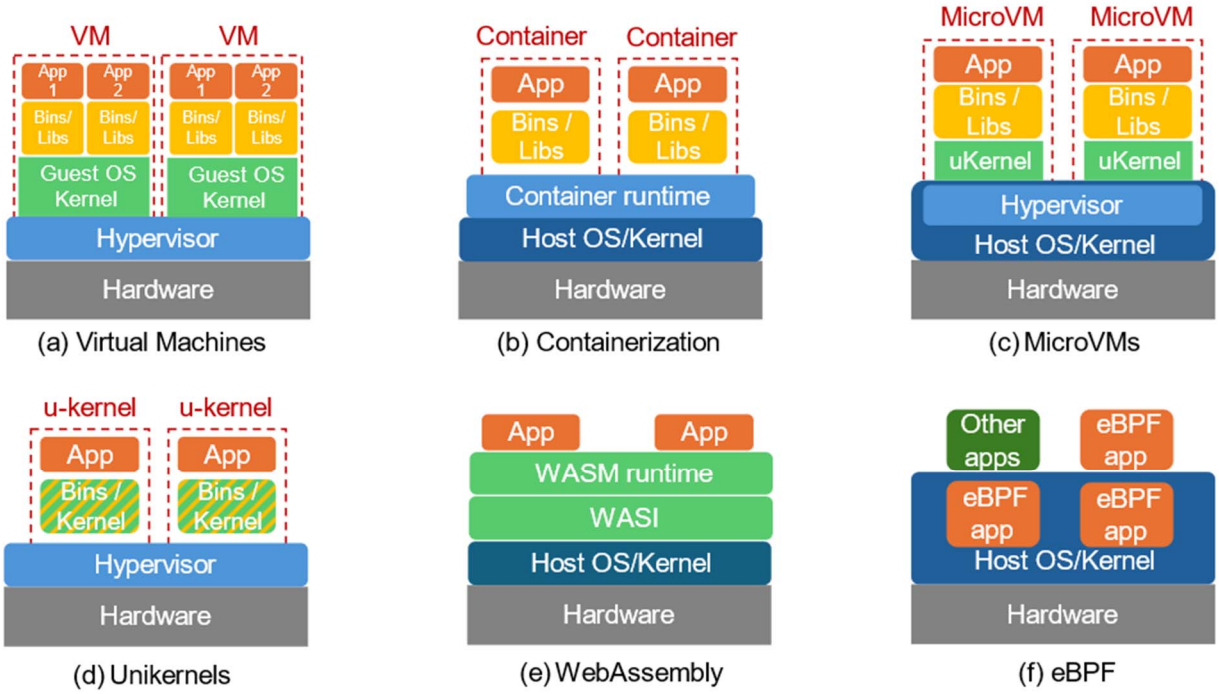

Figure 4.3.6-1: Comparison of high-level architecture

Table 4.3.6-1 describes the main characteristics and potential usages of the virtualisation technologies analysed in the present document.

Table 4.3.6-1: Virtualisation technologies: characteristics and use cases

Technology	Key Characteristics	Potential use cases
VM	• Hardware level virtualisation. • Runs on a hypervisor and every VM uses its own OS. • Heavy image size and slow boot-up time (minutes): a VM needs to load an entire OS each time it starts, and the hypervisor needs to manage the overhead caused by emulating hardware components. • High memory and CPU footprint. • By means of security, full isolation provided. However, the unnecessary emulated devices and the usage of the hypervisor can potentially increase the attack surface. • Easily portable: cross platform software support, cross platform hardware emulation. • Low resource efficiency as the VMs may not use all the allocated resources.	• Complex NFs such as RAN or CN functions. • NFs that need perpetual computational resources. • Running untrusted workloads and/or workloads that require high security level.
Container	• OS-level virtualisation. • Requires host OS and uses the host's kernel. • Relatively light image size and fast boot-up time (seconds). • Containers share the underlying OS kernel which eliminates the need for a dedicated OS instance per container. • Relatively low memory and CPU footprint. • By means of security, containers share the host kernel, and the provided isolation is a process level isolation which makes them less secure. However, the attack surface is relatively reduced since only the necessary dependencies are included. • Containers are highly portable on the same type of platform but cannot run on a different platform (e.g. Docker® containers cannot run on Linux Containers platform). If the NF inside the container requires a special system call that the host is compiled without it then the portability might be difficult. • Efficient resources utilization.	• Complex NFs such as RAN or CN functions. • Suitable for microservices and Cloud native functions. • Lightweight applications at the edge.
MicroVM	• Hardware-level virtualisation. • Requires a host OS and uses dedicated kernel per microVM. • Relatively low memory and CPU footprint. • Fast boot-up time (milliseconds). • By means of security, microVMs offer strong isolation as each microVM runs its own independent, minimalist guest kernel. Additionally, the attack surface is reduced due to the minimal number of emulated devices. • Efficient resources utilization.	• Lightweight applications. • Short-lived NFs. • Pay-as-use NFs.
Unikernel	• Hardware-level virtualisation. • Runs on hypervisor and uses built-in kernel libraries. • Relatively light image size and fast boot-up time (milliseconds). • Low memory and CPU footprint. • By means of security, unikernels provide strong isolation comparable to that provided by VMs without the need for full hardware virtualisation. The attack surface is reduced by reducing the dependencies. • Easily portable. • Efficient resource utilization.	• Lightweight single process applications (e.g. edge applications, packet processing, etc.). • Application with high context switching between user and kernel mode. • Short-lived NFs (e.g. FaaS).
Wasm	• Requires host OS. • Uses host kernel. • Light image size and fast boot-up time (seconds to milliseconds). • Low memory and CPU footprint. • By means of security, security check is performed during the compilation and running. Furthermore, Wasm code can only interact with the outside through WASI which provides isolation between Wasm applications hosted on the same platform. • Wasm enables compiling the code into bytecode which can run on different OSs and physical instruction sets. • Efficient resources utilization.	• Edge applications. • IoT, AI applications.

Technology	Key Characteristics	Potential use cases
eBPF	• Requires host OS. • Uses host kernel. • Light image size and fast boot-up time (milliseconds). • Low memory and CPU footprint. • By means of security, a verifier guarantees that an eBPF program can be executed safely in the kernel. • eBPF programs are portable across kernel versions and are updateable (no need for nodes reboots). • eBPF program size and complexity are limited to guarantee the stability and the security of the kernel. • Efficient resources utilization.	• Lightweight applications. • Packet tracing. • Observability. • Networking. • Security. • Potential deployment of NFs in the kernel, e.g. in the case of in-network computing.
NOTE: Docker® is a registered trademark of Docker, Inc. in the U.S. and other countries.		

The industry is gradually adopting these technologies for the deployment of applications. Several projects have been launched to facilitate the orchestration of these technologies; for example, Kubernetes® has been extended to also orchestrate Wasm applications (runwasi containerd shim) [i.5]. Other projects aim at simplifying application deployment on MicroVMs such as Corewave [i.6]. Furthermore, exploring different combinations of these technologies (e.g. eBPF and containers) is essential to maximize their advantages in addressing specific requirements.

5 Use cases

5.1 Overview

Clause 5 presents several use cases related to serverless and new virtualisation technologies. These use cases involve processes related to the lifecycle management of applications or VNFs deployed as serverless applications or VNFs using new virtualisation technologies.

5.2 Use case #1: Onboarding a serverless application package

5.2.1 Introduction

The serverless application provider can on-board the serverless application package to the NFV-MANO system. After successful validation, the serverless application package is stored in the package repository and managed by the NFV-MANO. This use case describes the process.

A serverless application descriptor included in the serverless application package describes some elements that are essential for deploying and maintaining the serverless application, such as: runtime related information, triggering conditions, threshold of instance number (the threshold of serverless function instances number can prevent resource exhaustion), etc.

The package of serverless application can include or reference the function code. The function code can be provided in various forms, e.g. a file, a zip file, or through access information allowing to access an external code repository, etc. It can be stored in an internal repository that can be accessed by NFV-MANO system.

A serverless application is comprised of one or more serverless functions. For the sake of convenience, it is allowed to on-board a serverless function package in order to update one of the functions constituting the serverless application. The use case regarding on-boarding a serverless function package is described in the following clause.

5.2.2 Actors and roles

Table 5.2.2-1 describes the use case actors and roles.

Table 5.2.2-1: Use case #1 actors and roles

#	Actor and role	Description
1	Serverless application provider	An application provider (or an application developer) wanting to deploy a serverless application using infrastructure provided by the operator.
2	NFV-MANO system	Supports receiving requests from the application providers and managing the serverless application packages.
3	Package repository	A repository is used to store the serverless application package and function code. It can be accessed by NFV-MANO components.

5.2.3 Trigger

Table 5.2.3-1 describes the use case triggers.

Table 5.2.3-1: Use case #1 trigger

Trigger	Description
1	A service provider plans to onboard a package of the serverless application in the NFV system.

5.2.4 Pre-conditions

Table 5.2.4-1 describes the use case pre-conditions.

Table 5.2.4-1: Use case #1 pre-conditions

#	Pre-condition	Description
1	NFV-MANO is operational.	No additional description.
2	Package repository is operational.	No additional description.

5.2.5 Post-conditions

Table 5.2.5-1 describes the use case post-conditions.

Table 5.2.5-1: Use case #1 post-conditions

#	Post-condition	Description
1	The serverless application package and function code have been stored in the package repository.	NA

5.2.6 Flow description

Table 5.2.6-1 describes the use case flow.

Table 5.2.6-1: Use case #1 flow description

#	Actor/Role	Action/Description
Begins when	Serverless application provider -> NFV-MANO system	The serverless application provider provides a serverless application package to the NFV-MANO system. The package includes the serverless application descriptor and function packages that includes or references the serverless function(s) code constituting the serverless application.
1	NFV-MANO system	The NFV-MANO system checks the integrity of the package. A successful check means that the serverless application descriptor can be used to deploy the function instances of the serverless application.
2	NFV-MANO system -> Package repository	The NFV-MANO system stores the serverless application package in the package repository. The NFV-MANO system downloads the serverless function code according to the accessing information in the serverless application descriptor or copies the function code files from the serverless application package directly, and stores the serverless function code in the package repository.
3	NFV-MANO system -> Service provider	The NFV-MANO system sends the serverless application package on-boarding success response to the service provider.

5.3 Use case #2: On-boarding a serverless function package

5.3.1 Introduction

A serverless application can be realized as a set of serverless functions. After on-boarding the serverless application package, the application provider can further on-board a serverless function package to update the code and/or configuration of a specific function within that serverless application. The use case describes how to on-board the package of a serverless function.

The NFV-MANO system is responsible for checking the integrity of the package. The package of serverless function can include or reference the function code that can be stored in the package repository by the NFV-MANO system.

5.3.2 Actors and roles

Table 5.3.2-1 describes the use case actors and roles.

Table 5.3.2-1: Use case #2 actors and roles

#	Actor and role	Description
1	Serverless application provider	An application provider (or an application developer) wanting to deploy a serverless function, that is part of their application, using infrastructure provided by the operator.
2	NFV-MANO system	Supports receiving requests from the application providers and managing the function packages.
3	Package repository	A repository is used to store the serverless functions' package which include the code to be executed. It can be accessed by NFV-MANO components.

5.3.3 Trigger

Table 5.3.3-1 describes the use case triggers.

