



GROUP REPORT

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

get full document from standards.iteh.ai

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

Important notice

The present document can be downloaded from the
[ETSI Search & Browse Standards](#) application.

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the prevailing version of an ETSI deliverable is the one made publicly available in PDF format on [ETSI deliver](#) repository.

Users should be aware that the present document may be revised or have its status changed,
this information is available in the [Milestones listing](#).

If you find errors in the present document, please send your comments to
the relevant service listed under [Committee Support Staff](#).

If you find a security vulnerability in the present document, please report it through our
[Coordinated Vulnerability Disclosure \(CVD\)](#) program.

Notice of disclaimer & limitation of liability

The information provided in the present deliverable is directed solely to professionals who have the appropriate degree of experience to understand and interpret its content in accordance with generally accepted engineering or other professional standard and applicable regulations.

No recommendation as to products and services or vendors is made or should be implied.

No representation or warranty is made that this deliverable is technically accurate or sufficient or conforms to any law and/or governmental rule and/or regulation and further, no representation or warranty is made of merchantability or fitness for any particular purpose or against infringement of intellectual property rights.

In no event shall ETSI be held liable for loss of profits or any other incidental or consequential damages.

Any software contained in this deliverable is provided "AS IS" with no warranties, express or implied, including but not limited to, the warranties of merchantability, fitness for a particular purpose and non-infringement of intellectual property rights and ETSI shall not be held liable in any event for any damages whatsoever (including, without limitation, damages for loss of profits, business interruption, loss of information, or any other pecuniary loss) arising out of or related to the use of or inability to use the software.

Copyright Notification

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.

The content of the PDF version shall not be modified without the written authorization of ETSI.

The copyright and the foregoing restriction extend to reproduction in all media.

© ETSI 2026.
All rights reserved.

Contents

Intellectual Property Rights	6
Foreword.....	6
Modal verbs terminology.....	6
1 Scope	7
2 References	7
2.1 Normative references	7
2.2 Informative references.....	7
3 Definition of terms, symbols and abbreviations.....	8
3.1 Terms.....	8
3.2 Symbols.....	8
3.3 Abbreviations	8
4 Overview and background.....	9
4.1 Overview	9
4.2 Serverless and Function as a Service.....	9
4.2.1 What is Serverless computing.....	9
4.2.2 Introduction to FaaS.....	9
4.2.2.1 What is FaaS	9
4.2.2.2 Advantages and challenges of FaaS	10
4.2.2.3 FaaS runtime	11
4.3 Overview on new virtualisation technologies.....	11
4.3.1 Introduction.....	11
4.3.2 Unikernels.....	12
4.3.3 MicroVMs.....	12
4.3.3.1 Overview	12
4.3.3.2 Usage of MicroVMs.....	12
4.3.3.3 Limitations of MicroVMs	13
4.3.3.4 Example of MicroVMs	13
4.3.4 WebAssembly	13
4.3.5 eBPF	14
4.3.5.1 Introduction.....	14
4.3.5.2 How eBPF works	15
4.3.5.3 eBPF for VNF deployments in the telco Cloud.....	15
4.3.5.4 eBPF limitations.....	16
4.3.6 Comparison of different virtualisation technologies.....	16
5 Use cases	18
5.1 Overview	18
5.2 Use case #1: Onboarding a serverless application package.....	18
5.2.1 Introduction.....	18
5.2.2 Actors and roles	18
5.2.3 Trigger	19
5.2.4 Pre-conditions	19
5.2.5 Post-conditions	19
5.2.6 Flow description	19
5.3 Use case #2: On-boarding a serverless function package.....	20
5.3.1 Introduction.....	20
5.3.2 Actors and roles	20
5.3.3 Trigger	20
5.3.4 Pre-conditions	20
5.3.5 Post-conditions	21
5.3.6 Flow description	21
5.4 Use case #3: Instantiating a serverless function	21
5.4.1 Introduction.....	21
5.4.2 Actors and roles	22
5.4.3 Trigger	22

