TECHNICAL SPECIFICATION

# ISO/IEC TS 23167

First edition
2020-02

# Information technology — Cloud computing — Common technologies and techniques



Reference number
ISO/IEC TS 23167:2020(E)



ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

# Contents

Page

 

# Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see http://patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 38, *Cloud Computing and Distributed Platforms*.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

# Introduction

Cloud computing is described at a high, conceptual level in the two foundational standards ISO/IEC 17788 [1] and ISO/IEC 17789 [2].

However, as the use of cloud computing has grown, a set of commonly used technologies has grown to support, simplify and extend the use of cloud computing alongside sets of commonly used techniques which enable the effective exploitation of the capabilities of cloud services. Many of these common technologies and techniques are aimed at developers and operations staff, increasingly linked together in a unified approach called DevOps (see 10.2). The aim is to speed and simplify the creation and operation of solutions based on the use of cloud services.

This document aims to describe the common technologies and techniques which relate to cloud computing, to describe how they relate to each other and to describe how they are used by some of the roles associated with cloud computing.

This document (a Technical Specification) addresses areas that are still developing in the industry, where it is believed that there will be a future, but not immediate, need for one or more International Standards.

This document will be of primary interest to service developers in Cloud Service Providers and to standards developers working with ISO and other organizations.

# Information technology — Cloud computing — Common technologies and techniques

## 1 Scope

This document provides a description of a set of common technologies and techniques used in conjunction with cloud computing. These include:

— virtual machines (VMs) and hypervisors;

— containers and container management systems (CMSs);

— serverless computing;

— microservices architecture;

— automation;

— platform as a service systems and architecture;

— storage services;

— security, scalability and networking as applied to the above cloud computing technologies.

## 2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 22123-1:—[1)], *Information technology — Cloud computing — Part 1:Terminology*

## 3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 22123-1 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at http://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

**3.1**
**guest operating system**
**guest OS**
operating system that runs within a virtual machine

[SOURCE: ISO/IEC 21878:2018, 3.2]

1) To be published.

**3.2**
**host operating system**
**host OS**
operating system onto which virtualization software is installed

Note 1 to entry: "virtualization software" can include both hypervisor and virtual machines as well as container daemon (3.4) and containers.

**3.3**
**serverless computing**
cloud service category in which the cloud service customer can use different cloud capabilities types without the cloud service customer having to provision, deploy and manage either hardware or software resources, other than providing cloud service customer application code or providing cloud service customer data

Note 1 to entry: Serverless computing provides automatic scaling with dynamic elastic allocation of resources by the cloud service provider, automatic distribution across multiple locations, and automatic maintenance and backup.

Note 2 to entry: Serverless computing functionality is triggered by one or more cloud service customer defined events and can execute for a limited time period as required to deal with each event.

Note 3 to entry: Serverless computing functionality can be invoked by direct invocation from web and mobile applications.

**3.4**
**container daemon**
software service that executes on a host operating system (3.2) and is responsible for creating, starting and stopping containers on that system

**3.5**
**container management system**
**CMS**
software that provides for management and orchestration of container instances

Note 1 to entry: Capabilities include initial creation and placement, scheduling, monitoring, scaling, update and the parallel deployment of capabilities such as load balancers, firewalls, virtual networks and logging.

**3.6**
**cloud native application**
application that is explicitly designed to run within and to take advantage of the capabilities and environment of cloud services

**3.7**
**functional decomposition**
type of modular decomposition in which a system is broken down into components that correspond to system functions and subfunctions

EXAMPLE Hierarchical decomposition, stepwise refinement.

[SOURCE: ISO/IEC/IEEE 24765:2017, 3.1695]

**3.8**
**continuous deployment**
software engineering approach in which teams produce software in short cycles such that the software can be released to production at any time and where deployment to production is itself automated

**3.9**
**continuous delivery**
continuous deployment (3.8) where the deployment stage is initiated manually

**3.10**
**DevOps**
methodology which combines together software development and IT operations in order to shorten the development and operations lifecycle

**3.11**
**DevSecOps**
DevOps (3.10) extended to include security capabilities as an essential and integral part of the development and operations processes

**3.12**
**orchestration**
type of composition where one particular element is used by the composition to oversee and direct the other elements

Note 1 to entry: The element that directs an orchestration is not part of the orchestration (composition instance) itself.

Note 2 to entry: See ISO/IEC 18384-3:2016, 8.3.

[SOURCE: ISO/IEC 18384-1:2016, 2.16]

**3.13**
**virtual machine image**
**VM image**
information and executable code necessary to run a virtual machine

**3.14**
**virtual machine metadata**
**VM metadata**
information about the configuration and startup of a virtual machine

**3.15**
**microservice**
independently deployable artefact providing a service implementing a specific functional part of an application

**3.16**
**microservices architecture**
design approach that divides an application into a set of microservices (3.15)

**3.17**
**functions as a service**
**function as a service**
**FaaS**
cloud service category in which the capability provided to the cloud service customer is the execution of cloud service customer application code, in the form of one or more functions that are each triggered by a cloud service customer specified event

**3.18**
**serverless database**
cloud service category in which the capability provided to the cloud service customer is a fully cloud service provider managed database made available via an application programming interface

**3.19**
**firewall**
type of security barrier placed between network environments — consisting of a dedicated device or a composite of several components and techniques — through which all traffic from one network environment traverses to another, and vice versa, and only authorized traffic, as defined by the local security policy, is allowed to pass

[SOURCE: ISO/IEC 27033-1:2015, 3.12]

**3.20**
**container registry**
component that provides the capability to store and to access container images

**3.21**
**resource affinity**
placement of two or more resources close to each other

Note 1 to entry: Closeness relates to factors such as speed of access or high bandwidth of access between the resources.

## 4 Symbols and abbreviated terms

| | |
|---|---|
| API | Application programming interface |
| CMS | Container management system |
| CSC | Cloud service customer |
| CSP | Cloud service provider |
| DNS | Domain name service |
| GUI | Graphical user interface |
| HTTP | Hypertext transfer protocol |
| IaaS | Infrastructure as a service |
| IP | Internet protocol |
| MAC | Media access control |
| OCI | Open containers initiative |
| OS | Operating system |
| OVF | Open virtualization format |
| PaaS | Platform as a service |
| SaaS | Software as a service |
| VPN | Virtual private network |

## 5 Overview of common technologies and techniques used in cloud computing

### 5.1 General

This document provides a description of a set of common technologies and techniques used in conjunction with cloud computing.

A common technology is one that is used to implement one or more of the functional components of cloud computing described in ISO/IEC 17789:2014,9.2[2] cloud computing reference architecture. The common technologies often form part of a cloud service or are employed by the cloud service customer (CSC) when using a cloud service.

A common technique is a methodology or an approach to performing some of the activities associated with cloud computing, as described in ISO/IEC 17789:2014,10.2.2[2]. It is typical of the common techniques to either reduce the effort needed to make use of cloud services or to enable full use of the capabilities provided by cloud services.

Many of the common technologies and techniques are used in conjunction when developing and operating cloud native applications.

The various common technologies and techniques are described in detail in the following clauses.

In what follows, text that is extracted from other standards are indicated by placing the extracted text in quotes, using italic text, and providing the exact reference at the end of the extracted text.

## 5.2 Technologies

### 5.2.1 General

The common technologies principally relate to virtualization and the control and management of virtualized resources in the development and operation of cloud native applications. A cloud native application is an application that is explicitly designed to run within and to take advantage of the capabilities and environment of cloud services. These technologies address the three primary hardware resources identified in ISO/IEC 17789:2014,9.2.4.2[2] of processing, storage and networking but also address the platform capabilities type of cloud service. These technologies include:

— Virtualized processing is addressed by virtual machines (see Clause 6), by containers (see Clause 7), by serverless computing (see Clause 8).

— Virtualized storage is addressed by means of a variety of Data Storage as a Service (see Clause 12).

— Virtualized networking is one of the primary groups of technologies for the provision and use of networking capabilities in relation to cloud services (see Clause 13).

— The Platform as a Service category of cloud services are designed to enable more rapid development, testing and production of cloud native applications (see Clause 11).

Security and scalability technologies apply generally across all types of cloud services, although the explicit use of the technologies by the CSC is more common for some types of cloud service (see Clause 14 and Clause 15).

### 5.2.2 Infrastructure capabilities type of cloud services

Technologies commonly used with infrastructure capabilities type of cloud services include:

— virtual machines;

— containers;

— virtualized storage;

— virtualized networking;

— security.

### 5.2.3 Platform capabilities cloud services

Technologies commonly used with platform capabilities type of cloud services include:

— containers;

— serverless computing;

— PaaS cloud services;

— virtualized storage;

— virtualized networking;

— security.

### 5.2.4 Application capabilities type cloud services

Technologies commonly used with application capabilities type of cloud services include:

— virtualized storage;

— virtualized networking;

— security.

## 5.3 Techniques

The common techniques typically apply to all cloud service categories, although some techniques are more useful with some categories of cloud service than others.

Orchestration and management of virtualized resources is achieved with tooling, including CMSs (see Clause 10 and 7.4).

Techniques commonly used with cloud computing include:

— Automation of various kinds, applied throughout the DevOps processes (see Clause 10).

— Scalability approaches such as parallel instances (see Clause 14).

— Microservices design approach to applications and systems (see Clause 9).

— Firewalls, encryption, and Identity and Access Management (IAM) techniques for security and protection of privacy (see Clause 15).

# 6 Virtual machines and hypervisors

## 6.1 General

Virtual machines and hypervisors are technologies that provide virtualized processing (also known as virtualized compute) for cloud services. These technologies primarily relate to cloud services of infrastructure capabilities type and IaaS as described in ISO/IEC 17788 and ISO/IEC 17789.

One of the key characteristics of cloud computing is its ability to share resources. This is fundamental to its economics, but it is also important to characteristics such as scalability and resilience. Sharing of processing resources requires some level of virtualization. Virtualization in general means that some resource is made available for use in a form that does not physically exist as such but which is made to appear to do so by software. In other words, virtualization provides an abstraction of the underlying resource, being converted into a software defined form for use by other software entities. The software performing the virtualization enables multiple users to simultaneously share the use of a single physical



 

resource without interfering with each other and usually without them being aware of each other. (See ISO/IEC 22123-1:—, 5.5).

One approach to the virtualization of processing resources is the use of virtual machines, which involves a hypervisor providing an abstraction of the system hardware and permitting multiple virtual machines to run on a given physical system, with each VM containing its own guest operating system (guest OS), as shown in Figure 1. This permits the system to be shared by the applications running in each VM.

The hypervisor is typically software that is installed and operated by the CSP. The cloud service that runs the VM offers the capability for the CSU to load software from a VM image and run the software within a VM on the CSP system. The VM is managed by the hypervisor, but this is not seen directly by the CSU.

## 6.2 Virtual machines and system virtualization

*A virtual machine (VM) is an isolated execution environment for running software that uses virtualized physical resources. In other words, this involves the virtualization of the system – and the software within each VM is given carefully controlled access to the physical resources to enable sharing of those resources without interference. Sometimes termed system virtual machines, VMs provide the functionality needed to execute complete software stacks including entire operating systems and the application code that uses the operating system* (ISO/IEC 22123-1:—, 5.5.1). This is as depicted by the "guest OS" and "App x" within each VM shown in Figure 1.

The purpose of VMs is to enable multiple applications to run at the same time on one hardware system, while those applications remain isolated from each other. The software running within each VM appears to have its own system hardware, such as processor, runtime memory, storage device(s) and networking hardware. Isolated means that the software running within one VM is separated from and unaware of software running within other VMs on the same system and is also separated from the host OS. Virtualization commonly means that a subset of the available physical resources can be made available to each VM, such as limited numbers of processors, limited RAM, limited storage space and controlled access to networking capabilities.

Each VM contains a complete stack of software, starting with the operating system and continuing with whatever other software is required to run the application(s) that are executed within the VM. The software stack could be very simple (e.g. a native application written in a language like C, using only functions supplied by the operating system itself) or complex (e.g. an application written in a language such as Java™ which requires a runtime and which makes extensive use of libraries and/or services which are not present in the operating system and which have to be supplied separately).

Each VM can in principle contain any operating system. Different VMs on a single hardware system can run completely different operating systems such as Linux® and Windows®. The only requirement is that all the software running within the VM is designed for the hardware architecture of the underlying system – the hardware is virtualized, but not emulated. So, for example, code built for an ARM processor will not run in a VM running on an Intel x86 system.

## 6.3 Hypervisors

### 6.3.1 General

*The hypervisor, sometimes termed a virtual machine monitor, is software that virtualizes physical resources and allows for running virtual machines. Virtualization means control of the abstraction of the underlying physical resources of the system. The hypervisor also manages the operation of the VMs. The hypervisor allocates resources to each running VM including processor (CPU), memory, disk storage and networking capabilities and bandwidth* (ISO/IEC 22123-1).

Hypervisors exist as one of two types:

— "Bare metal", "native" or "type I";

— "Embedded", "hosted" or "type II".

Type I hypervisors can be faster and more efficient, since they do not need to work via a host operating system. Type II hypervisors may be slower, but have the advantage of being typically easier to set up and are compatible with a broader range of hardware than type I hypervisors, since hardware variations have to be dealt with in the type I hypervisor code, whereas the type II hypervisors take advantage of the hardware support built in to the host operating system.

### 6.3.2 Type I hypervisors

Type I hypervisors run directly on the underlying system hardware and control that hardware directly as well as managing the VMs. The organization of a system using a Type I hypervisor is shown in Figure 1.