Table 5.3.3-1: Use case #2 trigger

Trigger	Description
1	A service provider plans to onboard a serverless function package to update the serverless function in the NFV system.

5.3.4 Pre-conditions

Table 5.3.4-1 describes the use case pre-conditions.

Table 5.3.4-1: Use case #2 pre-conditions

#	Pre-condition	Description
1	NFV-MANO is operational.	No additional description.
2	Package repository is operational.	No additional description.
3	The serverless application package has been onboarded successfully in the NFV-MANO system.	No additional description.

5.3.5 Post-conditions

Table 5.3.5-1 describes the use case post-conditions.

Table 5.3.5-1: Use case #2 post-conditions

#	Post-condition	Description
1	The NFV-MANO system onboards the serverless function package	

5.3.6 Flow description

Table 5.3.6-1 describes the use case flow.

Table 5.3.6-1: Use case #2 flow description

#	Actor/Role	Action/Description
Begins when	Serverless application provider -> NFV-MANO system	The serverless application provider provides a serverless function package to the NFV-MANO system. The package includes the serverless function descriptor and includes or references its code.
1	NFV-MANO system	The NFV-MANO system checks the integrity of the package. A successful check means that the serverless function descriptor can be used to deploy the function instances.
2	NFV-MANO system -> Package repository	The NFV-MANO system stores the serverless function package in the package repository. The NFV-MANO system downloads the serverless function code according to the accessing information in the serverless function descriptor available in the package stored in the repository or copies the function code files from the serverless function package directly, and stores the function code in the package repository.
3	NFV-MANO system -> Serverless application provider	The NFV-MANO system sends the serverless function package on-boarding success response to the serverless application provider.

5.4 Use case #3: Instantiating a serverless function

5.4.1 Introduction

NFV-MANO is in charge of serverless function LCM. The NFV-MANO can associate triggering events with the appropriate serverless functions. In serverless computing, a triggering event is an external or internal signal that initiates the execution of a serverless function. In case resource requirements are provided in the function package (either in the descriptor or in manifests or other artifacts), NFV-MANO allocates the amount of infrastructure resources needed.

An entity can continuously monitors the occurrence of triggering events that may drive the instantiation of a serverless function. When one of the triggering conditions(see note) is satisfied, the NFV-MANO checks whether an available running function instance exists. If the NFV-MANO determines that it is necessary to create a new function instance, it triggers the function instantiation.

NOTE: Triggering conditions described in the application descriptor refer to the rules that define which specific triggering events are eligible to invoke a particular serverless function. It acts as a "filter" or "matching logic" between triggering events and functions.

5.4.2 Actors and roles

Table 5.4.2-1 describes the use case actors and roles.

Table 5.4.2-1: Use case #3 actors and roles

#	Actor and role	Description
1	Serverless Function Management	The Serverless Function Management supports monitoring corresponding event and controlling the scaling process of serverless function. It is also responsible for serverless function LCM.
2	Infrastructure Management	The Infrastructure Management component is responsible for allocating and managing the infrastructure resources.
3	Package repository	A repository used to store the serverless function package which includes the code to be executed. It can be accessed by the NFV-MANO components.

5.4.3 Trigger

Table 5.4.3-1 describes the use case triggers.

Table 5.4.3-1: Use case #3 trigger

Trigger	Description
1	One of the trigger conditions is satisfied for the instantiation of the serverless function instance(s).

5.4.4 Pre-conditions

Table 5.4.4-1 describes the use case pre-conditions.

Table 5.4.4-1: Use case #3 pre-conditions

#	Pre-condition	Description
1	Instance(s) of the serverless functions are deployed.	The Serverless Function Management is monitoring the trigger conditions. Infrastructure resources are available for the scale out of the same serverless function.

5.4.5 Post-conditions

Table 5.4.5-1 describes the use case post-conditions.

Table 5.4.5-1: Use case #3 post-conditions

#	Post-condition	Description
1	NFV system creates the serverless function instance(s)	NA.

5.4.6 Flow description

Table 5.4.6-1 describes the use case flow.

Table 5.4.6-1: Use case #3 flow description

#	Actor/Role	Action/Description
Begins when	Serverless Function Management	The Serverless Function Management finds that one of the trigger conditions is satisfied, e.g. detecting the occurrence of an event or receiving a message from an external entity (e.g. OSS) that requires instantiation of the serverless function.
1	Serverless Function Management	The Serverless Function Management evaluates the need of a new serverless function instantiation. If an instance is already available, jump to step 6. Otherwise, continue to instantiate the serverless function.
2	Serverless Function Management <-> Package repository	The Serverless Function Management fetches the serverless function descriptor and other related information from the package repository.
3	Serverless Function Management -> Infrastructure Management	The Serverless Function Management determines the amount of resources needed for the serverless function instantiation and communicate it to the Infrastructure Management. See note.
4	Infrastructure Management -> Serverless Function Management	The Infrastructure Management reserves the requested resources to deploy the serverless function instance and informs the Serverless Function Management.
5	Serverless Function Management	The Serverless Function Management deploys the serverless function instances.
6	Serverless Function Management	The Function Management sends the data to be processed to the serverless function instance, e.g. the message received from the external network.
NOTE: Requirements related to resources can be also included in the serverless function package, either in the descriptor or into manifest files. In that case the Serverless Function Management is responsible to interact with the Infrastructure Management service to grant resources.		

NOTE: In this use case, LCM operations related to the serverless functions are, for the time being, consider to be performed by a logical NFV-MANO component, which is the Serverless Function Management. Similarly, package management operations are considered to be performed by a logical NFV-MANO entity. These capabilities can be in the future considered as standalone function/functionals block or they can be extra capabilities added to the VNFM or NFVO.

5.5 Use case #4: Deleting a serverless function instance

5.5.1 Introduction

One of the characteristics of the serverless computing model is the efficient usage of resources. Thus, if no associated events are received for a determined duration and the serverless function instance is not involved in processing any workload, the serverless function instance can be deleted and the related infrastructure resources can be reallocated automatically. The timeframe for deleting serverless function instances can be set by the serverless application providers or pre-configurated by the operators.

5.5.2 Actors and roles

Table 5.5.2-1 describes the use case actors and roles.

Table 5.5.2-1: Use case #4 actors and roles

#	Actor and role	Description
1	Serverless Function Management	The Serverless Function Management is responsible for observing the workload of serverless function instances.
2	Infrastructure Management	The Infrastructure Management is responsible for managing and recycling the infrastructure resources.

5.5.3 Trigger

Table 5.5.3-1 describes the use case triggers.

Table 5.5.3-1: Use case #4 trigger

Trigger	Description
1	The serverless function instance has not received any events within the defined time period (or timeframe).

5.5.4 Pre-conditions

Table 5.5.4-1 describes the use case pre-conditions.

Table 5.5.4-1: Use case #4 pre-conditions

#	Pre-condition	Description
1	A timer has been set by the Serverless Function Management for the serverless function instance.	When the Serverless Function Management creates a serverless function instance, it can start a timer for monitoring the instances' load. The timer is based on a predetermined value that specifies the time duration after which the instance can be terminated/deleted if no triggering events are received.
2	The serverless function instance is running normally.	The function instance has not processed any load within a certain period.

5.5.5 Post-conditions

Table 5.5.5-1 describes the use case post-conditions.

Table 5.5.5-1: Use case #4 post-conditions

#	Post-condition	Description
1	The serverless function instance is deleted.	All the related infrastructure resources have been recycled.

5.5.6 Flow description

Table 5.5.6-1 describes the use case flow.

Table 5.5.6-1: Use case #4 flow description

#	Actor/Role	Action/Description
Begins when	Serverless Function Management	The Serverless Function Management monitors the time-out timer of the serverless function or receives a time-out event indicating that the function instance did not receive/process any load within a specified time period.
1	Serverless Function Management	The Serverless Function Management terminates the function instance(s).
2	Serverless Function Management -> Infrastructure Management	The Serverless Function Management requests the Infrastructure Management to recycle the infrastructure resources related to the function instance.
3	Infrastructure Management -> Serverless Function Management	The Infrastructure Management recycles the infrastructure resources related to the function instance.

5.6 Use case #5: Onboarding the package of a VNF with eBPF® components

5.6.1 Introduction

This use case describes the onboarding procedure for the VNF package of a VNF with eBPF components. The VNF package contains or reference the eBPF program(s).

The VNF can be designed to perform observability or networking tasks, entirely composed of eBPF program(s), or it can be a VNF not specifically aimed at such tasks but partially constructed from eBPF program(s) (e.g. the eBPF part serves to perform packet filtering at the kernel level prior to transmitting them to the other part of the VNF within user space).

5.6.2 Actors and roles

Table 5.6.2-1 describes the use case actors and roles.

Table 5.6.2-1: Use case #5 actors and roles

#	Actor and role	Description
1	Operator	A human being or an organization seeking to deliver network services to end-users.
2	OSS	The entity that is managing the VNFs through NFV-MANO. It receives the operator's request for the initiation or termination of the VNFs.
3	NFV-MANO system	Network management and orchestration framework comprising NFVO(s), VNFM(s), VIM(s), etc. It is in charge of applications, VNFs, and NSs lifecycle management and OAM support.

5.6.3 Trigger

Table 5.6.3-1 describes the use case trigger.

Table 5.6.3-1: Use case #5 trigger

Trigger	Description
1	The use case is triggered when an Operator requests to onboard the VNF package of a VNF with eBPF components.

5.6.4 Pre-conditions

Table 5.6.4-1 describes the use case pre-conditions.

Table 5.6.4-1: Use case #5 pre-conditions

#	Pre-condition	Description
1	The NFV-MANO system is operational.	
2	The VNF provider delivers the VNF descriptor.	The eBPF program(s) are contained or referenced in the VNF package. The VNF descriptor is contained in the VNF package.

5.6.5 Post-conditions

Table 5.6.5-1 describes the use case post-conditions.

Table 5.6.5-1: Use case #5 post-conditions

#	Post-condition	Description
1	The VNF Package is onboarded and is available for the VNF with eBPF components instantiation.	

5.6.6 Flow description

Table 5.6.6-1 describes the use case flow.

Table 5.6.6-1: Use case #5 flow description

#	Actor/Role	Action/Description
1	Operator -> OSS	The operator request, via the OSS to onboard the VNF package of a VNF with eBPF component(s).
2	OSS -> NFV-MANO system	OSS sends to the NFVO a create VNF package information request.
3	NFV-MANO system	Creates the individual VNF Package resource.
4	OSS -> NFV-MANO system	The OSS provides the VNF package to NFVO and requests the onboarding. See note.
5	NFV-MANO system	Proceeds with the onboarding, i.e. processing the VNF package content, checking the VNFD. The NFVO stores the VNF package contents in its repository (e.g. storing eBPF programs in an eBPF program repository, etc.).
6	NFV-MANO system -> OSS	The NFV-MANO system notifies the OSS that the VNF package has been successfully onboarded.
NOTE: See ETSI GS NFV-SOL 016 [i.17], clause 5.1 for a detailed description of the VNF package onboarding process.		

5.7 Use case #6: instantiating a VNF as an eBPF program

5.7.1 Introduction

An eBPF program can be compiled into a bytecode and deployed within the kernel (see clause 4.3.5), however, it is executed in a restricted environment enforced by the kernel verifier, which is agnostic to the program characteristics. Furthermore, the eBPF program needs to be attached to the appropriate kernel hooks. Also, eBPF programs call helpers to perform specific tasks. Helpers are predefined kernel functions acting as interfaces between the eBPF program and the kernel. It is therefore essential to consider these dependencies for the deployment of the eBPF programs. This use case describes how the instantiation of a VNF that is provided as an eBPF program (or a set of eBPF programs) is performed taking into account the points mentioned above.