5.4.4	Pre-conditions	22
5.4.5	Post-conditions	22
5.4.6	Flow description	22
5.5	Use case #4: Deleting a serverless function instance	23
5.5.1	Introduction.....	23
5.5.2	Actors and roles	23
5.5.3	Trigger	23
5.5.4	Pre-conditions	24
5.5.5	Post-conditions	24
5.5.6	Flow description	24
5.6	Use case #5: Onboarding the package of a VNF with eBPF® components	25
5.6.1	Introduction.....	25
5.6.2	Actors and roles	25
5.6.3	Trigger	25
5.6.4	Pre-conditions	25
5.6.5	Post-conditions	25
5.6.6	Flow description	26
5.7	Use case #6: instantiating a VNF as an eBPF program	26
5.7.1	Introduction.....	26
5.7.2	Actors and roles	26
5.7.3	Trigger	27
5.7.4	Pre-conditions	27
5.7.5	Post-conditions	27
5.7.6	Flow description	27
5.8	Use case #7: Onboarding the VNF package of Wasm-based VNF	28
5.8.1	Introduction.....	28
5.8.2	Actors and roles	28
5.8.3	Trigger	28
5.8.4	Pre-conditions	28
5.8.5	Post-conditions	28
5.8.6	Flow description	29
5.9	Use case #8: instantiating a Wasm-based VNF	29
5.9.1	Introduction.....	29
5.9.2	Actors and roles	29
5.9.3	Trigger	30
5.9.4	Pre-conditions	30
5.9.5	Post-conditions	30
5.9.6	Flow description	30
6	Key issue analysis	31
6.1	Introduction	31
6.2	Key issues on serverless computing model	31
6.3	Key issues on serverless applications and functions packages and descriptors.....	31
6.4	Key issues on management of serverless applications and functions.....	32
6.5	Key issues on deploying serverless application efficiently	33
6.6	Key issues related to new virtualisation technologies	33
6.6.1	Key issues related to eBPF®	33
6.6.2	Key issues related to Wasm	34
6.6.3	Key issues related to microVMs	35
6.6.4	Key issues related to unikernels.....	36
7	Analysis and potential solutions.....	36
7.1	Introduction	36
7.2	Potential solutions	37
7.2.1	Solution #1: FaaS runtime software management	37
7.2.1.1	FaaS runtime software repository functionality	37
7.2.1.2	FaaS runtime in container environment	38
7.2.1.3	Related issues	38
7.2.1.4	Gap Analysis.....	38
7.2.1.4.1	FaaS function direct deployment on the host.....	38
7.2.1.4.2	FaaS function deployment based on Container	39
7.2.2	Solution #2: PaaS service supporting triggering function instantiation	39

7.2.2.1	Introduction	39
7.2.2.2	Solution description	39
7.2.2.3	Key issues address	41
7.2.2.4	Gap analysis	41
7.2.3	Solution #3: Reducing cold start-up latency by runtime resource pool	41
7.2.3.1	Introduction	41
7.2.3.2	Solution description	41
7.2.3.3	Key issues address	42
7.2.3.4	Gap analysis	42
7.2.4	Solution #4: Modelling serverless application	43
7.2.4.1	Introduction	43
7.2.4.2	Solution description	43
7.2.4.3	Key issues address	44
7.2.4.4	Gap analysis	44
7.2.5	Solution #5: High-reliability Serverless application update	44
7.2.5.1	Introduction	44
7.2.5.2	Solution description	44
7.2.5.3	Key issues address	45
7.2.5.4	Gap analysis	45
7.2.6	Solution #6: Rapid distribution of serverless function image	46
7.2.6.1	Introduction	46
7.2.6.2	Solution description	46
7.2.6.3	Key issues address	47
7.2.6.4	Gap analysis	47
7.3	Evaluation of solutions	47
8	Recommendations	48
8.1	Overview	48
8.2	Recommendations related to the NFV architectural framework and functional aspects	48
8.3	Recommendations related to interfaces	49
9	Conclusion	49
Annex A:	Change history	50
	History	52

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.