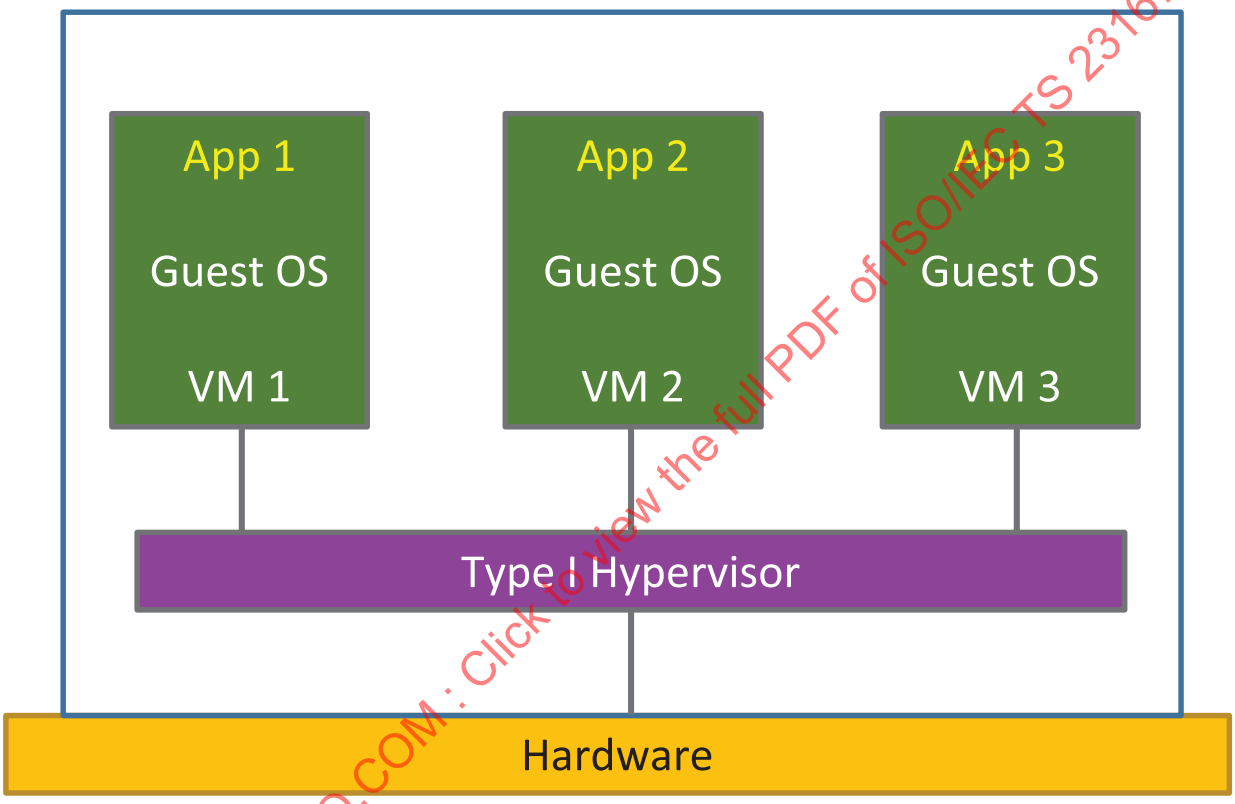


**Figure 1 — Type I hypervisor virtualization of system hardware**



### 6.3.3 Type II hypervisors

Type II hypervisors run on top of a host operating system, more specifically the host OS kernel. It is the host operating system that controls the system hardware, while the hypervisor makes use of its capabilities to run and manage the VMs. The organization of a system with a Type II hypervisor is shown in Figure 2.

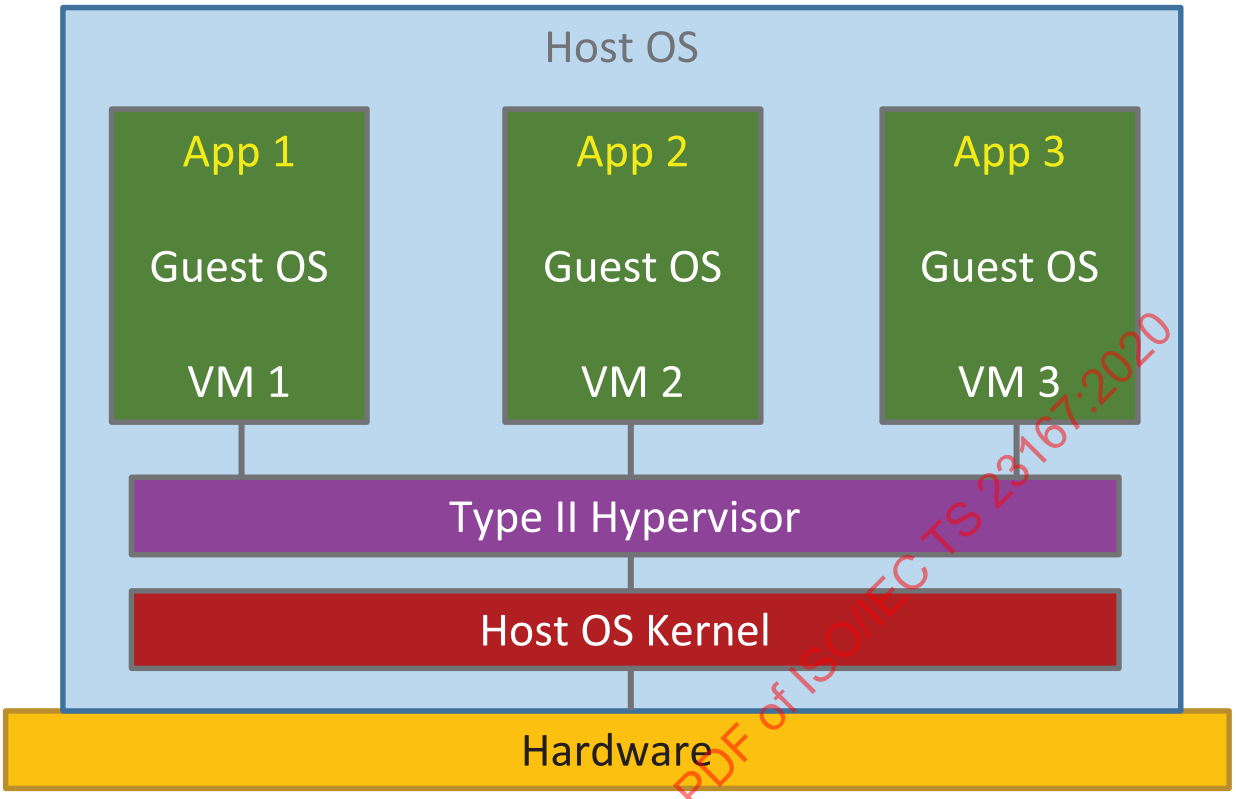


**Figure 2 — Type II hypervisor virtualization of system hardware**



### 6.4 Security of VMs and hypervisors

For hardware systems, the operating system runs at the highest privilege level since it must control access to all hardware resources. However, in a hypervisor host, since the hypervisor must control all access to CPU and memory by guest VMs (providing processor and memory virtualization), it should run at a privilege level higher than all VMs. To facilitate this, hypervisors are installed on hardware systems that provide assistance for virtualization. Specifically, the hardware system provides two processor states: root (hypervisor) mode and non-root (guest) mode. All guest OSs run in non-root mode while the hypervisor alone runs in root mode.

Despite the hardware support for virtualization, the runtime process isolation for VMs provided by the hypervisor could be subverted by rogue or compromised VMs which have gained access to areas of memory belonging to the hypervisor or other VMs. Rogue or compromised VMs exploit certain hypervisor design vulnerabilities with respect to certain software structures such as virtual machine control block (VMCB) and memory page tables which are used by the hypervisor to keep track of the execution state of VMs and memory mapping from VM addresses to host memory addresses respectively. These vulnerabilities of hypervisors have been known for some time and as a result, many of the vulnerabilities have been addressed or are being addressed. More recent hypervisor versions have been updated and hardened. The CSC and CSP should check that any hypervisors in use are up-to-date and hardened against known security vulnerabilities.

Another security implication in a hypervisor host platform stems from software used for providing device virtualization. Unlike instruction set and memory virtualization, device virtualization is not directly handled by the hypervisor but by using supporting software modules. Primary sources of vulnerabilities include: (a) code emulating physical hardware devices running in the hypervisor as a loadable kernel module and (b) device drivers for direct memory access (DMA) capable devices which can access memory regions belonging to other VMs or even the hypervisor.

Potential downstream impacts of a rogue VM taking control of the hypervisor include the installation of rootkits or attacks on other VMs on the same hypervisor host. All device virtualization software should be verified against security flaws before installation and use on a system using a hypervisor and VMs.

## 6.5 VM images, metadata and formats

A virtual machine image (VM image) is a package of data that contains the information and executable code necessary to run an instance of a VM. The VM image is used to instantiate a new instance of a VM, as required. The VM image can include the complete software stack required to run an application, starting with the operating system, libraries, runtimes, the application code itself, configuration files and other metadata used by the application. The VM image can also include metadata associated with the instantiation of the VM itself.

The VM metadata contains information about the configuration and startup of the VM. This might include properties of the VM such as RAM size, CPU requirements and so on. The VM metadata also typically references the disk images contained in the VM image, in particular indicating how they are deployed into a VM instance.

The concept of the VM image is that it should contain all the entities required to run an instance of a VM, where the VM image is used as input data to a hypervisor to enable it to create and start the VM. Broadly, the VM image consists of two sets of data – firstly, VM metadata and secondly disk images. It is important to understand that there are in existence many different formats of both VM metadata and disk images. A particular hypervisor used to instantiate a VM might only understand specific formats for the VM metadata and disk images. Some of the formats are proprietary, while others are open or standardised. See Annex A for information about VM image formats.

VM images are based on data held in files – files on filesystems, which are held in the VM image as one or more disk images. These files can be those of the operating system, the application and any other part of the software stack that is required. There is at least one disk image, but there can be multiple disk images if this is the organization of files that is used by the application and its software stack. It is often the case that the volume of data held in the disk images is very large and as a result, the formats used to store the data involve the use of compression in one form or another.

There are many VM image and disk image formats in use, a substantial proportion of which are proprietary or which are open source. Examples of standardised VM image and disk image formats include:

— OVF ("Open Virtualization Format") (see ISO/IEC 17203:2017[18])

  The OVF package has a number of files placed in a single directory. There is an OVF descriptor file (with extension .ovf) which has XML format contents describing the packaged virtual machine including the metadata such as the name, hardware requirements and references to the other files in the package. The OVF package also contains one or more disk images, plus some optional files such as certificate files. The OVF image format has a relatively wide range of support, either directly or via import/export tools.

— ISO disk format – the archive format used for optical disc contents (see ISO 9660[72] and ISO/IEC 13346[73])

  ISO 9660 is a file system for optical disk media, principally CD-ROMs.

  ISO/IEC 13346 (also known as Universal Disk Format or UDF) is often used on DVDs and Blu-ray disk (BD) formats and is particularly suited to recordable and (re)writable optical media.

Disk images are commonly compressed due to their large size, although there are cases where raw uncompressed disk images are used to obtain better VM start up performance at the cost of consuming more space.

Which VM image and disk image formats are accepted by a particular hypervisor are stated in the documentation for the hypervisor.



 

# 7 Containers and container management systems (CMSs)

## 7.1 General

Containers are a technology that can provide virtualized processing for cloud services. The technology relates both to infrastructure capabilities type and to platform capabilities type of cloud services as described in ISO/IEC 17788 and ISO/IEC 17789.

Containers provide a software execution environment through virtualization of the operating system kernel running on a system. Containers represent another approach to the provision of a software execution environment using the virtualization of compute resources. Containers involve the virtualization of the OS kernel, as compared with the virtualization of the system hardware involved with VMs. The goal of containers is to permit multiple different sets of software to run on a single system at the same time without interfering with each other, i.e. they permit secure sharing of the system.

A cloud service supporting containers offers the capability for the CSU to load software from a container image and run that software within a container on the CSP system. The container is managed either by the CSP or by CSU, depending on the capabilities type of the cloud service. In either case, it is typical that management is performed by means of a CMS (see 7.4).

## 7.2 Containers and operating system virtualization

### 7.2.1 Description of containers

A container is an isolated execution environment for running software that uses a virtualized operating system kernel. Containers run within an operating system which is termed the host operating system (host OS).

As described in 6.2 a VM presents a virtualized version of the system hardware to the software within the VM. Access to the virtualized hardware resources is mediated and the software within the VM only gets to see and use a carefully controlled and limited version of those resources (e.g. limited number of CPUs, limited number of threads, limited RAM).

In the same way, a container presents a virtualized version of the host OS kernel. Access to the virtualized resources of the OS kernel is mediated and the software within the container gets to see and use a carefully controlled and limited version of the OS resources.

The isolation of the execution environment means that the software running within one container is separated from and unaware of software running in other containers, and is also separated from the host OS. The only software running outside a container that can access or affect software running inside a container is the container daemon.

Figure 3 shows three containers running on the system hardware. The physical system has its own host OS. Each container contains its own application software (App x), and runs that software in one or more OS processes using resources such as memory, CPU, storage and networking, isolated from the other containers running on the same system, but all sharing the kernel of the host OS.

The kernel of the host OS is being shared by all the containers, which essentially means that the OS used by the software in the containers must be compatible with the host OS kernel. This can allow for the different containers to potentially use different variants of the Linux OS where the host OS is a Linux variant, for example, but it does not allow for the Windows OS to be used within a container if the host OS is a Linux variant (and vice-versa).

The containers are created and managed by a container daemon, which runs as a separate process in the host OS.

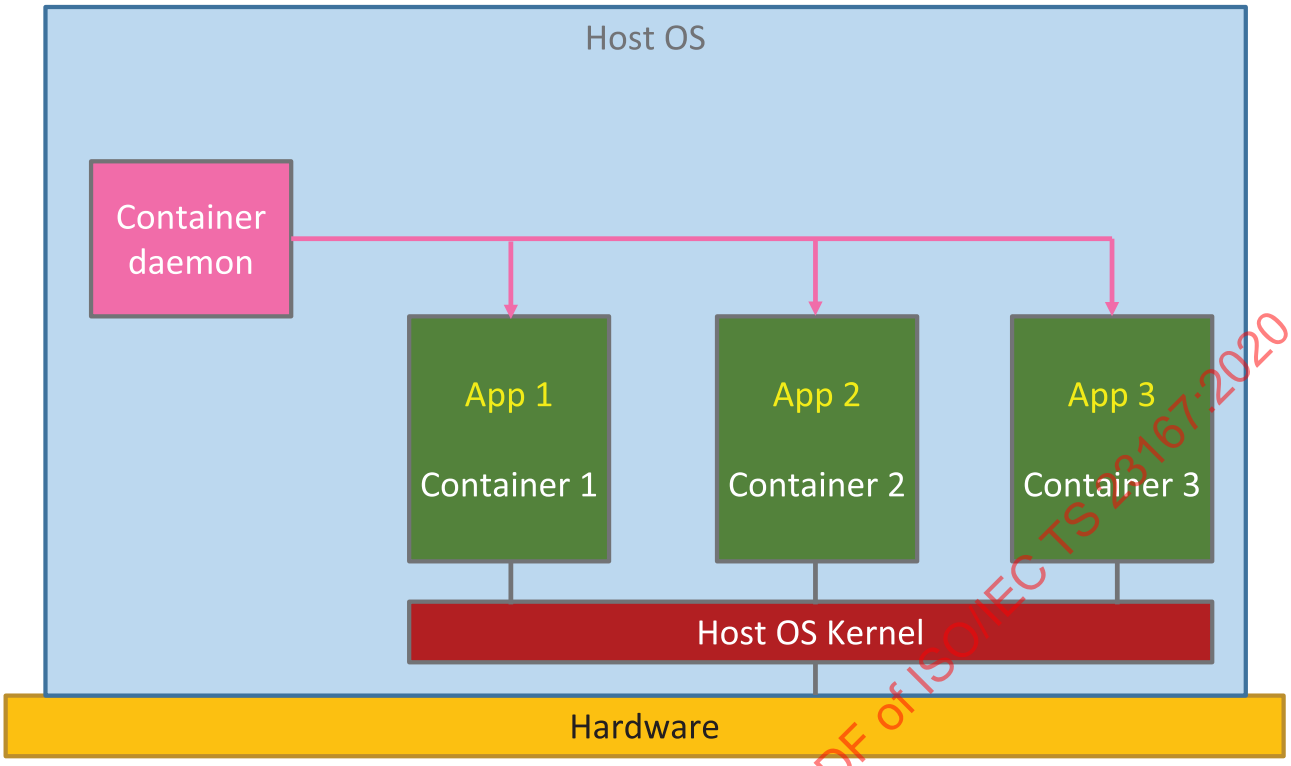


**Figure 3 — Container virtualization**

The software stack running in each container can vary, but typically it contains the application itself ("App x") and whatever software dependencies that the application has. In principle, the software stack could be quite "lean", especially where the application code depends only on functions supplied by the host OS kernel. However, it is the case that the container code can include elements of the OS outside the kernel, such as libraries and utilities, particularly if the application code depends on specific versions of these libraries and utilities.