5.7.2 Actors and roles

Table 5.7.2-1 describes the use case actors and roles.

Table 5.7.2-1: Use case #6 actors and roles

#	Actor and role	Description
1	Operator	A human being or an organization seeking to deliver network services to end-users.
2	OSS	The entity that receives the operator's request for the initiation or termination of VNFs and NSs.
3	NFV-MANO system	Network management and orchestration framework comprising NFVO(s), VNFM(s), VIM(s), etc. It is in charge of applications, VNFs, and NSs lifecycle management and OAM support.
4	NFVI	Includes the physical and the virtualised resources as well as the necessary components for the deployment of eBPF programs. The NFVI provides the required resources for the deployment and operation of VNFs/NSs instances.

5.7.3 Trigger

Table 5.7.3-1 describes the use case trigger.

Table 5.7.3-1: Use case #6 trigger

Trigger	Description
1	The use case is triggered when an Operator requests to instantiate a VNF provided as an eBPF program or as a set of eBPF programs.

5.7.4 Pre-conditions

Table 5.7.4-1 describes the use case pre-conditions.

Table 5.7.4-1: Use case #6 pre-conditions

#	Pre-condition	Description
1	The NFV-MANO system is operational.	
2	The eBPF program (or the set of eBPF programs) making up the VNF, as well As the VNFD containing the reference to the eBPF(s) program(s) are available.	

5.7.5 Post-conditions

Table 5.7.5-1 describes the use case post-conditions.

Table 5.7.5-1: Use case #6 post-conditions

#	Post-condition	Description
1	The VNF is instantiated successfully using the kernel(s) of host machine(s) in the NFVI-PoP, and it is ready for subsequent lifecycle management by the NFV-MANO system.	

5.7.6 Flow description

Table 5.7.6-1 describes the use case flow.

Table 5.7.6-1: Use case #6 flow description

#	Actor/Role	Action/Description
1	Operator -> OSS	The operator request, via the OSS to deploy a VNF comprising eBPF program(s).
2	OSS -> NFV-MANO system	The OSS requests the deployment of the VNF based on the related VNFD. The request can include specific requirements regarding the kernel version, the available hooks and helpers in the kernel, etc.
3	NFV-MANO system <-> NFVI	The VNFM in the NFV-MANO checks the VNFD and the requirements therein, the resources capacity provisioned by the NFVI, and decides for an appropriate placement of the eBPF program(s) of the VNF.
4	NFV-MANO system <-> NFVI	Proceeds with the instantiation of the VNF. Upon completion of the eBPF program(s) verification by the kernel verifier, a notification of successful deployment (step 5) or rejection (with error message) is returned to the NFV-MANO system (step 6).
5	NFV-MANO system -> OSS	If the eBPF program(s) is successfully deployed the NFV-MANO system notifies the OSS that the VNF has been instantiated successfully.
6	NFV-MANO system -> OSS	If the eBPF program(s) is rejected by the verifier the NFV-MANO system notifies the OSS.

5.8 Use case #7: Onboarding the VNF package of Wasm-based VNF

5.8.1 Introduction

A description of the Wasm technology is provided in clause 4.3.4. This use case describes the onboarding process of VNF package of a Wasm-based VNF. The package contains information about the choice of the runtime to use to instantiate a Wasm module as well as (if necessary) the extensions and/or the plugins that are required to be supplied with the Wasm module.

The VNF package includes the VNFD of the Wasm-based VNF, the Wasm application code and the additional metadata information mentioned above. The runtime software and the extensions/plugins software could be included in the package or referenced and they can be stored by the NFV-MANO system in a Wasm software repository.

5.8.2 Actors and roles

Table 5.8.2-1 describes the use case actors and roles.

Table 5.8.2-1: Use case #7 actors and roles

#	Actor and role	Description
1	Operator	A human being or an organization seeking to deliver network services to end-users.
2	OSS	The entity that is managing the VNFs through NFV-MANO. It receives the operator's request for the initiation or termination of the Wasm-based VNFs.
4	NFV-MANO system	Network management and orchestration framework comprising NFVO(s), VNFM(s), VIM(s), etc. It is in charge of VNFs and NSs lifecycle management and OAM support.

5.8.3 Trigger

Table 5.8.3-1 describes the use case trigger.

Table 5.8.3-1: Use case #7 trigger

Trigger	Description
1	The use case is triggered when an Operator requests to onboard the VNF package of a Wasm-based VNF.

5.8.4 Pre-conditions

Table 5.8.4-1 describes the use case pre-conditions.

Table 5.8.4-1: Use case #7 pre-conditions

#	Pre-condition	Description
1	The NFV-MANO system is operational.	
2	Wasm-based VNF Provider delivers the VNF descriptor, artefacts and other CSAR content in a VNF package.	The Wasm module as well as the necessary information about the runtime and the extensions are contained or referenced in the VNF package.

5.8.5 Post-conditions

Table 5.8.5-1 describes the use case post-conditions.

Table 5.8.5-1: Use case #7 post-conditions

#	Post-condition	Description
1	The VNF Package is onboarded and is available for Wasm-based VNF instantiation.	

5.8.6 Flow description

Table 5.8.6-1 describes the use case flow.

Table 5.8.6-1: Use case #7 flow description

#	Actor/Role	Action/Description
1	Operator -> OSS	The operator request, via the OSS to onboard the VNF package of a Wasm-based VNF.
2	OSS -> NFV-MANO system	OSS/BSS sends to the NFVO a create VNF package information request.
3	NFV-MANO system	Creates the individual VNF Package resource.
4	OSS -> NFV-MANO system	The OSS provides the VNF package to NFVO and requests the onboarding. See note 1.
5	NFV-MANO system	Proceeds with the onboarding, i.e. process the VNF package content, checking the VNFD. NFVO stores the VNF package contents in its repository (e.g. storing Wasm modules in the Wasm software repository, etc.). See note 2.
6	NFV-MANO system -> OSS	The NFV-MANO system notifies the OSS that the VNF package has been successfully onboarded.
NOTE 1: The VNF package can reference external artifacts, in that case access info are provided by OSS, or the VNF package content is included in the CSAR or the VNF package content is uploaded indirectly.		
NOTE 2: See ETSI GS NFV-SOL 016 [i.17], clause 5.1 for a detailed description of the VNF package onboarding process.		

5.9 Use case #8: instantiating a Wasm-based VNF

5.9.1 Introduction

Use case #7 describes the VNF package onboarding of a Wasm-based VNF. The NFV-MANO can instantiate the VNF according to the information contained in the VNF package. For example, installing the adequate runtime, and providing the necessary extensions or plugins needed (e.g. to access specific hardware resources or networking capabilities, etc.). Besides the setting the runtime and the extensions, Wasm has another specificity, that is, the runtime itself operates as a scheduler managing how modules execute within its process. This differs from VMs or containers, where resource limits are specified at the OS or at the container orchestration level. Configuring Wasm resources is performed via the runtime.

The present use case describes the instantiation of a Wasm-based VNF, while considering the related requirements and resource configuration.

5.9.2 Actors and roles

Table 5.9.2-1 describes the use case actors and roles.

Table 5.9.2-1: Use case #8 actors and roles

#	Actor and role	Description
1	Operator	A human being or an organization seeking to deliver network services to end-users.
2	OSS	The entity that receives the operator's request for the initiation of Wasm-based VNFs.
3	NFV-MANO system	Network management and orchestration framework comprising NFVO(s), VNFM(s), VIM(s), etc. It is in charge of VNFs and NSs lifecycle management and OAM support.

5.9.3 Trigger

Table 5.9.3-1 describes the use case trigger.

Table 5.9.3-1: Use case #8 trigger

Trigger	Description
1	The use case is triggered when an Operator requests to instantiate a Wasm-based VNF.

5.9.4 Pre-conditions

Table 5.9.4-1 describes the use case pre-conditions.

Table 5.9.4-1: Use case #8 pre-conditions

#	Pre-condition	Description
1	The NFV-MANO system is operational.	
2	The VNF package containing the VNFD and the reference to the Wasm module and information about the module's requirements is onboarded and available.	

5.9.5 Post-conditions

Table 5.9.5-1 describes the use case post-conditions.

Table 5.9.5-1: Use case #8 post-conditions

#	Post-condition	Description
1	The VNF is instantiated successfully, and it is ready for subsequent lifecycle management by the NFV-MANO system.	

5.9.6 Flow description

Table 5.9.6-1 describes the use case flow.

Table 5.9.6-1: Use case #8 flow description

#	Actor/Role	Action/Description
1	Operator -> OSS	The operator request, via the OSS to deploy a VNF made up of Wasm module(s).
2	OSS -> NFV-MANO system	The OSS requests the deployment of the VNF.
3	NFV-MANO system	Proceeds with the instantiation of the VNF by creating the VNF instance resources, performing the necessary resource orchestration and granting (if required).Installing the required runtime and adding the necessary plugins and/or extensions. See note.
4	NFV-MANO system -> OSS	The NFV-MANO system notifies the OSS that the VNF has been instantiated successfully.
NOTE: The entities responsible for managing the Wasm-based VNF LCM are yet to be defined.		

6 Key issue analysis

6.1 Introduction

In clause 4, the serverless computing model and the new virtualisation technologies are introduced, and their pros and cons are also analysed. Clause 5 involves the use cases illustrating the relevant scenarios to the NFV framework. In the following clauses, key issues regarding how to efficiently integrate the serverless computing model and new virtualisation forms in the NFV framework.

6.2 Key issues on serverless computing model

The serverless computing model introduced in clause 4.2.1 is a cloud computing model that is considered to better assist application developers in reducing their focus on infrastructure management. The Function as a Service (FaaS) introduced in clause 4.2.2 is an implementation of the serverless computing model, where developers only need to provide code at the function level to deploy applications. It is necessary to investigate how to enhance the NFV framework to accommodate serverless computing.

Key issue #1.1: Key components of serverless framework

There is a need to identify the key components for implementing the serverless computing model in the NFV framework and the roles of these components, and to research which technologies can support the deployment of serverless applications.

Questions relevant to this key issue include:

- What are the key components needed to realize serverless cloud computing model in NFV framework?
- What are the virtualisation and runtime technologies needed to deploy and run serverless applications?

Key issue #1.2: Modelling serverless applications and functions using NFV object model

Deploying applications that follows the serverless architecture using the NFV-MANO system means that it is necessary to describe serverless applications. There is a need to analyse how to map serverless applications and functions with the NFV model (VNFCs / VNFs / NSs).

Questions that can be associated with this key issue include the following:

- How to model serverless applications and functions using the NFV object model?
- Is it necessary to introduce new objects to model serverless applications and functions?

6.3 Key issues on serverless applications and functions packages and descriptors

As described in clause 4.2.2 of the present document, the serverless applications packages as well as the packages of the functions comprising it need to be provisioned to the NFV system. The following key issues are identified regarding packages and code management.

Key issue #2.1: Structure and contents of serverless application package and function package.

Use case #1 in clause 5.2 and use case #2 in clause 5.3 state that the serverless application provider provides a serverless application/function package to deploy the serverless functions of the application. The serverless application/function package may include necessary information for the application/function deployment, e.g. code files, code runtime image, dependency files, resource requirements, etc.