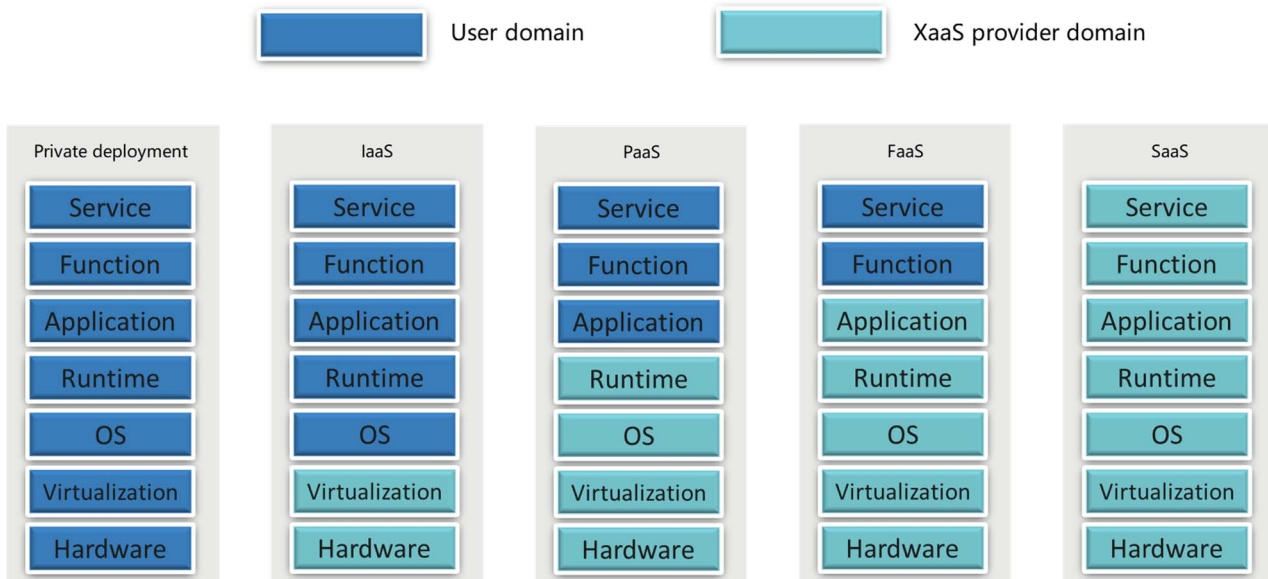


Figure 4.2.2.1-1: XaaS Comparison

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.

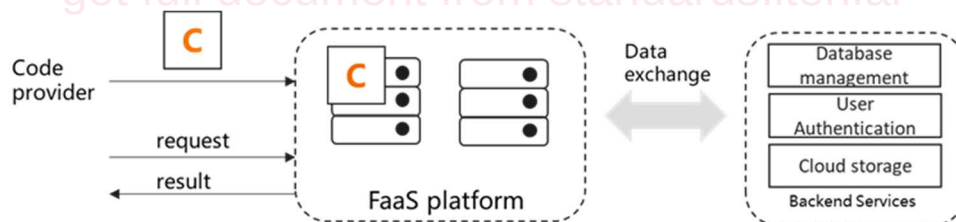


Figure 4.2.2.1-2: FaaS Architecture

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.

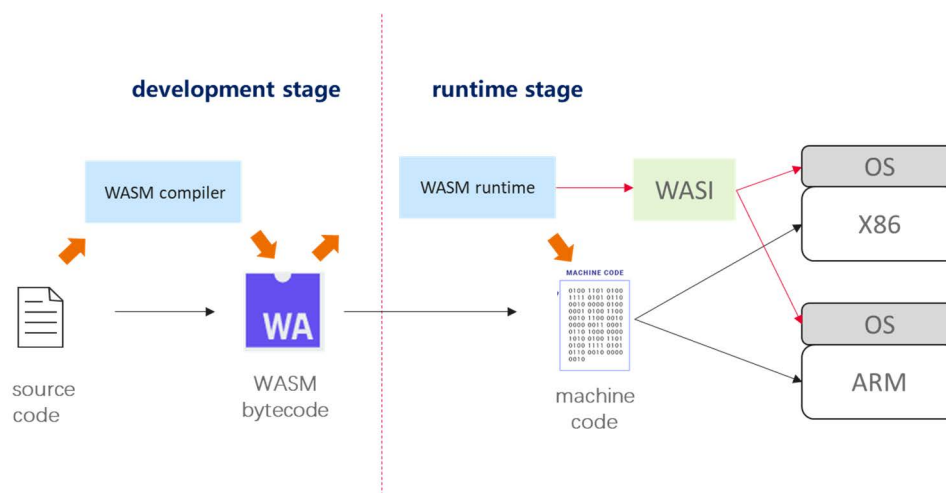


Figure 4.3.4-1: Wasm workflow

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.

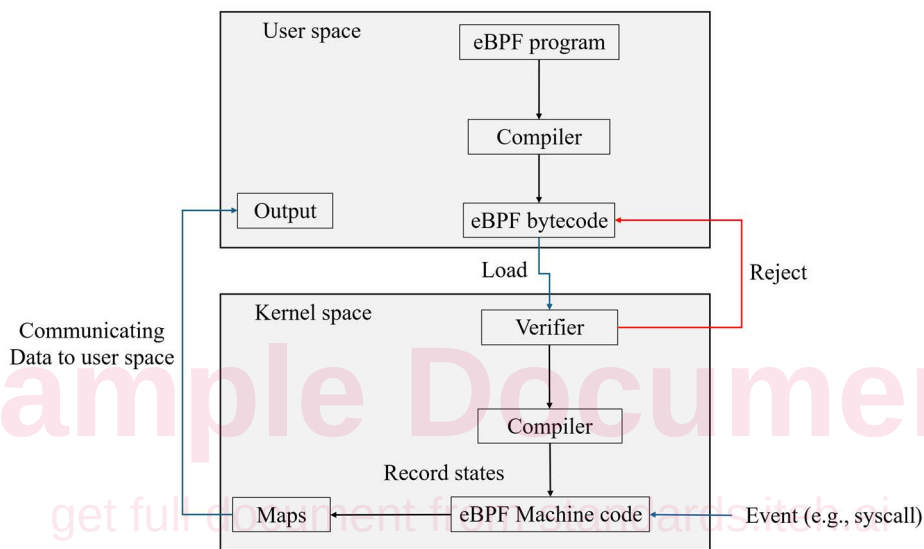


Figure 4.3.5.2-1: eBPF workflow

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.