Note that the host operating system used by containers could itself be running inside a virtual machine, rather than directly on the physical hardware. Software running in the containers is unaware of whether the host operating system is running in a virtual machine or not.

### 7.2.2 Container daemon

The container daemon is a software service that executes on the host operating system and which is responsible for creating and managing containers on that system. A particular container is an executable instance of a software stack that is held within a container image (see 7.3 for a description of container images). The container image includes metadata and parameters used by the container daemon. The container daemon uses the container metadata to specify certain capabilities of the container and it uses the container service parameters to affect how a container is instantiated and run.

The container daemon offers a service interface through which its capabilities can be invoked by client applications. Client applications can run on the same system as the container daemon or can run on remote systems and invoke the capabilities of the container daemon over the network.

The container daemon offers a set of container operations:

— **Create**: this operation creates a new container. The operation references the container image to use, instantiated in a filesystem directory (termed a "bundle") accessible by the host operating system. Creating the container allocates a set of resources to the container and configures the container as described in the container metadata held in the bundle. The container is given a unique ID, by which it is later referenced.



 

— **Start**: this operation starts the container by running the application program specified for the container, with whatever parameters are supplied by the metadata relating to the container.

— **Kill**: this operation stops the program(s) running in the container. This is typically done by sending a specific signal to the process running in the container.

— **Delete**: this operation deletes the resources allocated to the container and destroys the container. The unique ID no longer identifies a container, although the same ID might be used later to create a new container.

The container daemon typically also offers an event interface, which enable the container daemon to report on significant events relating to the containers which it manages. The event interface allows one or more client software components to listen for particular events and react to them.

### 7.2.3 Container resources, isolation and control

A container provides a software execution environment, which is isolated and resource controlled.

Isolation means that the software running inside the container is given the illusion that it has the system all to itself – that the only process(es) that exist are the ones started within the container. In reality, there may be many other processes running on the same system, but the software within the container is not aware of them and cannot see them or interact with them.

Resource controlled means that the set of resources available to the software in the container are allocated to the container (by the container daemon) when the container is created and these resources are monitored and limited. These resources include CPU allocation, runtime memory, networking, filesystem(s). It appears to the software in the container that only these resources exist. The resources are allocated in such a way that the resources allocated to one container cannot interfere with resources allocated to other containers running on the same system.

The isolation and control of resources are handled through capabilities of the host operating system, exploited and managed by the container daemon. The detailed capabilities available and used for containers vary from operating system to operating system. The capabilities used on the Linux operating system are described in this document for illustration. Consult the documentation relating to other operating systems to understand the equivalent capabilities.

For access to block I/O, the container by default has access to a filesystem consisting of the container image used to create the container in read-only mode, plus the addition of a read/write container layer. The default filesystem is transient and the container layer is deleted when the container is deleted. Additional, usually permanent, storage facilities can be made available to the software within the container through the configuration of the container by the container daemon, either as mounts to the filesystem within the container from some location outside the container, or via the provision of one or more specific storage services (which are typically cloud storage services). If there is a need for the software in the container to access storage objects that have a long lifetime, such additional storage facilities are necessary. In all cases, the apparent location of the files and storage objects within the container are mapped to actual locations outside the container.

Similarly, for access to networking capabilities, the resources available to the code running in the container are controlled by the container daemon and the configuration applied to the container, i.e. the networking capabilities are pluggable and configurable. It is possible to make no network capabilities available to the container (i.e. no exposed ports, no network devices available, so no routes to any target network endpoints). It is also possible to control access to the container and access from the container in detail.

Ports exposed by the container can be controlled and can be mapped between the network addresses and ports exposed by the container and those visible externally. For example, the container can expose port 80 for HTTP traffic, but this can be mapped to port 8080 for external (e.g. internet) access. In general, IP addresses, ports, hostnames, MAC addresses, routing services and DNS services can be controlled and mapped.

One significant form of networking that is used with containers is where the network exclusively connects a set of related containers, e.g. containers which represent a single application implemented using microservices architecture with different components of the application running in different containers. This is a form of virtual networking, where only designated containers can talk to each other (other than any specific externally exposed endpoints) as if they were the only entities on the network. The network is able to span across different systems and also across different underlying networks, permitting great flexibility in the location of each container, i.e. container location transparency is provided while still having the ability to control and limit communications for security purposes.

On Linux, control over resources is handled through a capability called control groups or cgroups. cgroups provides control over resources available to sets of processes, including CPU, memory, I/O to block devices (i.e. filesystems), access to devices, networking.

On Linux, isolation is implemented through namespaces. Effectively, any resources accessed by one container are part of one namespace, while the resources accessed by other containers are each allocated to other namespaces. The namespaces operate in such a way that software running in a process which is started under one namespace can only see resources within that namespace.

The following kinds of namespace exist in Linux (as from Linux kernel 4.10):

— Interprocess Communication (ipc): relates to interprocess communication. Only processes within the same namespace can establish communication (e.g. allocate shared memory).

— Mount (mnt): relates to mount points, i.e. places where (additional) filesystems are mounted. An initial set of mounts is available when the container is created by the container daemon, but after that, any new mounts are only visible within the container.

— Network (net): contains network related resources such as interfaces, IP addresses, routing table, socket listing, connection tracking table, etc.

— Process ID (pid): contains a set of process IDs – the first process in the namespace has id number 1 – and this process has special treatment equivalent to the init process on the underlying operating system.

— User ID (user): provides user IDs enabling both user identification and also privilege control – the user namespace maps user IDs within the namespace to user IDs in the underlying system – this allows close control of privileges and can provide for higher privileges within the namespace which are not provided for any resources outside that namespace.

— UTS: enables different processes to appear to have different *host* and *domain names*.

The combination of cgroups and namespaces together provides the resource control and isolation required for containers.

## 7.3 Container images and filesystem layering

### 7.3.1 Image purpose and content

A container image is an executable package that contains everything that is necessary to run software such as an application or a microservice. This can include the code of the application itself, a runtime, libraries, environment variables, configuration files and other metadata used by the application. The aim is that the container image is self-describing and encapsulated so that a container daemon can take the container image and create a container from it, without extra dependencies and regardless of the underlying system ("infrastructure-agnostic") and regardless of the contents of the container image ("content agnostic").

The container image contains sets of files which represent the code of the application, its dependencies and other files and metadata used by the application. The container image also contains structured metadata about the container image contents themselves and how to convert those contents into



 

a container. The container metadata can vary depending on the particular container image format concerned. The container metadata described in the OCI Image Format Specification[9] includes:

— Image index. "Top level" metadata which has the purpose of supporting container images which support multiple different platforms (this is sometimes called a fat manifest). Where multiple platforms are supported, each platform has its own specific image that contains the artefacts to use when running a container on that platform. Effectively, the image index references one or more image manifests.

— Image manifest. Contains information for a single container image for a specific CPU architecture and operating system, consisting of a configuration and a set of layers.

— Image layout. Specific layout of directories and files within the image with metadata about the filesystem layers.

— Filesystem layers. One or more serialized filesystems (i.e. structure of directories and files) and filesystem changes (removed or updated files). The layers are applied on top of each other to create a complete filesystem in the running container. (See 7.3.2 for a description of filesystem layering).

The functionality behind the image index or fat manifests allows for a single container image to provide support for platform specific images. The platform can include CPU type (machine architecture), operating system type and potentially operating system level. Thus, a single container image can be structured to enable the delivery and deployment of the same application to a number of different target systems.

One of the typical characteristics of the container metadata contained in container images is that it provides extensive security features aimed at ensuring that the content of the container image has not been tampered with since its creation. Data lengths are recorded, along with digests of the data (essentially a collision-resistant cryptographic hash of the bytes of the data). The data concerned can be the content of the filesystem layers, or elements of the metadata. The digest can also serve as a unique identifier for the content, which can also be used to support content-addressable access to the data. Separate secure communication of the digest to the user of the container image permits verification of the content of the container image even if it is retrieved from an untrusted source.

### 7.3.2 Filesystem layering

Container images, and the containers generated from them, make use of the technology of filesystem layering when dealing with the files they contain.

Filesystem layering is an approach to creating the content of the filesystem used by the container. The principle is that the filesystem content is built up as a stack of layers, each containing some set of directories and files, all having a common root directory. Directories and files are contributed from each layer in turn, starting with the base layer and proceeding upwards through the stack of layers. Each succeeding layer can contribute new files, but can also replace a file in a lower layer with a different version, or it can remove ("obliterate") a file present in a lower layer.

Filesystem layering allows for efficient handling of files in container images. Filesystem layering is also a practical approach to the creation of container images, given that typical applications have software stacks as dependencies which enable them to run.

Consider the example of a node.js application. The code of the node.js application might be contained in an app.js script file, plus other script files, data files and configuration files. The node.js application has a dependency on a node.js runtime plus some series of (external) packages. In turn, the node.js runtime depends on various operating system libraries.

In a container image for the node.js application, this could be rendered using 3 layers (starting with the topmost):

— the app.js script and associated artefacts constitute the topmost layer;

— the node.js runtime and associated packages form the next layer;

— the operating system libraries form the bottom layer.

Note that the operating system kernel does not need to be present in the container image. The operating system kernel is provided by the host operating system on which the containers run.

Layering reflects an efficient process for building (creating) container images. While it is possible to create a container image with a single layer containing all the necessary files, it can be much more efficient to separate the software stack used by an application into separate layers, since the application and each of its dependencies are typically separate independent sets of files, as described for the node.js application example.

One container image can be built using as the base (or "parent") another container image. Therefore, using the node.js application example again, the first container image built can be one for the operating system. Then a second container image can be built for the node.js runtime and its associated packages, using the operating system image as its parent. Finally, a third container image can be built for the app.js application using the node.js runtime as its parent. Each parent image provides the lower layers for the new image built on top. Therefore, in the example, the operating system libraries become the lowest layer, the node.js runtime the middle layer and the app.js application the topmost layer.

This enables each image to concern itself only with its own needs. For example, if the node.js runtime does not need all of the files from the operating system libraries, it can delete unneeded files. The main concern when building a container image thus becomes the question of which base image(s) to use.

Filesystem layering applies to containers as well, with a twist. When the container is instantiated from a container image, the same filesystem layers are built up as in the container image, but they are treated as read-only. These layers are called the image layers. The application running in the container cannot modify the files in the image layers. However, an additional writable layer is added on top of the layers present in the container image – this is called the container layer (called the sandbox layer in some container environments). All changes to files made by the application running in the container are written to the container layer, whether creating new files, modifying existing files or deleting files. This implies a copy-on-write strategy for files in the container.

A consequence of read-only image layers and the copy-on-write strategy is that the image layers can be shared between different containers, saving on storage and runtime memory and reducing the start-up time for containers.

### 7.3.3 Container image repositories and registries

The capability to store and to access container images is a key aspect of the container ecosystem. Individual container images are typically used in multiple different systems, for example to support scalability and to enable redundancy for improving availability. The recommended process for building images also places an emphasis on accessing existing images which form the parent image for the one being built.

Re-use is encouraged in the container ecosystem. The container images for common elements of the software stack(s) used by applications are used as the parent images for many applications. Typical examples are those for operating system libraries and those for middleware and runtimes. It is highly likely that the container images for these software packages will be (re)used over and over again in the container images for applications that use those software packages in their software stacks.

It is typically better and less work to reuse a container image created by someone with expertise in the software package concerned than to create a new image for that software package. In addition, such images are typically kept up to date with revisions to the underlying software.

Providing a capability to store and to access container images is the responsibility of a *container registry*. Container registries can be provided as public cloud services or can be provided as a private cloud service. An example of a public container registry service is Docker Hub[80]. Container registries have service interfaces which at least provide for push and pull operations. The push operation uploads one or more images to the registry while the pull operation downloads one or more images from the registry.



 

A repository is a collection of related container images. An example of a collection of related container images is a set of container images for operating system libraries for a specific operating system, where each image is for a specific version of that operating system.

For example, a container repository could contain a set of four container images with names *some_os_libs:16.01, some_os_libs:16.02, some_os_libs:16.03, some_os_libs:latest*. In this (simple) case, the repository has four entries for different versions of the operating system, each with a tag indicating the version number. The version tagged "latest" actually points to the same container image as the one tagged "16.03".

The reason for this arrangement is that when other container images want to always use the latest version of the operating system container image as a parent, they can use the "latest" tag when retrieving the image and this is automatically updated when new container images of later versions of the operating system are uploaded to the container registry. Other uses for the tags applied to image repositories are to make images designed for specific target environments, although fat manifest images are an alternative approach to achieve this capability.

## 7.4 Container management systems (CMSs)

### 7.4.1 General

As described in 7.2.2, it is typical of the container daemon and related tools to provide capabilities to manage the lifecycle of a single container. However, the deployment of typical cloud computing applications usually involves the deployment and operation of multiple containers often on multiple host systems. An application can involve multiple instances of a particular container running in parallel, both to provide redundancy against the failure of a single instance and also to provide scalability to handle the workload of incoming requests. An application can also involve multiple different components, with each component running in its own container instance(s), through the use of microservices architecture or the separation of capabilities in multiple tiers such as a web application using a database. A CMS orchestrates and manages defined sets of containers.