Questions relevant to this key issue include:

- What is the structure of the serverless application package and function package?

- How to describe the requirements and attributes of a serverless application and/or function?
- How can a serverless application/function be packaged and described using the generic NFV model (VNF Package/ VNFD)?

Key issue #2.2: Management of serverless applications and functions relevant information in the NFV system

One of the important characteristics of the serverless computing model is that the application provider can deploy the serverless application by providing code rather than a VM image or an OS-container image. Use case #1, #2 and #3 describe the procedures of storing/fetching the code files to/from an internal repository. As time efficiency is essential for serverless services, downloading code files from the internal repository to the host and unzipping it could severely impact the serverless function start-up time. In addition, code runtime image and other dependency files should be managed by NFV-MANO as well. How to manage the code files, code runtime image and other dependency files efficiently in the context of the NFV framework is to be investigated.

Questions that can be associated with this key issue include the following:

- How and where to provision/store the function code files and code runtime image in the NFV system?

6.4 Key issues on management of serverless applications and functions

A serverless application can comprise multiple serverless functions. Unlike traditional deployment methods for VNFs or CNFs, which require resource allocation and instance creation upfront, serverless deployments defer these actions until the occurrence of a specific triggering event. Hence, the management of serverless applications may be different from the management of VNF/CNF. The following key issues are identified regarding serverless application management.

Key issue #3.1: Triggering serverless function instantiation

An application designed following the serverless computing model can require specific capabilities related to LCM and OAM operations. For instance, rapid function instantiation based on traffic is required by serverless applications, different types of triggering events need to be monitored by the NFV-MANO to ensure proper functioning of the applications, e.g. when an event occurs, there is a need to quickly find the associated function and trigger the instantiation of the function, etc. Therefore, it is important to identify the key components to enhance the NFV framework to accommodate serverless computing and the roles of the corresponding management components.

Questions relevant to this key issue include:

- How to trigger a serverless function instantiation?
- Which management entity is monitoring the triggering events for serverless function?

Key issue #3.2: LCM of serverless function

When a triggering event occurs, the management system will need to create a function instance to perform some processing on incoming traffic (this is in case there are no instances to handle the incoming traffic). After processing the traffic, the function instance is terminated at a specified time, and the resources are released. This lifecycle management of function instances can improve the utilization rate of resources. When instantiating an application designed following the serverless computing model, the infrastructure resource allocation method needs to consider how to support the instantiation of functions and how to prevent resources from being depleted by rapidly growing request demands.

Questions that can be associated with this key issue include the following:

- Which entity can manage the serverless application?
- Which entity is responsible for creating/deleting/updating serverless function instances?
- Which entity is responsible for configuring serverless function instances?
- How to implement infrastructure resource management of serverless function?

- How is the LCM of the serverless function related to the LCM of the serverless application?

Key issue #3.3: Performance/Fault management of serverless applications and functions

After creating a function instance, NFV-MANO needs to periodically obtain information about the function's operation to determine its running status, in order to decide whether to assign events to this function instance for processing, or whether to reclaim the resources of this function instance. Similarly, NFV-MANO needs to be able to perform Fault management for serverless applications/functions.

Questions that can be associated with this key issue include the following:

- How to monitor the status of the serverless application/functions instances?
- How to get the performance metrics/fault information from the serverless application/functions instances?

6.5 Key issues on deploying serverless application efficiently

In addition to the implementation of basic functions, there is a need to also study how to enhance the serverless deployment process in the NFV system to achieve better performance or stronger service capabilities. Relevant key issues include:

Key issue #4.1: Reducing the cold start-up time

The cold start issue refers to the situation in serverless computing where, when a function is invoked for the first time, it requires resource allocation, installing of the runtime environment, and initiation of the function code. This process results in a longer response time. This issue can delay the response of time-sensitive applications, increase resource consumption which can result in poor user experience.

Questions relevant to this key issue include:

- How to reduce the cold-start time of serverless functions?

Key issue #4.2: Combine the usage of serverless computing model with other technologies

Clause 4.3 describes several new virtualisation technologies. These virtualisation technologies can be used to support serverless deployments and achieve better resource virtualisation or performance.

Questions that can be associated with this key issue include the following:

- How virtualisation technologies can be used in the framework to support serverless applications scenarios?

6.6 Key issues related to new virtualisation technologies

6.6.1 Key issues related to eBPF®

Key issue #5.1: Modelling eBPF-based VNFs

Deploying eBPF programs through the NFV-MANO system necessitates an analysis of how to align eBPF programs with the NFV model (e.g. using VNFCs or VNFs). Relevant questions to this key issue include the following:

- How can eBPF programs be modelled using the NFV object model? Is there a need to create new objects to represent eBPF programs?

Key issue #5.2: Information about eBPF programs resource usage

eBPF programs run inside the kernel at specific hooks points (e.g. syscall, XDP for packets). An eBPF program only uses 11 registers and a stack fixed at 512 bytes per program invocation [i.22]. Hence, an eBPF program cannot use arbitrary CPU, RAM, or storage. The CPU usage depends only on how often the hook is triggered (e.g. by every packet or by occasional process launch), the complexity of the program itself, and if JIT was used for compilation or in-kernel interpreter. For larger and persistent storage, eBPF relies on maps, which are kernel-allocated key-value stores that use kernel RAM. It is possible for the eBPF program provider to choose the size of the maps. Given these specificities about eBPF programs the following questions can be raised:

- Can the NFV-MANO estimate the memory footprint from the maps' sizes used by eBPF programs (e.g. how many entries, key/value size that can be provided in the descriptor)?
- Can the NFV-MANO with the help of information about the eBPF program (this information can be made available in the descriptor), e.g. the program type (XDP, kprobe, tracepoint, etc.), the trigger frequency, etc., estimate the expected running time to size and monitor the deployment?

Key issue #5.3: eBPF-based VNF package and descriptor

eBPF source code needs to be compiled into the machine instructions, i.e. eBPF bytecode that can be loaded to the kernel. The eBPF-based VNF package may include this bytecode along with the necessary information to make the NFV-MANO aware about, for example, where the program will be hooked, when it can be detached and unloaded, which other eBPF programs the program is calling, what are the maps that the eBPF program will use, and which kernel version is suitable for the eBPF program (some kernel versions can provide certain helper functions while other versions cannot).

Questions relevant to this key issue include:

- Can the same structure of the existing VNF packages and descriptors be used for eBPF-based VNFs?
- How can the information regarding the eBPF programs be incorporated in the existing VNFDs?

6.6.2 Key issues related to Wasm

Key issue #6.1: Modelling Wasm-based VNFs

Deploying Wasm modules or components through the NFV-MANO system necessitates an analysis of how to represent these modules/components based on the NFV object model (e.g. VNFCs, VNFs, MCIOs, MCCOs). Relevant questions to this key issue include the following:

- How can Wasm module/component be modelled using the NFV object model? Is there a need to create new objects to represent Wasm module/component?

Key issue #6.2: VNF Packages of Wasm-based VNFs

In general, VNF delivered as Wasm module(s) or component(s) can be deployed on resources incorporating a Wasm runtime. For containerized Wasm deployments the VNF packaging procedure remains the same considering MCIOP, where the container image can include Wasm module(s) or component(s). The image can then run in a container. With regard to this context, questions related to this key issue can be:

- Can the same structure of the existing VNF packages and descriptors be used for Wasm-based VNFs?
- How can the information regarding the Wasm module/component (e.g. Wasm runtime used, WASI rights, etc.) be incorporated in the existing VNFDs and the VNF Package?

Key issue #6.3: Resource allocation management

Wasm modules or components have their own sandboxed block of memory that the runtime controls, through the runtime the NFV-MANO system can manage, for example the initial number of pages and the maximum growth limit to prevent the module/component from consuming more memory. The NFV-MANO system will also need to configure which WASI rights each VNF instance has (like network, file read/write, etc.).

- Given these examples, how can such operations be modelled in the NFV-MANO to carry the allocation and management of resources of Wasm modules and components?

Key issue #6.4: LCM and OAM management

In case Wasm modules are deployed as VNFs, their LCM operations will be performed by VNFM and the OAM operations by VNFM and VNF generic OAM functions, defined in ETSI NFV-IFA 049 [i.20]. In that case additional investigation is needed on whether existing operations and interfaces supported by VNFM/CISM are applicable for the case of exposing Wasm modules as VNFs and whether updates are needed in the VNF generic OAM functions framework to support related OAM operations.

- Given that Wasm modules are deployed as VNFs, investigate whether VNFM and/or CISM and/or CCM operations are sufficient to cover LCM and OAM related operations.
- Investigate whether operations performed by VNF generic OAM functions are sufficient to cover OAM related operations.

6.6.3 Key issues related to microVMs

Key issue #7.1: Modelling MicroVMs

MicroVMs are instantiated and managed via lightweight Virtual Machine Monitors (VMM) such as Firecracker®, Cloud hypervisor® operating on top of the KVM hypervisor in the Linux kernel. Given this operational model and its differences from conventional virtualisation, the following question can be raised:

- How can microVMs be modelled within the NFV framework?

Key issue #7.2: MicroVMs VNF Packages

In contrast to conventional VNF packages, which in case of VMs include or reference a bootable disk image containing the operating system and filesystem, microVMs do not rely on a bootloader and start directly from a kernel image. The kernel therefore requires a separate root filesystem to initialize the system. Additionally, microVM deployments can require a configuration file that specifies boot parameters and virtio device definitions needed for proper startup. Questions that can be raised in this context are:

- How to adapt VNF packages to include these microVMs specific artifacts and parameters?

Key issue #7.3: Management of microVMs

To instantiate a microVM, the VMM loads the guest's kernel binary directly into the memory space allocated for it. This contrasts with VM deployment, where the VMM (e.g. QEMU) emulates a BIOS or UEFI firmware that loads a bootloader (e.g. GRUB); which starts the guest kernel from a disk image. MicroVM deployments skip the firmware and bootloader entirely, and the VMM is responsible for the instantiation and LCM operations of the microVMs. In this context, the following questions can be raised:

- Can microVMs be defined as virtualised infrastructure resources managed by VIM or is another management function needed?
- How can the VMM be modelled within the NFV architecture?
- How VNF LCM operations related for example to granting can be performed in case of using microVMs resources?
- Which management entity will be responsible for microVM OAM related operations?

6.6.4 Key issues related to unikernels

Key issue #8.1: Modelling Unikernels

Unlike conventional VMs, unikernels are single-images and immutable deployments that are optimized for a single task where the entire software stack of system libraries, language runtime, and applications is compiled into a single bootable VM image that runs directly on a hypervisor. This architecture results in fast boot times, minimal memory footprints, and a reduced attack surface compared with both VMs and containers. Given this architecture, the following questions can be raised:

- How can unikernels be represented within the NFV framework? Is it possible to reuse the VM model to represent them?

Key issue #8.2: Unikernel description in the VNFD

VNFD is a declarative model that defines the structure of the VNF (VDUs, connection points, etc.) deployment requirements (flavours, images) and other aspects related to LCM operations. Unikernels, compile the application code together with the necessary OS components (like networking, file system, or drivers) into one executable image. The VNFD could reference and parameterize unikernel images (e.g. specify image format, boot parameters, resource flavours), but it does not have a specialized construct for "unikernels" as a distinct workload type:

- How can the VNFD reference and parameterize a unikernel-based workload, including its image format and minimal virtual interface requirements?