The CMS can abstract the underlying infrastructure, treating the set of containers as a single deployment target, while at the same time enforcing policies for deployment such as the separation of parallel container instances for redundancy and failover purposes.

Various CMSs are available and in common use, including Docker Swarm[79], Kubernetes[17], Apache Mesos[57], HashiCorp Nomad[58], and CloudFoundry (a PaaS system)[59].

### 7.4.2 Common CMS capabilities

The common capabilities of a CMS include:

a) Orchestration

CMSs provide for orchestration of container instances, including initial creation and placement, scheduling, monitoring, scaling, updating, and the parallel deployment of capabilities such as load balancers, firewalls, virtual networks and logging capabilities.

In essence, the CMS orchestration tools can abstract the underlying infrastructure, treating the set of containers as a single deployment target, while at the same time enforcing policies for deployment such as the separation of parallel container instances for redundancy and failover purposes.

Orchestration is the key component a CMS requires to support scale, since scale requires efficient automation.

b) Scheduling

The scheduler ensures that demands for resources placed upon the infrastructure can be met at all times. The scheduler selects the node based on its assessment of resource availability, and then tracks resource utilization to ensure the component does not exceed its allocation. It maintains

and tracks resource requirements, resource availability, and a variety of other user provided constraints and policy directives.

c) Monitoring and health checks

Automation is the essence of cloud computing systems, especially where fault-tolerance and rapid scalability are concerned. Such automation can only be provided for production systems by means of the CMS continuously monitoring the application's distributed set of containers and evaluating their health.

This capability enables faults and failures to be detected and actions taken to ensure that the desired configuration of the application is maintained, as defined in the declarative configuration. This can involve removing failed instances and starting new instances.

d) Autoscaling

Monitoring can support the dynamic scaling of the resources applied to the application to match the incoming workload, with the aim of keeping the resources deployed to the minimum necessary to service the load, since cloud computing is often charged on the basis of the resources used.

e) Resource management

In a CMS, a resource is a logical construct that the orchestrator can instantiate and manage, such as a service or an application deployment.

f) (Virtual) networking

A typical application consists of multiple separate components which act together to provide the functionality of the application. The separate components typically need to communicate with each other via networking, since the components often run in different locations. The CMS is responsible for setting up the necessary networking to enable the components to communicate. The networking is often virtual networking, removing the need for the components to understand the underlying network infrastructure and also improving security by restricting communications to those components belonging to the application.

g) Service discovery

Discovery is a key element associated with container deployments – applications consist of multiple containers and associated components running across potentially widely distributed infrastructure. As a result, individual components need to discover the other components which they depend on.

For example, a load balancer needs to identify all the component instances that it is using to distribute the incoming requests. It is typical of CMSs to provide capabilities that assist with discovery.

h) Updates and upgrades

CMSs typically manage the process of updates and upgrades to components of the application. Such changes can be the result of new functionality or fixes for the application code itself, or it may be the result of updates to the software stack used by the application code, such as runtimes. The CMS typically manages the upgrade to provide zero downtime, through a phased introduction of instances using the updated code and removal of instances using the older code.

i) Declarative configuration

It is common for CMSs to provide a means for the DevOps team to configure the orchestration for an application declaratively, using a defined schema written in a language such as YAML[81] and JSON[82]. Declarative configuration usually also contains essential information about container repositories, networking configuration, storage facilities and security capabilities that support the application. The declarative configuration is essential in enabling the CMSs to automate the process of managing the application and its components. In essence, the declarative configuration



 

indicates to the CMS the desired configuration and the CMS aims to both create and then maintain that configuration, deciding on the actions required to achieve this using knowledge of the target systems and internal deployment strategies.

# 8 Serverless computing

## 8.1 General

Serverless computing is a cloud service category in which the CSC can use different cloud capabilities types without the CSC having to provision, deploy and manage either hardware or software resources, other than providing CSC application code or providing CSC data. Serverless computing provides automatic scaling with dynamic elastic allocation of resources by the CSP, automatic distribution across multiple locations, and automatic maintenance and backup. Serverless computing capabilities are triggered by one or more CSC defined events and execute for a limited time period as required to deal with each event. Serverless capabilities can be invoked by direct invocation from web and mobile applications.

The underlying concept behind serverless computing started with functions as a service, where the idea is for the CSP to allocate appropriate runtime resources required for CSC application code dynamically on demand, without the CSC needing to preallocate and manage specific machines, VMs or containers and any associated stack of software. Other kinds of cloud service are also available which follow the serverless computing model, notably serverless databases.

Another way of describing serverless computing is that it is a form of platform cloud service category (or PaaS), since only the application code and/or data itself is supplied by the CSC, while all other resources and capabilities required to run the application are supplied and managed by the CSP.

It is typically the case that cloud services of serverless computing category scale automatically to deal with incoming requests. Cloud services of serverless computing category often have fault-tolerant capabilities, such as the ability to place the CSC application code in multiple locations with automatic fail-over when a fault occurs.

Serverless computing still needs servers in order to run, so in that sense, the name is a misnomer. What is not required is allocation and management of server resources by the CSC.

Serverless computing often has a charging model that charges for work executed by the cloud service in a granular fashion, rather than a charge for allocated resources (e.g. a VM or a container). Therefore, charging can be per API call, or per HTTP request, for example. This can be viewed as a more extreme form of "pay as you go" charging.

The benefits of serverless computing to the CSC can include:

— reduced operational costs, particularly lower costs associated with scaling to meet varying workload, since charging is directly related to the work executed by the cloud service, rather than costs being related to allocated resources. Also, required software stacks do not need handling as is typically the case when using VMs and containers.

— reduced development costs and reduced development time, since there is no need for developers to concern themselves with any aspect of resource management including deployment and scaling, e.g. application code can simply be uploaded and executed.

— lower packaging and deployment complexity. In particular, there is no need to consider anything except the CSC application code or CSC data. Any related software stacks are supplied by the CSP as part of the cloud service and are not packaged or deployed by the CSC.

## 8.2 Functions as a service

### 8.2.1 Overview

A common form of serverless computing is functions as a service (FaaS). FaaS is a form of serverless computing in which the capability used by the CSC is the execution of CSC application code, in the form of one or more functions that are each triggered by a CSC specified event. FaaS is also a form of Compute as a Service (CompaaS) as defined in ISO/IEC 17788.

FaaS can execute customer application code written in one or more programming languages. Each FaaS cloud service supports applications written in one or more programming languages including, but not limited to, C#, Go, Java™, JavaScript (node.js[78]), PHP, Python, Ruby, Swift. FaaS embodies the capability of platform as a service in providing the runtime software stack required by the application code, meaning that the CSC is not required to deploy and maintain the runtime software stack. It is typically the case that FaaS does not require the CSC application code to be written to use any specific application framework. In addition, FaaS takes on the responsibility of running the application and the runtime software stack on demand to service any events that trigger the application.

Effectively, FaaS aims to make infrastructure invisible to the developer of applications and services. When using FaaS, the underlying servers, virtual machines and/or containers are invisible to the user of the service. The developer not only does not access them, they cannot access them, since they are automatically managed by the CSP as part of the cloud service.

A significant aspect of FaaS is that resources are only consumed while a particular function is executing. It is typical that when an application uses a VM or a container in a compute cloud service, that each VM or container instance that is running consumes resources continuously, whether it is executing incoming requests or not. For FaaS, when there are no events being processed, no resources are consumed. As a result, this can mean less cost to the CSC, especially for less frequently used functions. The FaaS fires up the necessary resources (such as an underlying container) when an event trigger occurs for a given function.

The automated management provided by FaaS is key – scalability is automatically provided. If the rate of events for a given FaaS increases, then the resources allocated to that cloud service increase automatically to deal with those events, and are deallocated once the rate of events falls.

A number of serverless runtime cloud services are available, including Apache OpenWhisk[6], AWS Lamba[5], Azure Functions[60], Google App Engine[3], Google Cloud Functions[61], IBM Cloud Functions[62], Oracle Fn[63].

### 8.2.2 Functions within FaaS

Using FaaS means writing one or more functions, where each function is a piece of code dedicated to one specific task. It is this aspect of programming a serverless runtime that gives rise to the name for these cloud services: Functions as a service (FaaS). In effect, this is a major change in the way that applications and services are developed, embodying some of the principles of microservices architecture (see Clause 9). There are some principles that apply to functions and the way in which they execute.

Functions are stateless, which means that each function does not keep any state between successive invocations of the function. Effectively this means that functions don't store any data themselves. If a function needs to store and access data that has a lifetime longer than a single invocation of the function, then the function integrates with one or more cloud storage services (see Clause 12).

Each function is executed when triggered by some event, where an event notification results from some change of state, i.e. FaaS have an associated event-driven architecture and this involves asynchronous behaviour relating to the sending and receiving of events. This approach can enable much greater scalability and resilience for applications, especially when the applications are implemented on distributed systems.

Functions are time bounded in that they cannot execute for more than some specified time, as determined by the CSP. The time limit varies from one FaaS offering to another but it is often a small number of minutes. Thus, any long-lived tasks are not suitable for implementation as a function.

Related to the time bounding of functions is the question of function startup latency, i.e. the amount of time it takes for the FaaS to make available a running instance of a function when an event occurs. This can either involve a cold start, where a new instance has to be started from scratch, or a warm start where the FaaS is able to reuse an instance used to handle a previous event. A cold start involves much greater latency than a warm start. Cold starts are much more likely to occur for functions that are used infrequently since the FaaS typically deallocates an instance that has not been used for more than a given (usually short) amount of time.

Each function is made available via an API and can be called either by a client entirely outside the cloud system (e.g. an end-user application running on a client device) or by a client that is another part of the overall application. Each function can be considered as a microservice and in turn each function can depend on using other microservices to achieve its capabilities.

How events are described in data structures becomes a significant concern for functions within serverless runtimes. As a result, specifications have emerged to help describe events in a clear and consistent way, such as the CloudEvents specification of the Cloud Native Computing Foundation[49].

Since it is possible to deploy a single function at a time, there is considerable flexibility in the serverless approach. Applications can be built one function at a time, each deployed and scaled independently. This also implies an increase in the speed of development and deployment (there is no need to wait for the build of an application or of a service that contains multiple capabilities) each capability can be created, tested and deployed on its own.

### 8.2.3 Serverless frameworks

A serverless framework is a tool to assist in the creation and deployment of functions for FaaS cloud services, in particular supporting deployment of functions to different FaaS offerings of different CSPs.

It is commonly the case that FaaS services are proprietary to a CSP, although there are some open source FaaS implementations.

To deal with the problem of developing functions for deployment on any one of a variety of CSP FaaS offerings, serverless frameworks have been developed which enable the development of functions that can be targeted to different FaaS offerings on demand, taking care of the differences between the offerings, particularly in respect of upload and deployment processes.

Examples of open source serverless runtimes are Apache OpenWhisk[6] and Oracle Fn[63]. An example of a serverless framework is the open source Serverless Framework[50].

### 8.2.4 FaaS relationship to microservices and containers

Using FaaS essentially involves utilising a cloud microservices architecture for applications. FaaS implies using a "cloud first" approach to applications, very different in style from "monolithic" applications which embody all functions in a single package.

Thus, using FaaS and functions is one way to implement a microservices-based application architecture, but one that does not require the use of containers and the associated CMSs like Kubernetes.

However, it is possible to mix the use of FaaS with microservices implemented using containers (or using VMs), with functions invoking container-based microservices and container-based microservices invoking FaaS functions as desired.

### 8.3 Serverless databases

Serverless database is a form of serverless computing in which the capability used by the CSC is a database, where the database is provisioned, managed and operated by the CSP and its functions are made available via an API.

With respect to serverless computing, the allocation of storage resources is managed by the CSP. The amount of storage is automatically and dynamically scaled to match the amount of CSC data that is placed into the database. Replication and backup is managed by the CSP, and this includes placing the data in locations suitable for the use that is being made of the data and also keeping the multiple replicas in step with each other. Equally, the processing resources needed to service queries and updates to the database are also managed and scaled by the CSP.

Examples of serverless databases include Amazon Aurora Serverless[64], FaunaDB[65], Google Cloud Firestore[66], IBM Cloudant[67], Microsoft Azure Data Lake[68], Oracle Autonomous Database[83], Oracle NoSQL Database[84].

## 9 Microservices architecture

### 9.1 General

Microservices architecture is a design approach for building a cloud native application. A cloud native application is an application which is explicitly designed to run within and to take advantage of the capabilities and environment of cloud services. Microservices architecture is an architectural style that involves breaking up an application into independently deployable microservices that can be rapidly deployed to any infrastructure resource as required[10][11]. In the microservices architecture, the application is divided into a series of separate processes called *microservices*, deployed independently and connected to each other via service interfaces. The concept is that the microservices within the application are designed to implement some specific area of function, perhaps a particular business process or a specific technical capability. The architecture makes it possible to operate and to update each microservice independently. Microservices architecture is thus the umbrella technique in which microservices are the major components.

NOTE The terms service and service interface are used here with the definitions given by ISO/IEC 18384-1 [69], which also provides a good explanation of service-oriented architecture of which microservices architecture is a particular example. A service or a microservice should be distinguished from a cloud service. It can be the case that a microservice is implemented as a cloud service, but this is at the choice of the application developer and there is no requirement to do so.

A simple example of a microservices-based application is given in Figure 4. In this example, the application has a core plus two microservices, one handling the business capability of handling user accounts with the second handling displaying and controlling video sequences. The example goes further and shows that microservices applications can also make use of other services, typically cloud services providing capabilities needed by the application. Therefore, in this example, the user account microservice uses a database service to store and retrieve account information; the video display microservice uses a video storage service which stores the video sequences; the core application uses an email service, a twitter feed service and an analytics service.