Key issue #8.3: Management of unikernels

The NFV MANO needs to handle lifecycle operations of unikernels which lack bootloaders, guest configuration mechanisms, etc. Hence, conventional initial setup as well as post-deployment management represented in NFV framework may not be fully applicable:

- Which NFV-MANO components can be reused or be extended to support the instantiation and the LCM and OAM management of unikernels?
- Is there a need to enhance/extend existing reference points to perform LCM and OAM operations related to unikernels?

7 Analysis and potential solutions

7.1 Introduction

Clause 7 documents potential solutions addressing the key issues discussed in clause 6 of the present document. Each solution includes solution description, related issues and gap analysis.

Table 7.1-1 includes a list of the solutions and the key issues they cover.

Table 7.1-1: Summary of solutions and related key issues

Solution	Title	Related key issues
Solution #1	FaaS runtime software management	Key issue #1.1: Key components of serverless framework Key issue #2.2: Management of serverless applications and functions relevant information in the NFV system
Solution #2	PaaS service supporting triggering function instantiation	Key issue #3.1: Triggering serverless function instantiation Key issue #3.2: LCM of serverless function
Solution #3	Reducing cold start-up latency by runtime resource pool	Key issue #4.1: Reducing the cold start-up time

Solution	Title	Related key issues
Solution #4	Modelling serverless application	Key issue #1.2: Modelling serverless applications and functions using NFV object model Key issue #2.1: Structure and contents of serverless application package and function package
Solution #5	High-reliability Serverless application update	Key issue #3.3: Performance/Fault management of serverless applications and functions Key issue #4.2: Combine serverless with other technologies
Solution #6	Rapid distribution of serverless function image	Key issue #1.1: Key components of serverless framework Key issue #4.1: Reducing the cold start-up time

7.2 Potential solutions

7.2.1 Solution #1: FaaS runtime software management

7.2.1.1 FaaS runtime software repository functionality

The present solution introduces a FaaS Runtime Software Repository (FRSR) that contains the software and/or artefacts needed for the execution of the serverless code (i.e. FRS) supplied by the serverless application provider. When the serverless application provider aims to deploy an application, they can specify the name and version of the target FRS in a metadata file of the application package. The FRSR is managed by the NFV-MANO to store FRS. This repository might support consumers (e.g. some NFV-MANO functions or functional blocks) in quickly indexing the target FRS by name and version.

NOTE 1: The mainstream FaaS runtime supported by public clouds or serverless open-source communities include support for Python, Java, Node.js, Go, Ruby, among others. Additionally, public cloud services enable the providers of serverless applications to also provide custom runtimes. The specific FaaS runtimes supported are determined by the operator's policies.

For custom FRSs, the NFV system might adopt the following two approaches to store FRSs:

- Option 1: The NFV-MANO system exposes a FaaS runtime software management interface, enabling serverless application providers to upload the serverless applications' packages with the necessary software to the FRSR before deploying the serverless applications.
- Option 2: Providers of the serverless application provide the name, version information, and a URL to download the FRS in the metadata file of the application package. During the on-boarding phase, the NFV-MANO system downloads the FRS from the specified URL and stores it to the FRSR. When providers of the serverless application remove the application package from the NFV-MANO system, the NFV-MANO system remove the FRS from the FRSR accordingly.

The main difference between Option 1 and Option 2 lies in the timing of storing the FRS. Option 1 has the advantage of pre-downloading the FRS, thereby reducing the latency during the on-board phase. Furthermore, security considerations need to be addressed, in particular regarding whether the NFV-MANO system can actually access the URL, e.g. if such storage system is on a different network that the network operator of the NFV-MANO system is not allowed to access. Within the NFV-MANO system, FRS can be stored in one or more FRSRs (see note 2).

NOTE 2: To reduce the latency of installing FaaS Runtime Software onto hosts, the FRSR can utilize distributed storage system.

To support the FaaS deployment, FRSR can be introduced into the NFV-MANO to manage FRS by exposing a runtime software FRS management interface, to perform CRUD operations directly in the FRSR. Some operations are recommended to be realized, for example, creation/update/deletion could be supported to maintain FRS. Potential consumers (e.g. some NFV-MANO function or functional blocks) may need to obtain either brief information about the FRS (such as name, version) or the FRS itself.

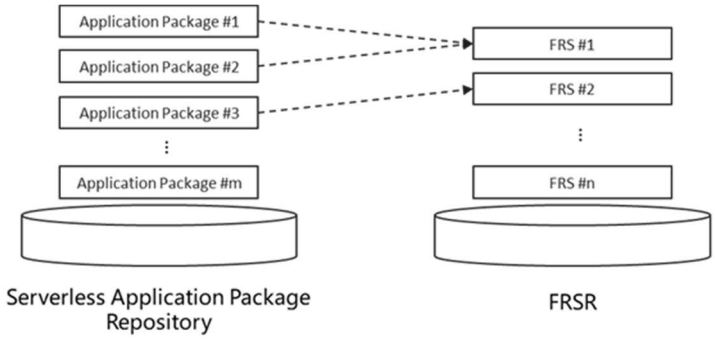

Figure 7.2.1.1-1: Telco cloud application package repository and FRSR

In figure 7.2.1.1-1, the relationship between the serverless application package repository and FRSR is depicted. As long as the FSR is available in the FRSR, it can be reused if needed by other serverless applications.

7.2.1.2 FaaS runtime in container environment

FaaS functions can be deployed using containers technology hosting their runtime environment. Containers can support FaaS in the following ways:

- Fast Startup: The lightweight nature of containers enables function instances to start quickly, which is crucial for meeting FaaS's low-latency requirements.
- Resource Isolation: Containers provide an isolated runtime environment for each function instance through namespaces and resource limits, ensuring that different functions do not interfere with each other.
- Scalability: Container orchestration tools can automatically scale function instances to meet the elasticity demands of FaaS.

In scenario #1 the FaaS function is directly deployed on the host or VM. In this model, function providers only need to provide the function code and the information about the FaaS runtime which is necessary. The operator is responsible for providing the infrastructure resources and hosts pre-installed with the FaaS runtime software. The NFV system then loads the function code onto the hosts or VM, upon request (see solutions in clause 7.2.1.1).

Scenario #2 is about container-based FaaS functions deployments. In this scenario, the serverless application provider needs to create an OS container image, which is provided to the NFV-MANO system for deployment. The OS container image contains the FaaS function code, the FaaS runtime software, and other necessary components. In this scenario, NFV-MANO stores the OS container image in the CIR and, during the function instantiation phase, loads the OS container image to the target host environment for execution. The function instantiation process is consistent with that of the standard container model.

7.2.1.3 Related issues

The present solution addresses the following key issues described in clause 6:

- Key issue #1.1; and
- Key issue #2.2.

7.2.1.4 Gap Analysis

7.2.1.4.1 FaaS function direct deployment on the host

For the case where FaaS function is directly deployed on the host (or VM) machine, the NFV-MANO system lacks the capability to manage the FRS. The FRSR needs to possess the ability to manage FaaS runtime software, enabling consumers to perform operations such as adding, deleting, updating, getting, and querying FaaS runtime software by invoking APIs.

Serverless application providers need to specify the name and version of the FRS that their FaaS function depends on in the serverless VNF's descriptor.

Gap #1.1: FRSR needs to expose interfaces to support CRUD operations of FRS.

7.2.1.4.2 FaaS function deployment based on Container

For the case that FaaS function deployment based on Container, if the FRS provided by the serverless application provider is packaged in an OS Container Image and delivered to the NFV system, it can be considered a special method of container deployment. Within the NFV system, the OS Container Image is managed by the CIR. Since serverless application providers do not need to select the FRS, the NFV system does not need to provide external query capabilities for FRS. No gaps are identified in this model.

7.2.2 Solution #2: PaaS service supporting triggering function instantiation

7.2.2.1 Introduction

Key issue #3.1 mentions that NFV-MANO needs to manage function instances based on actual traffic, monitor corresponding events according to the requirements of different functions, and trigger function instantiation when events occur. The present solution considers introducing a new PaaS service that can support serverless application instantiation.

7.2.2.2 Solution description

To deploy serverless application, a viable solution is to utilize the PaaS service mechanism, which helps serverless application providers in deploying serverless applications.

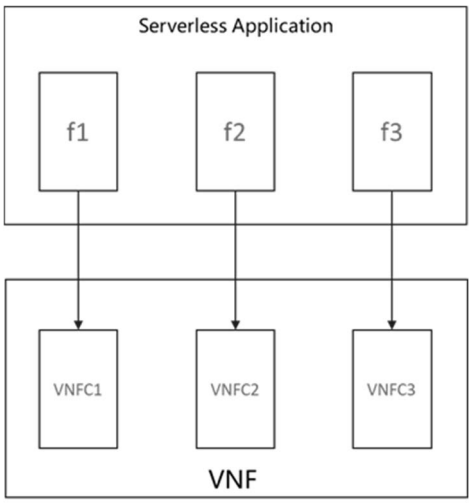

Figure 7.2.2.2-1: Mapping relationship between Functions and VNFCs within NFV-MANO

Figure 7.2.2.2-1 depicts the mapping relationship between serverless applications and VNFs. A serverless application may contain multiple functions and can be mapped to a serverless VNF, whilst each function can be mapped to a VNFC. When function provider prepares to deploy a serverless VNF (i.e. a serverless application), a PaaS service can be employed to monitor the corresponding events. The type of this PaaS service could be regarded as a "supporting serverless" PaaS service. The NFV-MANO can instantiate a PaaS service in accordance with ETSI GS NFV-IFA 049 [i.20], which outlines how NFV-MANO supports PaaS service deployment. For example, one possible approach is to deploy a PaaS service as a VNF. This PaaS service may expose an interface to enable the consumers (e.g. OSS, VNFO, VNFM, PSM etc.) to provide or update the PaaS service configuration parameters. For example, configuration parameters can be related to condition for triggering or terminating function instances.

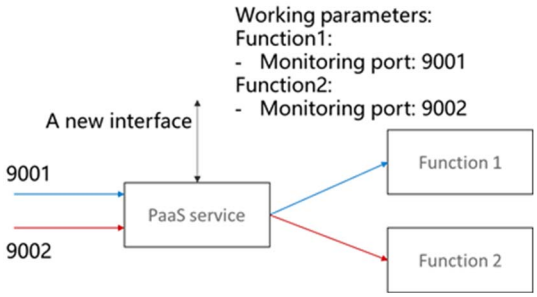

Figure 7.2.2.2-2: One example about working parameters

The working parameters can involve function instantiation conditions. Figure 7.2.2.2-2 gives an example about function instantiation parameters in the HTTP message scenario. The function provider can configure the accessing port information through a new interface, which is an interface exposed by the PaaS service. The messages accessing port 9001 can lead to function 1 instantiation, meanwhile the messages accessing port 9002 can lead to function 2 instantiation.

The working parameters can also involve the function de-instantiation conditions. For instance, after the function instance was created, the MANO needs to determine when it can be terminated. The function provider can configure a period though a new interface, after which the function instance can be terminated.

This PaaS service can monitor target events and determine whether the triggering conditions have been met. For example, given that the Instantiate VNF operation can be executed via the Ve-Vnfm-vnf reference point as specified in ETSI GS NFV-IFA 008 [i.21], when a "supporting serverless" PaaS service detects a target event and identifies that no available function instance exists, it can notify the VNFM to instantiate the serverless function.