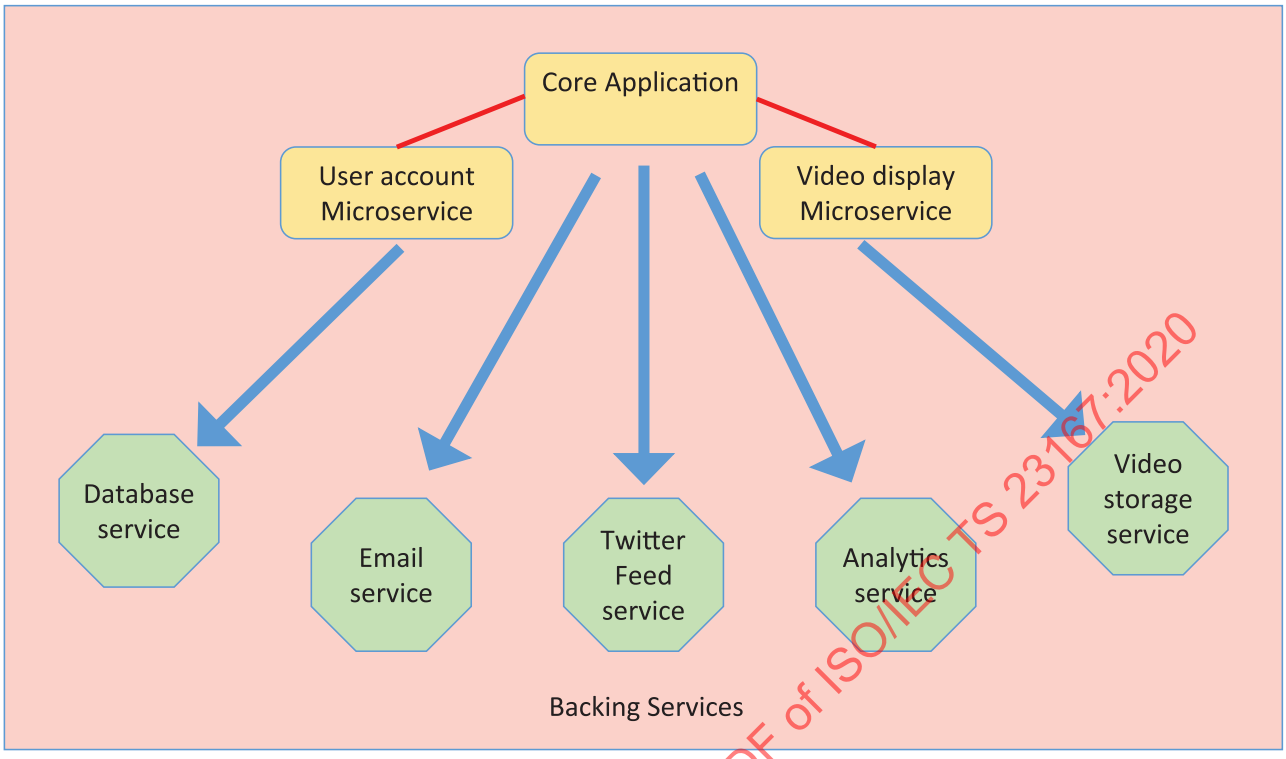


**Figure 4 — Example of application structured using microservices architecture**

It is important to understand that microservices architecture is a technique and that microservices are the primary part of microservices architecture. This is separate from the technologies that might be used to implement microservices. Microservices might be implemented using containers, or VMs, or using serverless computing and connected using some virtualised networking, but this is separate from the technique used to build the applications.

Functional decomposition often in a domain-driven design context is the key to building a successful microservice architecture. One viewpoint regarding this architecture is that it is a refinement and simplification of service-oriented architecture (SOA). Some of the characteristics of microservices architecture are as follows[12]:

— Each architectural component called a “service” has a well-defined and explicitly published interface.

— Each service is fully autonomous.

— Changing a service implementation has no impact on other services as communication between services takes place using interfaces only (usually a REST interface).

— The loose coupling and high cohesion between services enables composing multiple services to define higher level services or applications.

Microservices based applications are contrasted with "monolithic applications", where all the components of the application are built and bundled together in a single process, which is more typical of older, non-cloud, enterprise applications.

## 9.2 Advantages and challenges of microservices

The benefits[13][14] of a microservice architecture are:

— Simpler codebases for individual services.

— Ability to update and scale each service in isolation.

— Enable services to be written in different languages to meet the performance needs and ease of development. This is termed "polyglot programming".

— Use of varied middleware stacks and even varied data tiers for different services (flexibility).

One of the advantages of using a microservices architecture is that each component of the application built as a microservice can be scaled separately to match the load on that component alone. This differs from a monolithic application approach. In a monolithic application architecture, all components are deployed and operated as a single entity, with scaling only possible by scaling up/down the whole application. This can lead to resource inefficiency for those components of the application that are not under heavy load.

PaaS systems make it straightforward to deploy each microservice independently and link them together to create the full application. Each microservice can be managed independently: scaled, distributed, updated.

Another advantage of using a microservices architecture is that each component of the application built as a microservice can have a separate development lifecycle. This allows for smaller application components that can be modified, extended, tested and deployed more rapidly.

The increased benefits come with challenges that need to be addressed[15] to realize those benefits. A brief discussion of these challenges is given below:

— Communication optimization: Running an application in different processes results in increased communication overhead due to API calls between services as compared to function calls within a process. The overall strategy involves identifying the right protocol, response time expectations, timeouts and API design, with artifacts such as API Gateway (9.7), Circuit Breakers (9.6), Load balancers (14.2) and Proxies (14.2).

— Service discovery: This refers to the capability for services to discover each other in a consistent manner. It is necessary to have a standard and consistent process for services to register and announce themselves. The consuming services should be able to discover the end points and locations of other services. A specification of how API gateways are configured to report service availability and enable discovery is necessary.

— Performance: Trying to fulfil one single business functional requirement can result in orchestrating multiple service calls together. This can introduce additional lag in response time. Further, data that is frequently used by a single microservice can be owned by another microservice. This requires data sharing and synchronization capabilities to avoid communications overhead caused by data copying during service invocations.

— Fault tolerance: This is the ability of the system to recover from a partial failure. Microservice developers need to provide mechanisms to gracefully recover or stop any failure from propagating to other parts of the system. Further, some services are run in multiple copies for scalability and availability reasons. The number of copies, the version consistency among the copies, load balancing mechanism and the network locations are key decision factors for ensuring fault tolerance.

— Security: A critical decision is deciding on the trust relationship between the microservices based on the various ways services communicate with each other. When invoking another service, a service can use either a synchronous or asynchronous protocol. All these factors are to be taken into consideration when assigning chains of authorization within access tokens. The communication patterns among services should have specific and efficient authentication and authorization mechanisms based on risk-based security policies. Increased communication between components (as described under Communication Optimization above) calls for secure communication protocols that meet the requirements for the application.

— Tracing and logging: The process of decomposing monolithic applications into various microservices creates a need for additional techniques and solutions in relation to debugging and profiling systems. One feature that is needed is called *distributed tracing* which calls for the capability to track a chain of service calls to identify a single business transaction or a single user request. A central logging



 

system is typically required to obtain a wholistic view of system behaviour and this calls for an aggregation capability to integrate log information from individual microservices.

— Deployment: The proliferation of service processes requires automated mechanisms for deployment. Scalability and system integrity are primary concerns in the deployment of microservices. Containers are the predominant mechanism used for deploying microservices and the use of CMSs (see 7.4) (which assign resources and implement the connection topology) addresses deployment concerns. However, some of the assumptions and requirements in deployment models may not fit well with the functional requirements of certain microservices based applications. An example is the assumption of statelessness of a container hosting a microservice, where the overall system/application requirement calls for a stateful microservice.

— Functional decomposition: While decomposing a monolithic application, there are issues to decide such as:

  a) the proper boundaries of different services, and

  b) when a service is too big and hence needs to be broken up.

## 9.3 Specification of microservices

The design of a microservices architecture calls for use of description diagrams and platform-neutral description languages because of the heterogeneity involved in design of the component microservices. While UML is predominantly used for description diagrams, the following languages are generally used:

— Standard Modeling Languages, such as RAML and YAML.

— Standard Specification Languages, such as Javascript (Node.js), JSON and Ruby.

— Pseudocode for algorithms.

— Implementation-neutral interface specification language, such as the Open API specification[70].

## 9.4 Multi-layered architecture

Domain driven design[54] and an associated multi-layered architecture[52] is a common pattern used in software engineering. By dividing applications functionally into distinct layers, multi-layered architecture provides the following advantages:

— Efficient collaboration

  Each layer is developed by each layer's specialist: Web browser-based GUI is developed by Web designers and domain logic by Java™ programmers, for example. Specialists can concentrate on their own concerns with little interference.

— Easy maintenance

  Each layer's program code is logically independent of other layers' program code. As long as programmers don't break interfaces to other layers, it is flexible to change program code.

— Reusability

  One application is divided into smaller components in multi-layered architecture. Fine-grained software components can be reused more easily than coarse-grained ones.

The use of multi-layered architecture is effective in microservices based applications. It has been applied to applications developed using microservices and some practices relating to the use of multi-layered architecture are available in the published literature (e.g. See Toby Clemson[27]). Although there is no multi-layered architecture standardised for microservices, the layered architecture proposed in

Domain-Driven Design[52] has been referenced and the essence can be provided by means of four layers of components as shown in Figure 5:

— User interface

The software component defines accepts requests from users and provides responses.

— Application

The software component defines an application's boundary. It is the endpoint for interaction with clients and responsible for mediating requests and responses, invoking domain logic, and managing transaction contexts.

— Domain

The software component implements business logic.

— Infrastructure

The software component encapsulates physical resources including data and provides the domain layer with an abstract interface for data access.

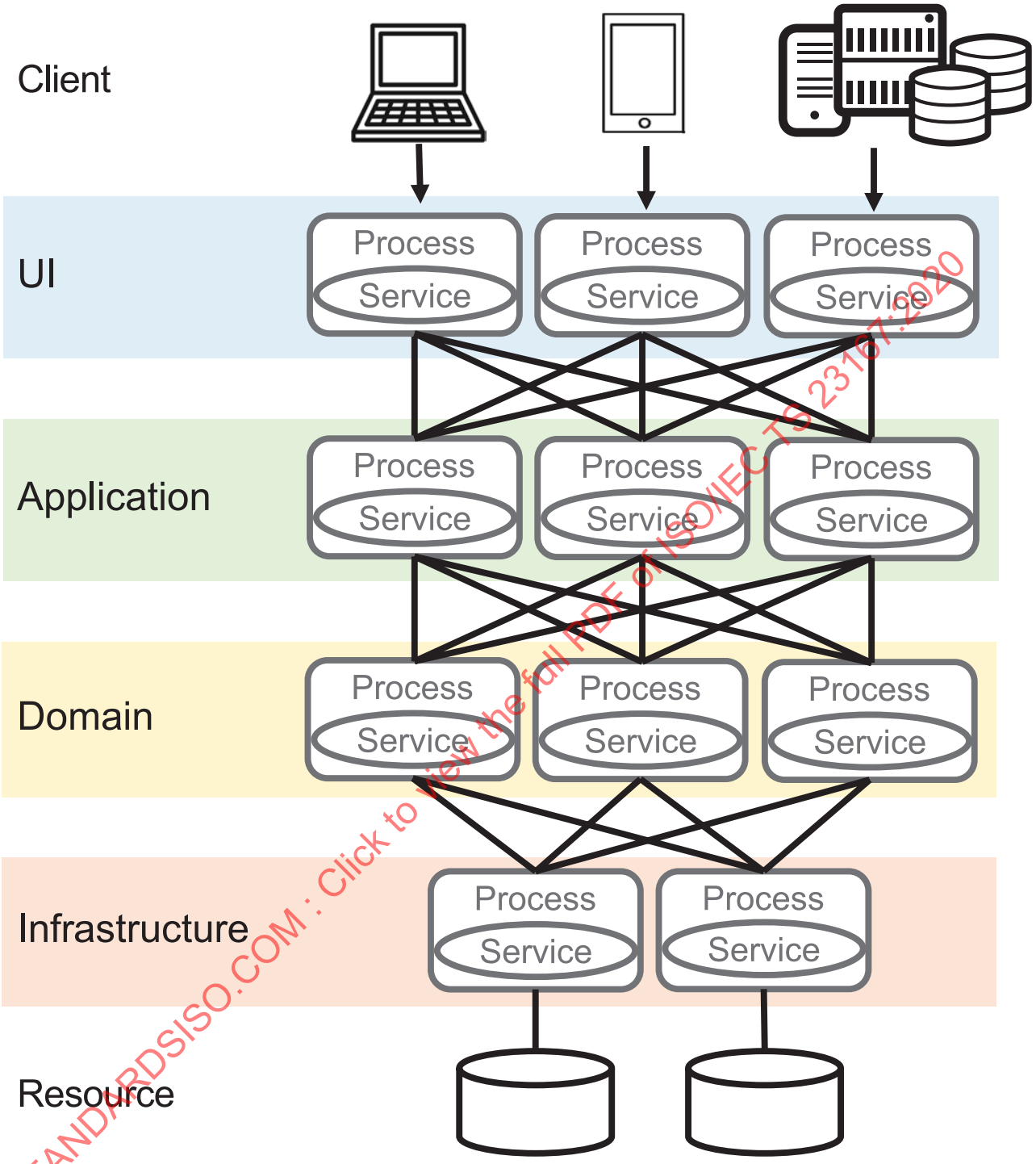


**Figure 5 — Multi-layer architecture used with microservices**

Each component shown in Figure 5 is a separate microservice, running in its own process and invoking other microservices as necessary.

Monolithic web applications have previously been developed based on a multi-layered architecture known as the model-view-controller pattern as shown in Figure 6. However, there are some differences in the implementation of the multi-layered architecture between an application designed using microservices and an application with a monolithic design, associated with software component packaging and application runtimes.

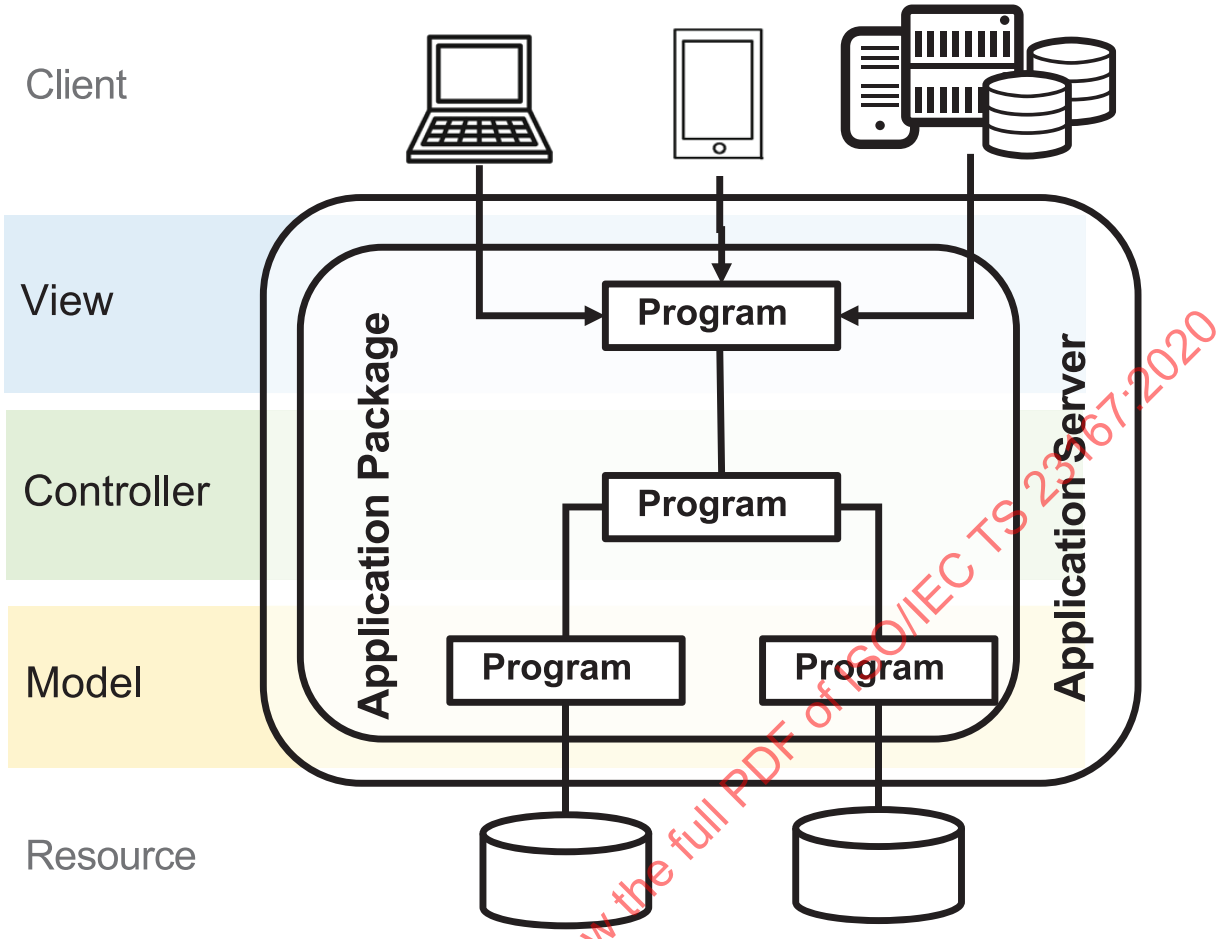


**Figure 6 — Monolithic web application pattern**

In monolithic application design, although an application is designed and implemented based on a multi-layered architecture, all software components are assembled in one software package and deployed in one application runtime. Even if a web designer adds a trivial update to the GUI, the whole software package has to be built, tested and the server runtime has to be stopped to deploy the new software package. This can be a lengthy process, even for a small change to the application.

On the other hand, in a microservices application design, each software component in each layer is packaged as a distinct microservice and independently deployed to a distinct process. Each process can be implemented using a virtual machine or using a container or as serverless functions, each of which can be started and stopped separately. If each microservice is designed well in a loosely-coupled fashion, a developer can update one component without having to build, test or redeploy other microservices. Microservices architecture enables easy and flexible change of the application.

## 9.5 Service mesh

In microservices architecture, the number of microservices associated with an application can become large. It is common in this case for each service to run with multiple instances with a cluster configuration, each instance with a distinct process implemented as a virtual machine or a container. The number of processes could become many times the number of services. This makes the overall topology a complex network called a *service mesh* as shown in Figure 7.

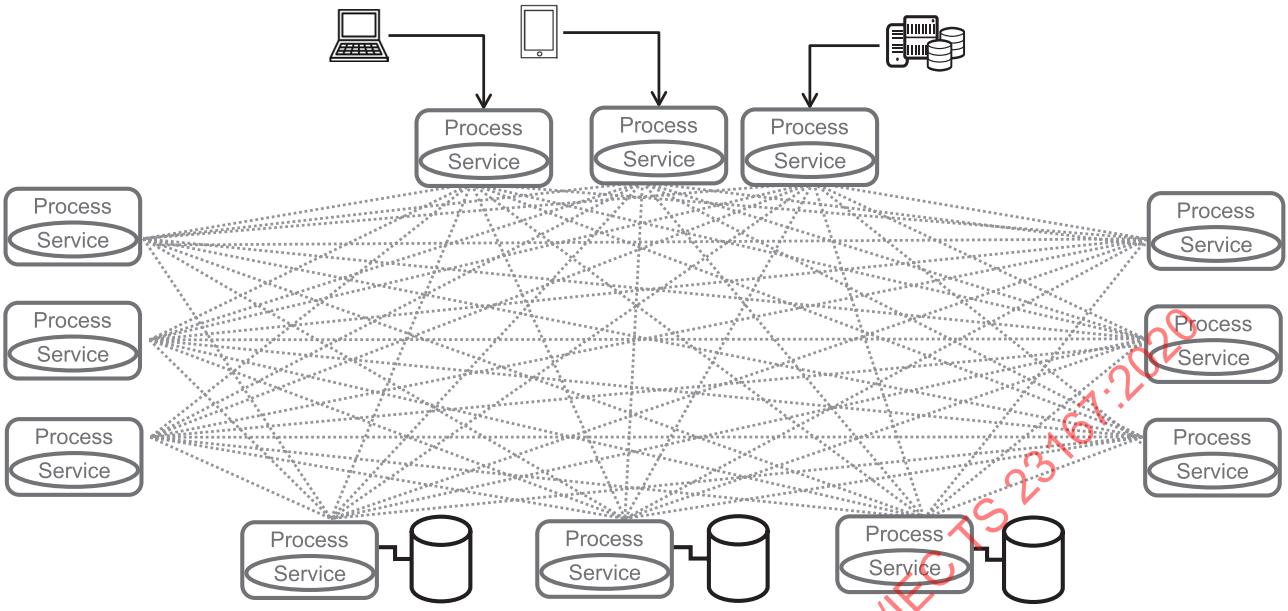


**Figure 7 — Service mesh for a microservices based application**

In order to run a microservices-based application and reap its benefits, it is necessary to deal with the challenges presented by the service mesh:

a) Traffic management
   - Fine-grained load balancing for a specific version of microservice.
   - Blue/Green deployment, in order to update a microservice without stopping the application.
   - Canary release.
   - Circuit breaker. (See 9.7)

b) Service discovery
   - Service registration.
   - Service lookup.

c) Test
   - Fault injection.

d) Security
   - Authentication.
   - Authorization.
   - Encryption.

e) Telemetry
   - Log and Trace integration.
   - Metrics integration.
   - Dashboard.

There are alternative approaches to manage the service mesh: a) API and b) service mesh fabric. For the API approach, application developers make use of a specific API in their programs in order to manage the service mesh. However, to do this developers have to take the effort to implement non-functional requirements as well as functional requirements and as a result, the application code involves non-functional implementation details, which is against the "separation of concerns" principle of software engineering and makes the code more complex and harder to modify. MicroProfile is an example of a service mesh API[28].

Service mesh fabric is an application infrastructure solution, located under the application layer and over the orchestration layer, and mediates all traffic between microservices. It manages the service mesh by manipulating traffic coming and going through itself. Then, the application program is freed from the implementation of capabilities required to manage the service mesh. Istio[29] and Linkerd[30] are examples of service mesh fabric implementations.

### 9.6 Circuit breaker

The circuit breaker is a design pattern, and also a software component based on that pattern[55].

Circuit breaker applies where one software component invokes another software component (such as a microservice) through an API call. The software components involved are running in different processes and the API call typically takes place over the network. Such remote API calls can fail or hang without a response. Where the target component is a commonly used service, this can lead to a cascade of failures across the application or system.

The idea of the circuit breaker is that any such remote API call is wrapped by a circuit breaker component which is effectively a part of the client software component. When the client invokes the API call, it is handled by the circuit breaker component which monitors for failures. When a failure state is recognised, calls made to the API are given a rapid error response by the circuit breaker. The circuit breaker can also generate alerts for monitoring purposes under these circumstances. The circuit breaker can continue to monitor the API and the target component for availability and reset itself automatically once the problem clears.

The recognition of the failure state can vary from one circuit breaker to another, and the circuit breaker can have settable parameters to control its behaviour (e.g. an error threshold, a timeout threshold).

The circuit breaker does not remove the need for the client component to deal with the failure of the API call, but it does make it easier to develop appropriate handling mechanisms.

### 9.7 API gateway

An API gateway is a software component that can be used to provide a single integrated API to a set of microservices that are being used together by a particular client component. (See Microsoft, 2019[56])

Each microservice presents its own API, based on its capabilities. A particular client can be using a whole series of microservices to achieve its goals. It can get complex for the client software to deal with all the different API calls which need to be made to the different microservices involved. An API gateway can present a simpler coherent API to the client software and invoke the microservice APIs as needed in order to implement the simpler API. The API gateway is thus a client-focussed component and multiple different API gateways can be required to satisfy the requirements of different clients.

## 10 Automation

### 10.1 General

Automation is a key feature of both the provision and the use of cloud services. Automation is applied to the complete lifecycle, through design, development, test, deployment, production and decommissioning. Automation is essential to achieve productivity and also to reduce the skills and effort required to provide and use cloud services.



 

One of the goals of automation is to reduce the effort and the burden to deploy applications and data into cloud services, recognising that this is done on a relatively frequent basis, either to fix problems or to provide enhancements to functionality. Automation is necessarily connected to the adoption of a series of software engineering techniques, which while not specific to cloud computing have become vital elements in the successful adoption of cloud computing.

## 10.2 Automation of the development lifecycle

One of the significant elements of automation is the adoption of either continuous deployment or continuous delivery. Continuous deployment is a software engineering approach in which teams produce software in short cycles such that the software can be released to production at any time and where deployment to production is itself automated. Continuous delivery is similar to continuous deployment, except that the deployment stage is initiated manually (i.e. the decision to deploy is made by a human rather than some automated system – the deployment process itself is usually automated). Generally, the use of continuous deployment or continuous delivery is also associated with the adoption of DevOps by the organization. DevOps involves a methodology which combines together software development and IT operations in order to shorten the development lifecycle, enabling frequent delivery of fixes and enhanced functionality closely aligned with business objectives.

Continuous deployment and continuous delivery place an emphasis on developing software in small increments, with a strong emphasis on automated testing during and after build and deployment steps. Small increments are closely allied to the development of applications using microservices (each microservice providing some part of the overall functionality) and the use of separate (cloud) services for more common functionality (e.g. database capability, messaging capability).

Continuous integration is an inherent part of continuous deployment and continuous delivery, where developer updates to the codebase are made frequently and the codebase is built and tested regularly (often many times a day). Continuous integration is built on a base of test-driven development, with the aim of automatically running unit tests and integration tests to check that updates to the codebase have not broken the code in any way and to give rapid feedback to the developers in the cases where there are problems.

Automated management is a key element of operations for cloud services. Tasks such as recovery of failed software instances, scaling up and down of resources, especially of parallel instances of application components, data replication and data back-up. All of these need to be automated when using cloud services or else it is possible for these tasks to overwhelm operations staff.

An important extension of the DevOps approach is termed DevSecOps. For DevSecOps, security capabilities are considered as an essential and integral part of the development and operations processes. The idea is to automate security tasks in parallel with the automation of development and operations tasks that is central to DevOps. The increase in the pace of development and operations tasks brought about by DevOps methodology is matched in DevSecOps by an increase in the pace of security related tasks, throughout the lifecycle of an application.

## 10.3 Tooling for automation

Tools are an essential part of all stages of the development process.

Typically, tooling starts with a source control management (SCM) system, which holds the source code and provides for controlled processes for performing updates on the code, including tracking all changes. The SCM system forms the base on which the tools for build, test, delivery and deployment operate. There are various SCM systems in use, however, the open source Git SCM[32] is very widely used, with a lot of associated tools, including host server capabilities.

An automation server is a tool used to automate the steps of continuous integration, continuous delivery and continuous deployment. It is particularly useful for performing builds of the code from the SCM and performing testing (unit tests, integration tests) on the built code. There are a number of automation server tools available and in use, although the open source Jenkins tool[32] is commonly used.

Security automation in support of DevSecOps can include tools that check code for vulnerabilities at the point where the code is checked in to the SCM, and check for vulnerabilities via testing during the build and during the continuous integration phase. This should also tie to the secure use of codebases which satisfy dependencies of the application, e.g. middleware libraries, container images and backing services. Such dependencies should be tied to security policies that determine which dependencies are suitable for use, backed by appropriate testing and a management system that responds to notification of vulnerabilities and the need to change to a later fixed version.

Configuration management software is used to automate software provisioning, configuration management and application deployment. The architecture used for cloud native applications increases the need for configuration management software, since there are typically multiple components installed in a variety of cloud services and associated locations, all of which have to be orchestrated to enable the correct operation of the application. A range of configuration management software tools are in use. Some of the commonly used open source tools include Ansible[34], CFEngine[35], Chef[36] and Puppet[37].

The configuration management software tools vary in their architecture. Ansible uses an agentless architecture, whereas the other tools are agent based (i.e. they require a software daemon installed on the target nodes or on an associated server).

A key element of the deployment of applications in a cloud environment is orchestration, since it is common for applications to consist of a significant number of separate components that must be deployed, configured and operated together. Automation of orchestration is the province of tools, such as the CMSs as described in 7.4.

A key element supporting automation is the provision of the capabilities of cloud services through APIs (application programming interfaces). APIs enable the various tools to configure, deploy, control and monitor each cloud service. This extends to the use of other tools via APIs, which includes tools such as Kubernetes for the deployment of containers.

Applications that are deployed and running in production within cloud services should be monitored and should be managed for performance and availability. Monitoring and management is typically done via APIs offered by the CSP. Tools to manage the restart of failed instances, tools to scale up and scale down the number of instances of a particular software component in response to workload changes all depend on such monitoring and management capabilities. It is the case that some of these capabilities are themselves supplied as cloud services ("auto scaling", for example), but in other cases they are supplied as separate tools that must be installed and configured.



# 11 Architecture of PaaS systems

## 11.1 General

Platform as a Service (PaaS) is a category of cloud services that involves the provision of platform capabilities, which is defined in ISO/IEC 17788 as capabilities *in which the cloud service customer can deploy, manage and run customer-created or customer-acquired applications using one or more programming languages and one or more execution environments supported by the cloud service provider.* A PaaS system generally involves a coherent set of PaaS cloud services intended to work together.

PaaS systems are primarily concerned with developing, deploying and operating customer applications. Other capabilities are often involved, such as the use of application, processing, storage and network resources, but they are not the main focus. PaaS systems typically involve diverse application software infrastructure (middleware) capabilities including application platforms, integration platforms, business analytics platforms, event-streaming services and mobile back-end services, plus sets of tooling that support the development process (See Gartner, 2014 and Gartner, 2018). In addition, a PaaS offering often includes a set of operational capabilities such as monitoring, management, deployment and related capabilities.

PaaS systems are targeted at application developers and also at operations staff, also supporting the combined concept of DevOps.