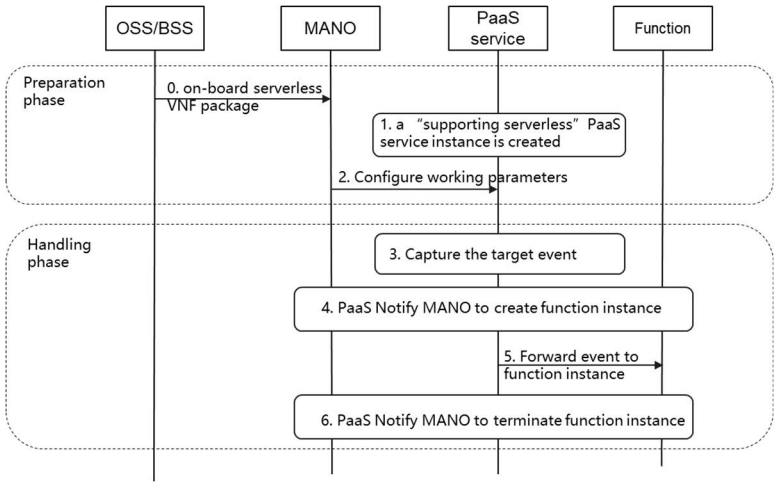

Figure 7.2.2.2-3: PaaS service triggers function instantiation using NFV-MANO

Figure 7.2.2.2-3 illustrates the deployment process of a serverless VNF:

- After a serverless VNF package is on-boarded, a 'serverless supporting' PaaS service can be instantiated and associated with the serverless VNF. Then, the working parameters can be configured to PaaS service instance via the interface exposed by PaaS service. The working parameters can indicate the function instantiation conditions and the function de-instantiation conditions.

- When the target event occurs, the PaaS service instance will determine if the associated serverless VNF needs to be instantiated. If the trigger conditions provided by the function provider can be satisfied, the PaaS service instance notifies VNFM to trigger the instantiation of the serverless function. Once the serverless function is instantiated, the PaaS service instance will forward the corresponding event/package to the instance for processing. The PaaS service instance can also notify VNFM to de-instantiate the function instance based on the de-instantiation trigger conditions provided by the serverless application provider.

7.2.2.3 Key issues address

The present solution aims at addressing aspects of the following key issues described in clause 6:

- Key issue #3.1; and
- Key issue #3.2.

7.2.2.4 Gap analysis

Gap #2.1: The usage of a PaaS service that exposes an interface to enable serverless application providers to configure parameters used for triggering serverless VNF instantiation and termination.

7.2.3 Solution #3: Reducing cold start-up latency by runtime resource pool

7.2.3.1 Introduction

Clause 4.2.2.2 highlights that cold start is a significant challenge for FaaS. In fact, this is a trade-off between resource utilization and response latency. To address this, this solution proposes introducing a runtime resource pool to mitigate the cold start issue.

7.2.3.2 Solution description

One of the key contributors to cold start latency is the preparation of infrastructure. The process of function instantiation involves the following steps:

Step 1 Allocating a compute node.

Step 2 Downloading the FaaS runtime software package to the compute node and unzipping it.

Step 3 Configuring the function runtime environment (e.g. setting environment variables).

Step 4 Downloading the FaaS code package to the compute node and unzipping it.

Step 5 Function initialization.

These steps can be categorized into two groups:

- Category 1: Steps specific to a particular FaaS function (Step 4 and 5).

These steps need to be executed after receiving the corresponding event and identifying the target function.

- Category 2: Steps applicable to the FaaS functions with the same runtime requirements (Step 1, 2, and 3).

These steps can be performed in advance of event capture to reduce cold start latency.

Some runtime resource pools can be pre-provisioned based on different runtime software requirements before function instantiation, as shown in figure 7.2.3.2-1. The runtime software can be installed on each compute node within the runtime resource pool. FaaS functions with the same runtime requirements can share a single runtime resource pool. When a corresponding event is captured, a pre-prepared compute node can be allocated from the pool, skipping steps 2 and 3 and proceeding directly to step 4. This approach can save significant time and reduce cold start latency.

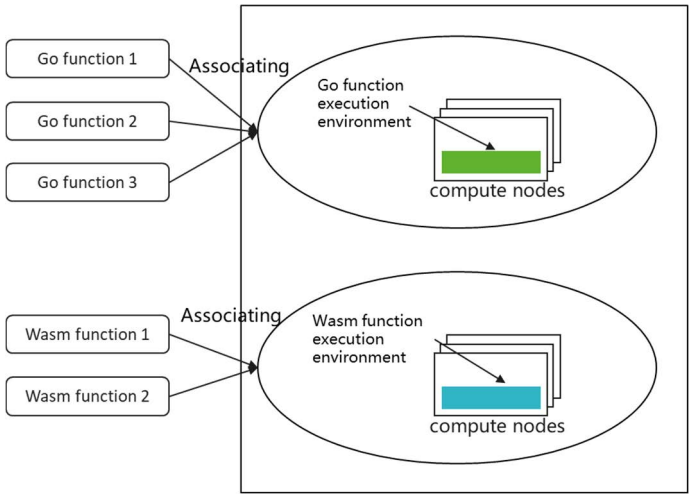

Figure 7.2.3.2-1: Runtime resource pools pre-created by the operator

The operators can establish runtime resource pools for commonly used runtime environments, such as Go, python, Wasm and Node.js, etc. A unique pool identifier can be generated by the operators and used to associate the runtime resource pool with serverless VNFs. It might be delivered to the trigger entity (e.g. a PaaS service described in clause 7.2.2) responsible for triggering function instantiation via a northbound interface. This trigger entity can then send a function instantiation request to the VNFM with the pool identifier. Consequently, the VNFM can instruct the infrastructure manager (e.g. VIM/PIM/CISM) to allocate a compute node from the specified runtime resource pool.

However, pre-provisioned runtime resource pools consume infrastructure resources regardless of whether corresponding events occur. This can lead to significant waste, especially when the event volume is low. In such cases, application providers are still required to pay for the reserved infrastructure resources. To address this issue, setting a maximum quick-start instance count can help achieve a better trade-off between resource utilization and low-latency. The maximum quick-start instance count determines the number of computer nodes that can be allocated from the runtime resource pool for a specific function. When an event is captured, if the number of running function instances allocated from the runtime resource pool is below the maximum quick-start instance threshold, a pre-prepared compute node can be allocated from the runtime resource pool with minimal delay. Otherwise, the entire process from Step 1 to Step 5 will be executed, leading to a longer delay.

The traffic load of some serverless applications might fluctuate significantly between weekdays and weekends. To optimize resource allocation, the maximum quick-start instance count provided by the serverless application provider can effectively limit the number of pre-provisioned instances to a reasonable level. For example, during weekdays, an application may receive around 100 requests per hour, according to historical data. On weekends, however, the application traffic surges. In such cases, the maximum quick-start instance count can be set to 100 to ensure a good user experience. If the access volume exceeds 100 requests, serverless computing can still handle all incoming requests efficiently through auto-scaling capabilities.

7.2.3.3 Key issues address

The present solution aims at addressing aspects of the following key issues described in clause 6:

- Key issue #4.1.

7.2.3.4 Gap analysis

The referenced ETSI NFV specifications in the present solution do not specify:

Gap #3.1: Creating a runtime resource pool based on serverless VNF requirements and associating serverless VNFs with it.

Gap #3.2: Maximum quick-start instance count is provided in the function descriptor.

Gap #3.3: VNFM monitors the number and the status of function instances and limits the number of compute notes allocated from the runtime resource pool based on the maximum quick-start instance count.

7.2.4 Solution #4: Modelling serverless application

7.2.4.1 Introduction

Conventional VNF packages typically comprise multiple components including VNFD, software images, and other artifacts. Key issue #1.2 mentions that it is necessary to abstractly describe serverless applications by mapping to the generic NFV model. Clause 7.2.4 focuses on how to model serverless application and analyse of serverless application packages and descriptor requirements.

7.2.4.2 Solution description

A software developer or team can sign a business agreement with an operator, and provide a serverless application package to deploy on the operator's infrastructure resources.

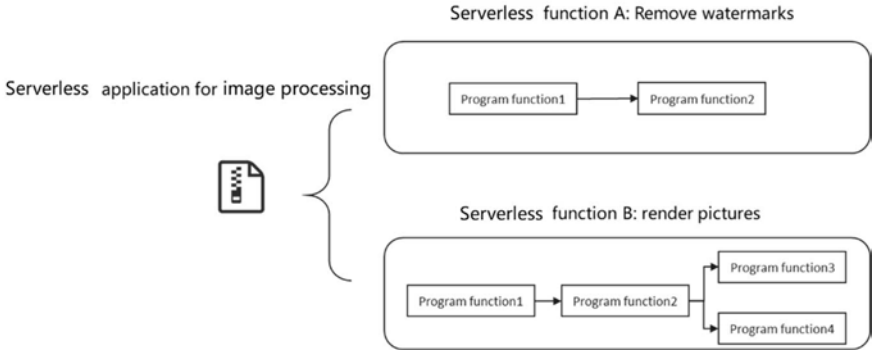

Figure 7.2.4.2-1: One example of serverless application for image processing

A single serverless application can be composed by one or more serverless functions and expose multiple independent services to consumers. A serverless function is not a program function (i.e. a code block enclosed in braces), but a deployable unit that may contain one or more program functions sharing the same deployment and execution requirements. Figure 7.2.4.2-1 illustrates an image processing application providing two kinds of services: Watermark removal service (serverless Function A) and Picture rendering service (serverless Function B). They can be called in combination or independently, and they may have different upgrading strategies. For instance, if a new rendering algorithm is introduced, only serverless Function B needs updating - without modifying the entire application. Hence, FaaS functions can be upgraded independently.

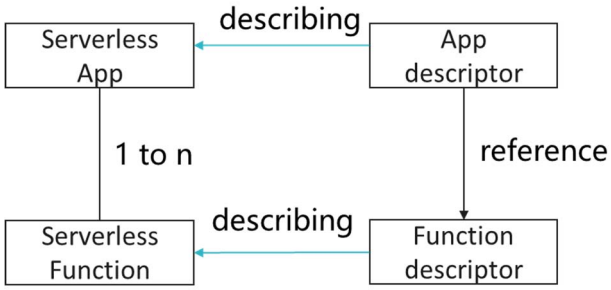

Figure 7.2.4.2-2: Relationship of serverless application and serverless function

Figure 7.2.4.2-2 depicts the relationship of serverless application and serverless function. A serverless application descriptor can be introduced to describe the common requirements, such as application provider name, runtime software information, referenced Serverless function descriptor ID, etc. The serverless function descriptors can be introduced to cover the specific requirement, such as memory, trigger conditions, threshold of instance, function code, etc.

NOTE: Mapping between serverless objects and VNF objects can be done when the new architecture for Release 6 is stable.

A serverless application package includes a serverless application descriptor, one or more serverless function descriptors, function code images, as well as manifest file, checksum, etc. Runtime software might be included in the package if the function needs a customized execution environment. Function code images can be included in the serverless application package, or a downloading URL can be present in the function descriptor. Function code can be identified and managed by the function name and version. The serverless application package can be assembled in one file and signed by the application provider.