One way of describing PaaS systems is that they represent a cloud service rendering of the application infrastructure offered by entities such as application servers, database management systems, integration brokers, business process management systems, rules engines and complex event processing systems. Such application infrastructure assists the application developer in writing business applications, reducing the amount of code that needs to be written at the same time as expanding the functional capabilities of the applications. The essence of a PaaS system is that the cloud service provider takes responsibility for the installation, configuration and operation of the application execution environment (including any underlying VMs, operating systems, containers, runtimes, libraries), leaving only the application code itself for the cloud service customers and their developers to provide. Thus, the essential difference between an IaaS and a PaaS is that for IaaS, the customer has to construct a VM image or container image to execute their application code, while a PaaS provides everything needed to upload and execute application code directly.

PaaS offerings also often expand on the platform capabilities of middleware by offering application developers a diverse and growing set of services and APIs that provide specific functionality in a managed, continuously available fashion. This approach aims to obscure the fact that there is middleware present at all, enabling immediate productivity for developers. Some PaaS systems also blend in features of IaaS and SaaS cloud services, offering some control of basic resource allocation on the one hand and providing complete off-the-shelf software capabilities on the other.

In addition, PaaS systems typically provide their capabilities in a way that enables the applications developed on them to take advantage of the native characteristics of cloud services, often without the application developer having to add special code to the application itself. This provides an approach to building *cloud native applications* without requiring specialized skills.

## 11.2 Characteristics of PaaS systems

PaaS systems typically express a set of major characteristics:

1. *Support for custom applications*:

   Support for the development, deployment and operation of custom applications. PaaS systems typically support cloud native applications that are able to take full advantage of the scalable, elastic and distributed capabilities of cloud infrastructure. This is often achieved without the application developer writing special code to take advantage of these capabilities.

2. *Provision of runtime environments*:

   PaaS systems generally offer runtime environments for applications, where each runtime environment supports either one or a small set of programming languages and frameworks, e.g. Node.js, Ruby and PHP runtimes. A characteristic of many PaaS offerings is support for a range of runtime environments. This enables developers to choose the most appropriate technology for the task in hand, sometimes termed a *polyglot* environment.

   Runtime environments can include the use of containers (see Clause 7) and serverless computing (see Clause 8).

3. *Rapid deployment mechanisms*:

   Many PaaS offerings provide developers and operators with an automated "push and run" mechanism for deploying and running applications, providing dynamic allocation of resources when the application code is passed to the PaaS cloud service via an API. Configuration requirements are kept to a minimum by default, although there is the capability to control the configuration if required, e.g. controlling the number of parallel running instances of an application in order to handle the anticipated workload or to meet resiliency goals.

4. *Support for a range of middleware capabilities*:

   Applications have a variety of requirements and this is reflected in the provision of a broad range of application infrastructure ("middleware") that supports a range of capabilities. One

example is database management, with both SQL and NoSQL database technologies provided. Other capabilities include integration services, business process management, business analytics services, rules engines, event processing services and mobile back-end services.

5. *Provision of services*:

   PaaS systems often supply some capabilities as a series of separate services, typically invoked via an API of some kind. Services are installed and run by the provider of the service, removing responsibility and effort from the cloud service customer. For example, in the case of a database service, the responsibility for ensuring availability and reliability, for having replicas and backups of the database data, for securing the data and so on all falls on the service provider. Provider services are an essential concept in reducing the effort and complexity in building software systems, rather than having to install and manage some potentially complex set of software, the capability is obtained off-the-shelf from the provider.

6. *Preconfigured capabilities*:

   Many PaaS systems are characterized by capabilities that are preconfigured by the provider, with a minimum of configuration available to developers and customer operations staff. This reduces complexity, increases productivity and lowers the potential for unexpected problems, with capabilities simpler to manage and easier to debug. Some offerings can auto-tune such configuration based on usage patterns and loads; this further reduces the expertise and time required to run the applications in the most efficient manner.

7. *API management capabilities*:

   Business applications often need to expose some capabilities via APIs. This can be required by the nature of the user interface to the application. Mobile apps usually need an API so that while operating independently of the business application, they can access data and transactions when required. In other cases, part of the enterprise solution is to enable other parties (partners, customers, suppliers) to integrate their own applications with those of the enterprise. Such integration is done via an API. Providing an API requires a level of control, so that only authorized users can access the API and each user can only access those capabilities for which they have permission. This requires some API management capabilities and API management capabilities are offered by many PaaS systems.

8. *Security capabilities*:

   Security is one of the most important aspects of any solution. PaaS systems usually provide built-in security capabilities, thus reducing the load on developers and operators. Capabilities include firewall, endpoint management, secure protocol handling, access and authorization, encryption of data in motion and at rest, integrity checking, plus resilience mechanisms such as redundant copies of data and automated backups. PaaS systems can offer these capabilities with minimal or no impact on application code, simplifying the developer's tasks. Also, because the underlying execution environment is part of the platform, the CSP assumes responsibility for operating system security patches, malware detection and removal, and other essential security maintenance tasks, thus freeing the customer to focus on avoiding security vulnerabilities introduced in their own code.

9. *Developer tools*:

   Many PaaS systems aim to unify and streamline development and operations, i.e. support *DevOps* by breaking down the divisions between development and operations. Development tools provided include code editors, code repositories, build tools, deployment tools, test tools and services and security tools. It is common to find application monitoring and analytics services, including capabilities such as logging, log analysis and app usage analytics and dashboards.

10. *Operations capabilities*:

    PaaS systems assist operators through operations capabilities both for deployed applications and for the PaaS system itself, via dashboards and also via APIs that enable customers to plug in their own operations toolsets. For example, capabilities to increase or decrease the number of



 

running instances of an application are common (to deal with varying application load), in some cases handled by automated services that vary the number of instances based on a set of rules established by the CSC operations staff.

11. *Support for porting existing applications*:

    Many cloud service customers have existing applications that can be ported to the PaaS environment with resulting business benefits. Some PaaS systems have application environments that aim to closely match those available on existing non-cloud middleware stacks and associated tools that assist with the porting process.

12. *Support for applications using microservices architecture*:

    PaaS systems typically offer a wide range of support for applications built using microservices architecture (see Clause 9). This includes support for the runtimes used for the microservices themselves, support for the underlying services used by the microservices and support for the service mesh that links all the components together.

13. *Networking capabilities*:

    Since the CSP controls and has full visibility of the network protocol stacks in use, PaaS systems can be more deeply integrated with the network capabilities of the host CSP. This makes it relatively easy (for both the CSP and CSC developer) to integrate the PaaS with network virtualization, network load balancing, failover, network optimisation, caching, message passing and queuing, and other network related technologies.



## 11.3 Architecture of components running under PaaS system

Putting together the several elements of a typical PaaS system leads to a schematic architecture of the components of a typical application created and deployed using a PaaS system, shown in Figure 8.

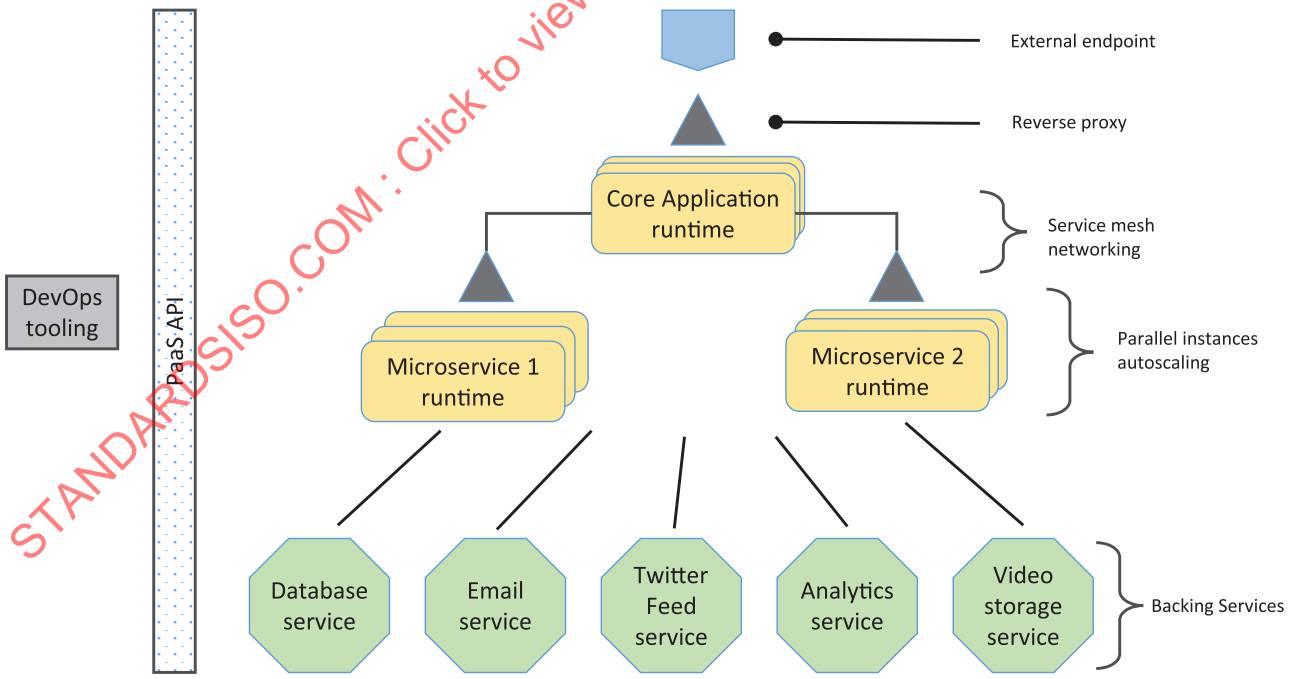


**Figure 8 — Schematic architecture of components running under PaaS system**

— External endpoint: provides an externally-available endpoint (e.g. visible across the internet), with associated endpoint security (e.g. https support, certificate management, DDoS attack handling, ID and Access management).

— Reverse proxy: for each component of the application which is scaled through the use of parallel instances (core application and microservices), there is a need for a reverse proxy and load balancing function to distribute incoming requests evenly across all the running instances.

— Service mesh: for the internal connections between application components and for connections with services, capabilities to enable effective and efficient connectivity.

— Autoscaling of parallel instances: the typical approach to the scaling of application components is to run multiple instances of each component in parallel and to distribute incoming requests over these instances (see Clause 14). The number of instances running at any one time can be scaled up and down to match the amount of work demanded by the requests. Typically this requires that the PaaS system monitors the instances to determine how busy they are. This capability can sometimes be linked with PaaS automatic network loading balancing, so that traffic levels to specific instances can be dynamically matched to their current capacity and availability.

— Backing services: it is commonly the case that many capabilities required by the application components are provided by a set of cloud services, to which the components are connected as necessary. Such services can be highly diverse, but examples include capabilities such as databases or other storage services (see Clause 12).

— PaaS API: the capabilities of the PaaS system and the individual cloud services that make up the system are made available for various DevOps tools to use by means of one or more PaaS APIs. For example, such an API can enable the code of an application component to be pushed to a runtime service for execution.

— DevOps tooling: developers and operations staff, ideally united into a seamless DevOps team, use a variety of DevOps tools to perform their work. Development and test tools are used during the creation and testing of an application and its microservices, while monitoring and management tools are used to observe and control the application components in production.

# 12 Data storage as a service

## 12.1 General

Cloud computing is based on the provision of cloud services, which are all ultimately based on the three infrastructure resource types of compute, storage and networking. This clause considers cloud services which offer storage resources.

The technologies of virtual machines and of containers described in Clause 6 and Clause 7 are fundamentally offering types of compute resource, i.e. a means of executing software in a virtualized environment of some type. Some storage capabilities are associated with both virtual machines and containers. There are filesystems associated with them which contain the files representing the software and the directly associated configuration and metadata, and these filesystems are also typically required to support the execution of the software.

However, it is necessary to understand that the filesystems associated with both virtual machines and with containers are essentially ephemeral in that they are brought into existence when the virtual machine or container is created and they are discarded when the same virtual machine or container is stopped and destroyed. This means that such filesystems are not capable of being used for long-term storage of information. Neither can they be used for information that needs to be accessible by multiple different virtual machines or containers, since the filesystems within a virtual machine or within a container are by design isolated.

Long term storage of information is provided by data storage as a service (DSaaS). DSaaS services offer storage capabilities of various types (the types are described in more detail later in this clause) and those capabilities can be offered both to CSC systems and also to other cloud services. DSaaS services are based on the storage resources of the CSP, typically accessed over the network through an API, although in some cases it may be possible and desirable to co-locate a DSaaS service with a compute service to remove network latency.



 

## 12.2 Common features of DSaaS

DSaaS enables users to store and retrieve information anywhere and anytime as long as there is connectivity to the DSaaS service. DSaaS services support scalability in terms of the volume of information stored and reliability in terms of access to the information from any type of application independent of the systems or devices on which those applications run.

Applications and systems use DSaaS to access cloud storage through related protocols. These protocols can support geographically remote storage resources and support virtualization of the storage locations used, so that when required, redundant or replicated storage is made available to provide resilience against point failures.

Storage services have the following common features:

a) **Durability**: Data are stored in one or more locations, controlled by the CSP. DSaaS services should provide data storage with no loss caused by natural disasters, human error or technical defects. This may be achieved through replicas or backups of the data, which can be provided as part of the service, or can be implemented by the CSC using the DSaaS service capabilities.

b) **Availability**: DSaaS services provide storage and retrieval of data on demand to meet the needs of the CSC's applications and systems.

c) **Security**: DSaaS services should store information securely. In particular there should be no unauthorised access to cloud service customer data. It is desirable to encrypt the information if appropriate, although this capability may be left for the CSC to implement since it can have cost and performance implications.

d) **Bounded costs**: With DSaaS, the customer typically pays only for the storage actually used. For some DSaaS services, information that is used less frequently can be stored in a lower-cost capability which might provide slower access as a means of reducing the cost.

e) **Manageability**: The DSaaS CSP has storage lifecycle management policies and processes that enable users and developers to focus on solving application problems and be free from concerns about the management of the information.

DSaaS can be classified by storage type and by service category as described in Table 1 and Table 2 respectively. Each storage service has one or more service interfaces such as block device driver, file system interface or object storage API, which is used by the client software. These APIs are network based since it is expected in most cases that the storage capabilities exist on a system other than the system running the client software.