The serverless function package can be delivered if updating a function independently. It contains the serverless function descriptor, function code, and other things specific to a function.

7.2.4.3 Key issues address

The present solution aims at addressing aspects of the following key issues described in clause 6:

- Key issue #1.2 and #2.1.

7.2.4.4 Gap analysis

The referenced ETSI NFV specifications in the present solution do not specify:

Gap #4.1: Serverless application descriptor can be specified for describing serverless application requirements.

Gap #4.2: Serverless function descriptors referenced by serverless application descriptors can be specified for describing serverless function requirements.

Gap #4.3: The structure of serverless application package need to be specified.

Gap #4.4: The structure of serverless function package need to be specified.

Gap #4.5: The work flow of upgrading a serverless function and relevant interfaces need to be specified.

7.2.5 Solution #5: High-reliability Serverless application update

7.2.5.1 Introduction

Serverless is well-adapted for lightweight application scenarios. Serverless implementations typically exhibit significantly reduced code size. Serverless applications might be developed in an agile way, which means that more frequent update operations are expected. Clause 7.2.5 will give a solution for high-reliability Serverless function updates.

7.2.5.2 Solution description

Clause 5.2.1 explicitly defines that a serverless application package incorporates the function code, which can be hosted in an internal code repository. Each function code is uniquely identifiable through a composite key comprising its code version and function name, enabling systematic code version control within the NFV-MANO. When application providers initiate Serverless function updates, they execute version-controlled modifications to the codebase repository rather than altering currently running function instances. Inactive function instances that remain event-free for a configurable duration are automatically terminated. Consequently, subsequent events are processed by function instances initialized from the latest code version.

In production environments, robust failure handling mechanisms are essential. For conventional VNF implementations, automated rollback procedures are typically triggered when post-upgrade anomalies are detected. Another risk mitigation strategy for software updates is Canary update methodology. Canary update, constitutes a phased feature release approach. This technique systematically exposes the updated version to a controlled user subset while conducting performance monitoring. Through this graduated deployment paradigm, version stability is validated before progressive scaling and eventual full-scale implementation.

For Serverless implementations, Canary update can be effectively operationalized by maintaining parallel function code versions within the NFV system. The Serverless application provider onboards function code with an associated version-mapping policy that governs event distribution across different code versions. This version-mapping policy can be configured to implement proportional event distribution. Figure 7.2.5.2-1 demonstrates the Canary update workflow for Serverless deployments. During initial deployment, a minimal event subset (20 %) is directed to Version 2 (the updated version). Following successful performance validation, the provider can dynamically adjust the version-mapping policy to progressively escalate Version 2's event allocation, ultimately achieving complete event migration.

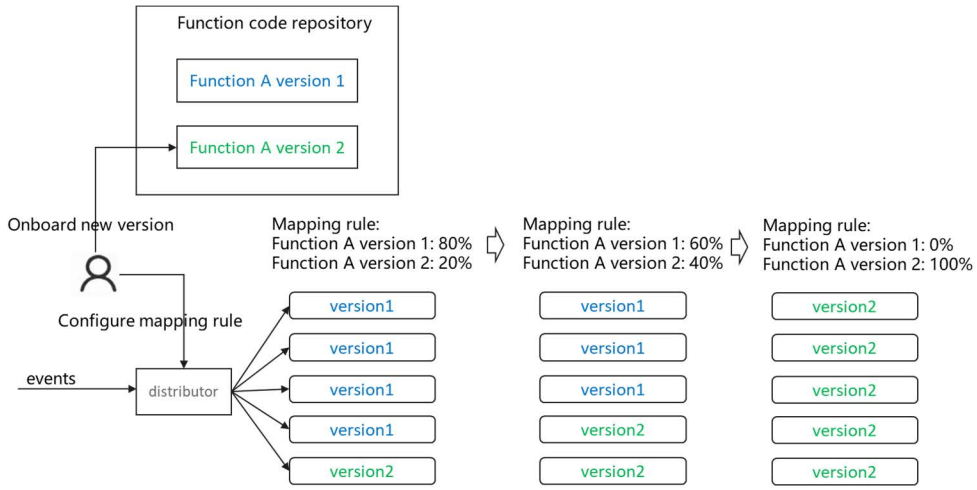

Figure 7.2.5.2-1: Canary Updating for Serverless

NOTE: The distributor depicted in figure 7.2.5.2-1 serves as an entity that undertakes the responsibility of load balancing. For example, the distributor is part of the PaaS service described in clause 7.2.2.

The Serverless application provider might trigger the rollback procedure if unexpected errors are present, e.g. more HTTP failure responses, longer processing time, etc. So, the provider needs to gain relevant metrics reflecting the running status of function instances to evaluate if rollback operation needs to be performed. Serverless metrics might include as follow:

- Invocations: Number of calls to the function code, including successful calls and calls that cause function errors.
- Errors: Number of calls that caused a function error.
- Duration: Amount of time the function code spends processing events (not include cold start time).

The Publish-Subscribe pattern enables Serverless application providers to subscribe to critical performance metrics. To facilitate metric reporting, OSS/BSS systems can include a callback URL in Serverless application update requests. The Serverless Function Management assumes responsibility for metric collection and forwards these metrics through the NFVO to the specified callback endpoint. This mechanism empowers application providers to monitor key performance indicators in real-time and make informed decisions regarding potential rollback operations.

7.2.5.3 Key issues address

The present solution aims at addressing aspects of the following key issues described in clause 6:

- Key issue #3.3 and #4.2.

7.2.5.4 Gap analysis

The referenced ETSI NFV specifications in the present solution do not specify:

Gap #5.1: Version-mapping policies and relevant procedures needs to be specified.

Gap #5.2: Serverless Function Management needs to collect the Serverless metrics, such as invocations, errors, duration, etc., and send the notification to a subscribing callback URL.

7.2.6 Solution #6: Rapid distribution of serverless function image

7.2.6.1 Introduction

In serverless computing model, the distribution requirements for function image, i.e. function code and other files related to the execution of function instances are more stringent in terms of latency, as serverless functions generally need to start within a few hundred milliseconds. The serverless function may face massive and unpredictable bursts of concurrency, which can require the distribution of images to hundreds of nodes within seconds. This requirement is difficult to meet through centralized image distribution methods. The present clause discusses how to achieve rapid distribution of Serverless function image.

7.2.6.2 Solution description

Decentralization is key to achieving rapid distribution of serverless function image in serverless computing model. Decentralization means there is no single central node responsible for pushing data to all target hosts; instead, hosts relay the data to each other, forming a dynamic distribution structure. The outbound bandwidth of a single central node distribution method is limited, and once a large number of sudden requests occur, the single central node may become a bottleneck. A separated distribution structure can be established for each function, in which serverless function images can be distributed rapidly and efficiently. One example of the distribution structure is the distribution tree, with the root node distributing to the first-level nodes, and the first-level nodes distributing to the second-level nodes, and so on.

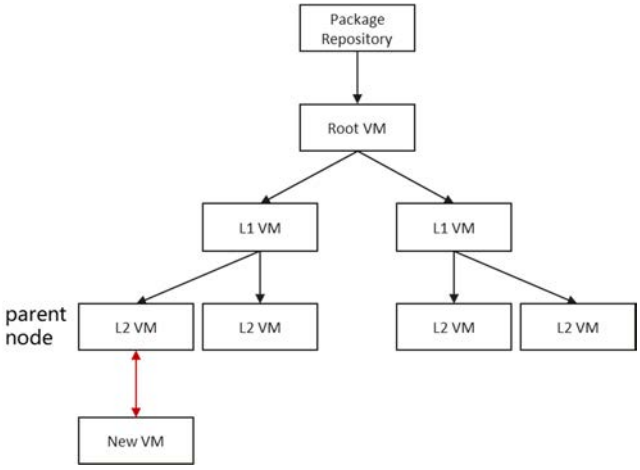

Figure 7.2.6.2-1: Serverless Distribution Tree in the VM example

As shown in figure 7.2.6.2-1, the advantage of distribution tree is that the distribution tree can maintain node balance through algorithms, meaning the tree height is kept as small as possible, thereby reducing the path length for image distribution. When a new node is to be added to the distribution tree, NFV-MANO can designate a parent node for the new node based on the node's topological location information, node load information, and content caching status, and establish a connection between the new node and its parent node. The new node requests to pull function code or other files from the parent node. If the parent node lacks certain content, it continues to request from its own parent node. When a node no longer participates in resource allocation of serverless functions, it is removed from the distribution tree, and the tree is readjusted. By adopting this method, it is possible to logically establish a separated distribution tree for each serverless function, as illustrated in figure 7.2.6.2-2. If a host can be shared by multiple functions, it can logically belong to multiple distribution trees simultaneously.

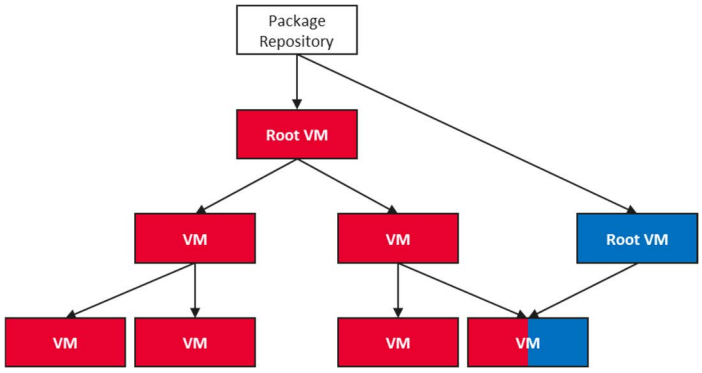

Figure 7.2.6.2-2: One Node Belonging to Multiple Distribution Trees in the VM example

When a serverless function is triggered for instantiation, the NFV-MANO can allocate a node to deploy the function instance. If the distribution structure corresponding to this function has not yet been established, the NFV-MANO can set it up. Otherwise, it can add the node to the existing distribution structure.

7.2.6.3 Key issues address

The present solution aims at addressing aspects of the following key issues described in clause 6:

- Key issue #1.1; and
- Key issue #4.1.

7.2.6.4 Gap analysis

The referenced ETSI NFV specifications in the present solution do not specify:

Gap #6.1: Establish and maintain decentralized distribution structures for distribution of serverless function images.

7.3 Evaluation of solutions

The present clause provides an assessment of the solutions outlined in clause 7.2. This solutions' analysis provides a foundation for developing recommendations and for conducting the normative work phase.

Table 7.3-1: Pros and cons analysis of solutions for key issues related to serverless

Solution	Pros	Cons
Solution #1 FaaS runtime software management	The solution introduces FRSR and exposes interfaces for operating FRS to consumers within or outside the NFV domain, enabling NFV-MANO to manage FRS capabilities. This solution can prevent the process of on-board from taking too long due to the inclusion of FRS in the application package. Application providers can upload FRS to the NFV system in advance through the interfaces exposed by NFV-MANO, and provide the runtime information that the serverless application depends on in the serverless application descriptor.	The management capabilities of FRS need to be defined in a standardized way. Since the FRS is not included in the serverless application package, users should confirm that the NFV system is capable of providing the corresponding FRS before performing on-board operations.
Solution #2 PaaS service supporting triggering function instantiation	The solution can reuse the already defined PaaS service management functions to manage and create PaaS service instances for monitoring serverless function events, without the need to introduce new functional blocks, thereby reducing system complexity.	It is necessary to define the northbound interfaces exposed by PaaS services to configure the parameters for triggering serverless function instantiation. Given the diverse use cases and trigger conditions of FaaS functions, the interface definitions may become relatively complex.

Solution	Pros	Cons
Solution #3 Reducing cold start-up latency by runtime pool	Different applications can provide a maximum number of quick-start instances based on their requirements, thereby achieving a good trade-off between function startup efficiency and infrastructure resource utilization.	NFV-MANO needs to maintain the correspondence between pool and runtime. The responsibilities of VNFM are expanded to perform operations related to serverless application/functions management.
Solution #4 Modelling serverless application	The two-level design of serverless application packages and serverless function packages enhances operational flexibility, allowing application providers to offer updates within a limited scope by providing function packages, thereby avoiding the need to update the entire application package every time.	This solution increases the complexity of the management package. When users update the function package, NFV-MANO needs to update the corresponding function part in the application package based on the association relationships.
Solution #5 High-reliability Serverless application update	The solution allows for the dynamic adjustment of the usage ratio of different code versions by modifying configurations, facilitating code version control and rollback operations. The NFV system provides a publish-subscribe mechanism that enables consumers to monitor the operational status of function instances.	The management of serverless function code is relatively complex, requiring the simultaneous maintaining of multiple versions of the same function's code.
Solution #6 Rapid distribution of serverless function image	The solution achieves decentralized and rapid distribution of serverless function images through inter-node distribution, with minimal impact on NFV-MANO.	It is necessary to establish and maintain a decentralized distribution structure for each serverless function.

8 Recommendations

8.1 Overview

The recommendations are structured and elaborated as follows:

- aspects related to the architectural framework and functions (refer to clause 8.2);
- aspects related to the interfaces (refer to clause 8.3).

8.2 Recommendations related to the NFV architectural framework and functional aspects

The present clause documents recommendations intended to enhance the NFV system from following aspects:

- Identifying potential new functions or functional blocks, and interactions among these functional blocks and functions.
- Identifying new or extended functionalities of the NFV architectural framework functional blocks and functions.

Tables 8.2-1 provides recommendations related to the NFV architectural framework and functional aspects.

Table 8.2-1: Recommendations about NFV architectural framework and functional aspects

Identifier	Recommendation description	Comments and/or traceability
serverless.arch.001	It is recommended to specify a requirement for the NFV to support management of FRS.	Refer to gap #1.1
serverless.arch.002	It is recommended to specify a requirement for the NFV to support creation and maintenance runtime resource pool for the purpose of rapid serverless function instantiation.	Refer to gap #3.1
serverless.arch.003	It is recommended to specify a requirement for the NFV to support utilizing PaaS services to monitor corresponding events and trigger serverless function LCM operations.	Refer to gap #3.3
serverless.arch.004	It is recommended to specify a requirement for the NFV to support serverless function metric collection.	Refer to gap #5.2

8.3 Recommendations related to interfaces

The present clause provides recommendations focusing on interfaces and associated information.

Tables 8.3-1 provides the recommendations related to interfaces and associated information for serverless application.

Table 8.3-1: Recommendations about interfaces

Identifier	Recommendation description	Comments and/or traceability
serverless.if.001	It is recommended to specify a requirement for the interfaces exposed by NFV-MANO to be able to support the management of FRS.	Refer to gap #1.1
serverless.if.002	It is recommended to specify a requirement for the interfaces exposed by PaaS service to configure parameters used for triggering the LCM of serverless functions and selecting the function version.	Refer to gap #2.1 and gap 5.1
serverless.if.003	It is recommended to specify a requirement for the interfaces exposed by NFV MANO to be able to support runtime resource pool management.	Refer to gap #3.1
serverless.if.004	It is recommended to specify a requirement for the interfaces exposed by NFV MANO to be able to support the management of serverless applications packages. NOTE: A VNF can be mapped to a serverless application and a VNFC to a serverless function. In Release 6, different LCM capabilities are expected for VNFCs.	Refer to gap #4.5
serverless.if.005	It is recommended to specify a requirement for the interfaces exposed by NFV MANO to be able to support the OAM of serverless applications.	Refer to gap #5.2

9 Conclusion

The present document introduces serverless computing and other virtualisation forms (microVM, unikernel, WebAssembly, eBPF®), describing their characteristics, and presents and analyses typical use cases and key issues. Possible solutions are proposed for the identified key issues (focusing on serverless computing in the current version of the present document). Finally, recommendations for potential enhancements to the NFV architecture and interfaces are provided.

Annex A:
Change history

Date	Version	Information about changes
April 2024	V0.0.1	Skeleton early draft
June 2024	V0.0.2	Implementation of the skeleton contribution approved at EVE#290: - NFVEVE(24)000085r3: FEAT38 EVE025 Overview and what is serverless - NFVEVE(24)000086r3: FEAT38 EVE025 what are FaaS and BaaS - NFVEVE(24)000087r2: FEAT38 EVE025 Benefits and challenges analysis - NFVEVE(24)000088r3: FEAT38 EVE025 Clause 4 Adding unikernels - NFVEVE(24)000089r2: FEAT38 EVE025 Clause 4 Adding kata-containers
September 2024	V0.0.3	Implementation of the below contributions: - NFVEVE(24)000100r2: FEAT38 EVE025 Clause 4 Adding webassembly merging 93 and 100 - NFVEVE(24)000117r3: FEAT38 EVE025 UC onboarding and instantiating a serverless - NFVEVE(24)000123r2: FEAT38 EVE025 UC instantiate the function instance
October 2024	V0.0.4	Implementation of the below contributions: - NFVEVE(24)000139r1: FEAT38 EVE025 delete a serverless function instance - NFVEVE(24)000147: FEAT38 EVE025 add Key issue clause - NFVEVE(24)000132r1 FEAT38 EVE025 Clause 4 comparison of the virtualisation tech - NFVEVE(24)000108r4 FEAT38 EVE025 Clause 4 Adding eBPF - NFVEVE(24)000131r1 FEAT38 EVE025 Clause 4 Adding introduction
November 2024	V0.0.5	Implementation of the below contributions: - NFVEVE(24)000157r2: FEAT38 EVE025 KI code provision - NFVEVE(24)000158r3: FEAT38 EVE025 KI trigger management - NFVEVE(24)000159r2: FEAT38 EVE025 KI function management - NFVEVE(24)000167r1: FEAT38 EVE025 KI trigger type and implementation - NFVEVE(24)000179: FEAT38 EVE025 key issue infrastructure management - NFVEVE(24)000142r2: FEAT38 EVE025 Clause 4 Generalizing kata containers contribution to microVMs
February 2025	V0.0.6	Implementation of the below contributions: - NFVEVE(24)000198r1: FEAT38 EVE025 UC instantiating a VNF as eBPF program - NFVEVE(24)000209: FEAT38 EVE025 Key issues on serverless function code and package - NFVEVE(24)000210r1: FEAT38 EVE025 Key issues on management of serverless function - NFVEVE(24)000211r1: FEAT38 EVE025 Key issues on serverless enhancement - NFVEVE(24)000212r2: FEAT38 EVE025 Key issues on serverless computing model
April 2025	V0.0.7	Implementation of the below contributions: - NFVEVE(24)000215r1: FEAT38 EVE025 UC instantiating a VNF provided as a Wasm module - NFVEVE(25)000003r1: FEAT38 EVE025 Clause 4 Adding eBPF helpers - NFVEVE(25)000016r1: FEAT38 EVE025 UC onboarding VNFP of a VNF provided as a Wasm module - NFVEVE(25)000021r4: FEAT38 EVE025 Solution for FaaS runtime image management - NFVEVE(25)000033r2: FEAT38 EVE025 FaaS Runtime in Container Environment - NFVEVE(25)000034r2: FEAT38 EVE025 gap analysis for FaaS runtime - NFVEVE(25)000035r3: FEAT38 EVE025 solution triggering function instantiation
June 2025	V0.0.8	Implementation of the below contributions: - NFVEVE(25)000058r1: FEAT38 EVE025 gap analysis of triggering instantiation - NFVEVE(25)000061: FEAT38 EVE025 modify solution title - NFVEVE(25)000067r1: FEAT38 EVE025 solution for reduce cold start latency - NFVEVE(25)000090r1: FEAT38 EVE025 review of clause 4.3 - NFVEVE(25)000092: FEAT38 EVE025 review from clause 5.5 to 5.7 - NFVEVE(25)000096r1: FEAT38 EVE025 solution of modelling FaaS application - NFVEVE(25)000097r3: FEAT38 EVE025 high-reliability FaaS application update

Date	Version	Information about changes
July 2025	V0.0.9	Implementation of the below contributions: - NFVEVE(25)000089r2: FEAT38 EVE025 review of clause 3 and 4 - NFVEVE(25)000091r1: FEAT38 EVE025 review from clause 5.1 to 5.4 - NFVEVE(25)000093r1: FEAT38 EVE025 review of clause 6 - NFVEVE(25)000100r4: FEAT38 EVE025 review of solution 1 - NFVEVE(25)000101r1: FEAT38 EVE025 review of solution #2
September 2025	V0.0.10	Implementation of the below contributions: NFVEVE(25)000017r2: FEAT38 EVE025 UC onboarding VNFP of a VNF provided as eBPF program NFVEVE(25)000121r2: FEAT38 EVE025 onboard a serverless app package NFVEVE(25)000125r2: FEAT38 EVE025 Rapid Distribution of Serverless Function Image NFVEVE(25)000129r1: FEAT38 EVE025 4.1 overview NFVEVE(25)000130r1: FEAT38 EVE025 EN regarding microservice and FaaS NFVEVE(25)000131r1: FEAT38 EVE025 5.1 overview NFVEVE(25)000134r1: FEAT38 EVE025 eBPF KI NFVEVE(25)000144: FEAT38 EVE025 remove EN in clause 7.2.3.4 NFVEVE(25)000145: FEAT38 EVE025 7.1 introduction NFVEVE(25)000146r1: FEAT38 EVE025 Evaluation NFVEVE(25)000147r2: FEAT38 EVE025 recommendations on architecture
October 2025	V0.0.11	Implementation of the below contributions: NFVEVE(25)000153r2: FEAT38 EVE025 recommendations regarding interface NFVEVE(25)000162r1: FEAT38 EVE025 scope NFVEVE(25)000163: FEAT38 EVE025 8.1 overview NFVEVE(25)000166: FEAT38 EVE025 Clause 6 WASM key issues NFVEVE(25)000167r1: FEAT38 EVE025 Clause 6 MicroVMs key issues NFVEVE(25)000170: FEAT38 EVE025 Clause 6 unikernels key issues NFVEVE(25)000171r1: FEAT38 EVE025 Conclusion
December 2025	V0.0.12	Implementation of the below contributions: NFVEVE(25)000186r1: FEAT38 EVE025 review from clause 3 to clause 5 NFVEVE(25)000191: FEAT38 EVE025 review from clause 6 to clause 9 NFVEVE(25)000226: FEAT38 EVE025 review clause 1 to clause 4 feedback from doc213 NFVEVE(25)000214r1: FEAT38 EVE025 review clause 5 feedback NFVEVE(25)000215r1: FEAT38 EVE025 review clause 6 feedback NFVEVE(25)000216r1: FEAT38 EVE025 review clause 7 feedback NFVEVE(25)000217: FEAT38 EVE025 review clause 8 feedback

History

Version	Date	Status
V6.1.1	March 2026	Publication