**Table 1 — DSaaS according to storage type**

| Services | Features |
|---|---|
| File storage service | File storage services offer storage using a conventional file system model, with files contained in directories within volumes. Storage is typically offered to client software using the NFS protocol (NFS 4.2 - IETF 7862) and relevant volume(s) are mounted into the client environment via an NFS client driver. In particular, file storage services can be mounted into virtual machines and into containers to provide long-lived storage capabilities for those environments.<br><br>File storage services are usually networked based (akin to Network Attached Storage (NAS) devices) and can be shared by many clients simultaneously. Since the storage is virtualized by the file storage service, the storage can be highly scalable and it is often durable, with replicated redundant copies of the files being maintained, potentially in physically separated locations. Stored data can also be encrypted. (See Amazon EFS, IBM File Storage).<br><br>Many applications need access to shared files and a file system. This type of storage service is primarily supported on Network Attached Storage (NAS). This type of service is ideal for the use cases such as large-scale content repositories, development environments, media stores, or user home directories. |

**Table 1** *(continued)*

| Services | Features |
|---|---|
| Object storage service | Object storage services store the data as data objects in a flat, non-hierarchical namespace (the storage pool, bucket or container), where each object has a unique identity or key. Effectively, the object storage service operates on the basis of a key / value model, with the value as the object. In addition, each data object can have an arbitrary amount of user-specified metadata associated with it, potentially far richer than is possible with standard filesystems.<br><br>Object storage services are highly scalable, since they are not tied to specific storage hardware, and can span multiple storage devices. Individual objects can be very large also. Object storage services can be used by many client applications simultaneously.<br><br>Object storage services are typically offered via a REST API, which is naturally network capable and so accessible from clients remotely. The REST API (e.g. Amazon S3 API) is not like a conventional file system interface and so client applications have to be specifically designed and written to use object storage services.<br><br>Object storage services are particularly useful for unstructured data (e.g. images) which are updated relatively infrequently (updates are done by replacing the whole object with a new version). Object storage services are also usually slower than file storage services. |
| Block storage service | Block storage services offer high bandwidth low-latency access to storage devices at the block level. These services are providing the equivalent of Direct Attached Storage (DAS) or Storage Area Network (SAN) to the client system. This low-level form of access to storage resources allows the client more control and potentially higher performance than other kinds of DSaaS.<br><br>Thus, when using a block storage service, it is as if additional hardware storage devices are being attached to the client system. Block storage services are typically intended to work within one datacentre, since the latency increases substantially if the services are accessed over remote networks.<br><br>Typically, specialized SAN protocols such as iSCSI (IETF 7143) are used to deliver block storage services across the network to clients. The remote storage devices are presented to the client software as a mounted volume, just as if it were a disk locally attached to the system on which the client software is running.<br><br>Block storage services can have file systems built over them by the client software, or alternatively, the client software can use the block interface directly. This latter case can take place where the client software is database software (e.g. an SQL database), for example, or stream processing software (e.g. Apache Kafka). |

There are categories of cloud services that are inherently based on storage capabilities, but where the capabilities offered are more evolved than the simple storage of data and where the service interfaces offered to service clients typically reflect specific requirements of the client software. These categories of cloud service, described in Table 2, each use one or more of the storage types described in Table 1, but the storage aspect is not the main capability presented to clients.

**Table 2 — Storage services according to service category**

| Storage services | Features |
|---|---|
| NoSQL database services | NoSQL database services offer capabilities to store and retrieve various forms of unstructured data such as documents, images, movies, and large binary data. There is a wide variety of underlying technologies in this category, which can be classified in a variety of ways, such as:<br>— Key-Value Cache<br>— Key-Value Store<br>— Key-Value Store (Eventually-Consistent)<br>— Key-Value Store (Ordered)<br>— Data-Structures Server<br>— Tuple Store<br>— Object Database<br>— Document Store<br>— Wide Column Store<br>— Native Multi-model Database<br>— Graph Database |
| SQL database services | SQL (or relational) database services store structured data in a tabular format permitting potentially complex queries to be made to extract data to suit client requirements and to perform dynamic updates on the database content.<br>SQL is a standard interactive programming language designed for querying, updating, and managing data and data sets in the database management system. SQL is standardised in ISO/IEC 9075-1:2016 [71]. Modern SQL databases support the discovery of columns across a wide range of data set: not only relational table/views, but also XML, JSON, spatial objects, image-style objects (Binary Large Objects and Character Large Objects), and semantic objects.<br>Many different underlying SQL database technologies exist and are offered as cloud services. |
| Message queue service storage | Message queue capabilities are available as cloud services and are generally associated with a distributed asynchronous form of processing and system architecture that is increasingly common.<br>Many message queue systems have the capability to persist messages in storage. Messages can be persisted to ensure that they are not lost and can be retrieved whenever required, but can also be persisted to assist with stream processing that requires the large-scale analysis of many events in order to extract useful insights. Message persistence can place significant demands on the underlying storage systems due to the high volume and the high rate of delivery of messages. |
| Blockchain and Distributed Ledger Technology services | Distributed ledger technology (DLT) cloud services, such as Blockchain cloud services, support the provision and use of distributed ledgers, which are a form of transaction database.<br>DLT cloud services typically provide the capabilities of running a DLT node, including both an instance of the DLT platform software and the provision of the storage capabilities for a replica of the distributed ledger itself. For more details see ISO 23257 [26]. |
| Analytic services | Analytic processing capabilities are based on the processing of large quantities of data. Such large volumes and high velocity of data processing need particular support from the storage services that hold the data. Many CSPs provide specialised cloud storage services that support analytics. |

**Table 2** *(continued)*

| Storage services | Features |
|---|---|
| File management services | File management services provide an application capabilities type of cloud service, typically enabling automatic replication of files between user devices and cloud storage and offering the capability for multiple users to share and update files. |
| Federated storage services | In the case of federated storage services, storage resources can be combined transparently across a set of different cloud storage locations, including on-premises, private cloud or public cloud, whether offering file, block or object storage. Note that simple replication or failover capabilities do not typically imply that a storage service is a federated storage service. |

NOTE These storage services can be provided as persistent storage or as in-memory storage.

## 12.3 Capabilities type of DSaaS

ISO/IEC 17788:2014, 6.4 identifies three distinct capabilities types:

— Infrastructure (as seen in IaaS);

— Platform (as seen in PaaS);

— Application (as seen in SaaS).

DSaaS can offer one or more of these capabilities types as described in Table 3:

**Table 3 — Capabilities types of DSaaS**

| Capabilities type | Services |
|---|---|
| Infrastructure | File storage service<br>Object storage service<br>Block storage service<br>Federated storage service |
| Platform | Customer-programable data storage, where customer-written code can be uploaded and used to manipulate the data storage<br>Analytic service<br>NoSQL database service<br>SQL database service<br>Message queue service<br>Blockchain and DLT service |
| Application | Human-facing user interface for manipulating storage, such as a web-based document repository:<br>File management service |

The cloud capabilities types applying to the cloud services covered by Table 1 and Table 2 vary as shown in Table 3. Most of the cloud services covered by Table 1 are typically provided as infrastructure capabilities type cloud services. For Table 2, the cloud services are typically of the platform or application capabilities types, depending on the details of the offering, except in the case of federated storage services, which are more typically infrastructure capabilities type services.

## 12.4 Significant additional capabilities of DSaaS

In addition to the expected capability of storing data, many cloud storage services offer significant additional capabilities that can be important to cloud service customers.

The first set of capabilities relates to resilience and resistance to point failures in the cloud service providers' infrastructure. Many cloud storage services store data in multiple redundant replicas, so that a failure of a single storage device or a failure of access to a single device does not lead to unavailability of the service or, worse, loss of data. The nature of the replication can vary. In some cases, replicas are deliberately placed in a physically separate location (e.g. a different data center or a different availability zone within one data center), with the aim of addressing major failures of a complete data center. In other cases, particularly a cloud service involved in high performance access, the replicated locations can be deliberately close to each other. Some cloud storage services provide capabilities for the cloud service customer to choose the policy relating to the placement of replicas.

The second set of closely related capabilities relate to the creation and storage of backups of the data. These backups can either be automated or performed at the request of the cloud service customer. The backups may be to a "live" location (i.e. storage that is online and available, but at another location) or to an offline location. The latter case could be used for long-term retention at lower cost.

Another capability offered by some cloud storage services is resource affinity. This relates to the relative physical placement of the cloud storage service instances in relation to other cloud services, mainly compute service instances such as virtual machines and containers. One of the major reasons to place compute service instances close to storage service instances is performance, both in terms of reducing latency to a minimum and also in terms of maximising bandwidth for data flowing between the compute and storage instances. Some categories of service such as block storage services are often only offered in this form. For example, block storage services might only be offered for use to compute service instances running on nodes in the same datacentre where there is a high speed link between the storage node and the compute node such as Ethernet or Fibre Channel.

# 13 Networking in cloud computing

## 13.1 Key aspects of networking

Networking is a key element of cloud computing. The very definition of cloud computing is based on its capabilities being accessed via networks: *paradigm for enabling network access to a scalable and elastic pool of shareable physical or virtual resources with self-service provisioning and administration on-demand* [ISO/IEC 17788:2014].

Networks and network-related capabilities are also some of the resources that are often provided by means of cloud services.

Thus, there are two broad areas of concern relating to networking in cloud computing. The first is the networking by which a given cloud service is accessed and by which any capabilities within the cloud service are accessed, such as an application running within a compute service. This is termed "cloud access networking", or "public cloud access networking" where public cloud services are involved. The second is the networking used to connect cloud service instances to each other. This networking is often by design intended not to be accessed outside of the particular cloud service instances involved. This is termed "intra-cloud networking".

## 13.2 Cloud access networking

Cloud services have externally accessible interfaces that enable use of their capabilities by cloud users and by systems acting on behalf of cloud users. Some cloud services also have externally accessible interfaces for CSC capabilities running inside the cloud service: examples include interfaces such as web interfaces or APIs for CSC applications running within a compute service instance (e.g. within a VM or within a container) and also interfaces to data storage capabilities of a storage service instance (e.g. a file store).

It is often the case that the externally accessible interfaces are publicly visible on the internet, at least for public cloud services. However, the externally accessible interfaces can in some cases be deliberately hidden from public view on the internet, even where user access takes place over the internet infrastructure. Private cloud services can be deployed in a way that makes them accessible

only within the organization's private networks (i.e. such cloud services are not accessible from the internet), although there can still be a requirement for some externally accessible interfaces to be available, for example where a web application deployed to a private cloud service needs to present a publicly available interface accessible by its users.

Applications running in a cloud service can require externally visible interfaces available on a publicly accessible network address and port number. It is typical for such a publicly accessible interface to be provided as a "virtual interface" where the external interface is presented on a network address and port known to the cloud service itself, while the code running within the cloud service runs on some internal network address and port, to which the external address and port are mapped. This endpoint virtualization is necessary to permit the resource sharing that is fundamental to cloud services. Endpoint virtualization also supports capabilities such as load balancing across multiple instances of an application and security capabilities including firewalls and DDoS attack handling.

Publicly visible interfaces can also require specific configuration. In particular, an organization running an application on a cloud service could find it highly desirable that the address used for the interface is one that belongs to the organization and not one that belongs to the CSP. The capability to support this is generally termed "Bring Your Own IP addresses" (BYOIP) and is a capability where the CSC can configure the publicly visible interface for an application running in a cloud service with an address belonging to the CSC. This can apply to both public cloud services and private cloud services.

## 13.3 Intra-cloud networking

Within the cloud service environment, it is typical that networking is used to connect together the various components that make up a system. For compute capabilities type cloud services, for example running software in VMs or containers, there are often multiple instances running that need to communicate, either with each other (for example, where the application is divided into separately running components, as with a microservices architecture) or with other components of the solution (for example, with a load balancer where horizontal scaling is used with multiple parallel instances of a given component). It is also typical for storage services to be connected to compute instances and this is usually done over the network.

There may also be multiple separate layers of intra-cloud networking. The application layer as described in the previous paragraph and also the management or control layer, used to monitor and control each of the cloud services. These different layers are deliberately isolated from each other so that they cannot interfere with each other.

It is commonly the case that intra-cloud networking is virtualized. The various cloud services do not use networking capabilities directly, but use virtualized networking capabilities that both permit sharing of the underlying networking resources and also provide isolation between different groups of cloud service instances both for security reasons and also to avoid interference.

The structure and organization of virtualized networks can also deliberately avoid reflecting the organization of the underlying physical networks. It is often the case that components of a solution are distributed across multiple availability zones in one data centre, or across multiple data centres. It can be highly undesirable for this physical organization to be made visible to the solution components and so a single unified virtual network is presented to those components, overlaid on the physical networks.

Virtualized compute environments, both VMs and containers, involve tight control and virtualization of network resources, including both the endpoints exposed by each VM/container and also the target network endpoints used by the software running within the environments. The capability of running multiple VMs on one system, or multiple containers on one operating system, clearly requires mapping of each of the network endpoints exposed by the software to actual endpoints in the containing system, simply to enable the sharing of the system without resulting in clashes. The deployment of both VMs and containers requires configuration to deal with these issues.

It is also commonly the case that both VMs and containers are used in distributed environments. Multiple instances of the same software can run on different systems and those systems can run in different physical locations. Applications are also commonly divided into multiple independent components (services, microservices), and these components can each run in separate locations. It is a

good practice that the actual locations of the components be hidden from the software since application components need to communicate with each other seamlessly wherever they are running. In addition, it is best if the application components can only communicate with each other. External communication should be carefully controlled through specifically designated external endpoints.

It is a best practice in relation to networking for these software architectures that networking is effectively defined at the application level. This is a virtual network that is used only by the application components and which transparently spans all the locations in which application components are running. The implication is that each application has an associated virtual network and that, as with virtual compute environments like containers, each virtual network is isolated from other networks.

Virtual networks can be built using a variety of techniques and technologies, including software defined networks (SDNs) and network function virtualization (NFV), some built for specific types of components and cloud services such as:

— VxLAN: an overlay network technology, specified in IETF RFC 7348[41];

— Kubernetes[17] networking;

— Container networking systems, such as Calico[43] and Weave Net[44].

## 13.4 Virtual private networks (VPNs) and cloud computing

One technology that can be useful in building solutions using cloud computing are virtual private networks (VPNs). VPNs offer a secure means of connecting systems together where part of the networking infrastructure involves using untrusted environments such as the internet. There are two typical configurations of VPN as shown in Figure 9, reflecting different usecases:

— Host-to-Gateway

  This is where a stand-alone system, typically a client machine or device, accesses a secured network remotely. In the case of cloud computing, the typical usecase would be for a client device to access cloud services and applications and other resources running within them.

— Gateway-to-Gateway

  This is where secure network communications are supplied between two separate secure networks. The typical usecase is where cloud services in a cloud environment either need to be connected to CSC in-house applications and systems, or need to be connected to other cloud services running in a different cloud environment.