Cloud Computing

> Cloud computing is Internet-based computing. Shared resources, software, and information are provided to computers and other devices on demand.

> Cloud Computing is a style of computing in which dynamically scalable and virtualized resources are provided as a service over the Internet.

> Cloud computing is a style of computing that provides scalable and elastic IT capabilities as a service to customers using internet technologies.

> Cloud computing is an infrastructure management technology: automatically aggregated to deliver services

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or interaction.

> Cloud: A coherent, large-scale, publicly accessible collection of compute, storage, and networking resources. Available via Web service calls through the Internet. Short or long-term access on a pay per use basis.

- On-demand self-service: little or no interaction with cloud provider
- Broad network access: capabilities are available over the network and accessed through standard mechanisms
- Resource pooling: capabilities are available over the network and accessed through standard mechanisms
- Rapid elasticity: Capabilities can be rapidly and elastically provisioned to quickly scale up; and rapidly released to quickly scale down, automatically. To consumers, the capabilities available for provisioning appear to be unlimited and can be purchased in any quantity at any time.
- Measured service: cloud systems automatically control and optimize resource usage by leveraging a metering capability at some level of abstraction. Resource usage can be monitored, controlled, and reported.

> A Cloud is a type of distributed system consisting of interconnected and virtualized computers dynamically provisioned and presented as one (or more) unified computing resource(s) based on service-level agreements established through negotiation between service providers and consumers

Central ideas:

- Utility Computing
    - users concern about what they can get, their quality of service, to pay as many as they used
- SOA: Service Oriented Architecture

    - Service is what you connect together using Web Services
    - Web service is self-describing and stateless module that perform discrete units of work and are available over the network
    - SOA is a collection of services which communicate with each other
    - SOA provides a loosely-integrated suite of services that can be used within multiple business domains
- SLA: Service Level Agreement
    - Quality of Service (QoS) is a set of technologies for managing network traffic in a cost effective manner to enhance user experiences for home and enterprise environments.
    - Now also: customer care evaluations, technological evaluations
    - A service-level agreement is a contract between a network service provider and a customer that specifies, in measurable terms (QoS), what services the network service provider will furnish

Enabling techniques:

- Hardware virtualization
- Parallelized and distributed computing
- Service-oriented computing
- Autonomic Computing

Properties and characteristics:

- High scalability and elasticity
    - Scalability: a property of a system, a network, or a process, indicates its ability to either handle growing amounts of work in a graceful manner or to be readily enlarged
    - Elasticity: The ability to enforce adaptive “autonomic” actions to enable scalability
    - How to achieve these properties?
        - Dynamic provisioning
        - Multi-tenant design
    - Dynamic provisioning: instances are provisioned or deployed from a administrative console or client application (Amazon auto-scaling groups)
    - Multi-tenant design: a principle in software architecture where a single instance of the software runs on a server, serving multiple client organizations.
    - Multi-tenant architecture: a software application is designed to virtually partition its data thus each client organization works with a customized virtual application instance.
- High availability and reliability
    - Availability: the proportion of time a system is in a functioning condition
    - Reliability: the ability of a system to perform its required functions under stated conditions for a specified period of time
    - How to achieve these properties?
        - Fault tolerance system: the property that enables a system to continue operating properly in the event of the failure of some of its components+
            - If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naively-designed system in which even a small failure

            can cause total breakdown.
        - Four basic characteristics :
            - No single point of failure
            - Fault detection and isolation to the failing component
            - Fault containment to prevent propagation of the failure: Mechanisms that isolate a rogue transmitter or failing component to protect the system are required.
            - Availability of reversion modes: System should be able to maintain some check points which can be used in managing the state changes.
    - Require system resilience
        - Resilience is the ability to provide and maintain an acceptable level of service in the face of faults and challenges to normal operation.
        - Disaster recovery is the process, policies and procedures related to preparing for recovery or continuation of technology infrastructure after a disaster.
    - Reliable system security: data protection, identity management, application security, privacy
- High manageability and interoperability
    - manageability: Enterprise-wide administration of cloud computing systems.
    - interoperability: a property of a system to work with other systems without any restricted access or implementation.
    - But how to achieve these properties ?
        - System control automation
        - System state monitoring: a System Monitor in systems engineering is a process within a distributed system for collecting and storing state data.
            - Billing system: Users pay as many as they used, get those information by means of monitoring system.
- High accessibility and portability
- High performance and optimization

AutoScaling: respond automatically to changing conditions. (launch or terminate EC2 instances automatically)

Cloud Computing Models:

- Public Cloud: the cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.
    - Homogeneous infrastructure
    - Common policies
    - Shared resources and multi-tenant
    - Leased or rented infrastructure
    - Economies of scale
- Private Cloud: The cloud infrastructure is operated solely for an organization. It may be managed by the organization or a third party and may exist on premise or off premise.
    - Heterogeneous infrastructure
    - Customized and tailored policies
    - Dedicated resources

    - In-house infrastructure
    - End-to-end control
- Community Cloud: The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns
- Hybrid Cloud: The cloud infrastructure is a composition of two or more clouds that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability

Stateless components: Implement the components in a way that they do not contain any internal state, but completely rely on external persistent storage (Distributed data stores).

Elastic components: Monitor the utilization of compute nodes that host application components and automatically adjust their numbers using the provisioning functionality provided by the elastic infrastructure

Elastic load balancer: use an elastic load balancer that determines the amount of required resources based on numbers of requests and provisions the needed resources accordingly using the elastic infrastructure’s API

Service Models

Infrastructure as a Service - IaaS

- The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources
- The consumer is able to deploy and run arbitrary software, which can include operating systems and applications
- Examples: OpenStack
- Virtual machine, virtual storage, virtual network
- virtual infrastructure manager: software toolkit responsible for orchestration, operating system that deals with multiple computers, presenting a uniform view to apps
- Needs a system monitoring interface
- Abstract the infrastructure

Platform as a Service - PaaS

- The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly application hosting environment configurations.
- Developers do not need to care about installation, configuration and operation of servers, run time and middleware
- The application life-cycle is turned into a continuous development/deploy
- Examples: App Engine, Heroku

- Integrated environment that supports the development, running and management of cloud-based applications
- Main services belonging to PaaS
    - Application development and execution services
    - Integrated lifecycle services
    - Workload management services
    - Data management services
- System control interface:
    - Policy Based Control: Make the decision according to some requirements
    - Workflow control: the flow of installation and configuration of resources

Software as a Serice - SaaS

- Enabling Technique – Web Service
- Provide service – Web-based Applications
- Provide service – a way for enterprises to provide a consistent look and feel with access control and procedures for multiple applications and databases

Service Science

> Service Science is a discipline that enconpasses the interdisciplinary study of computer science, operations research, mathematics, decision-making theory, social and cognitive sciences, and other fields.

> Service Science is the interdisciplinary application of science, engineering, and management for the purpose of improving services. Service Science also contributes to systematic innovation and improved productivity, and is the guiding force for the improvement of services through improved predicatbility in the productivity, quality, performance, compliance, development and operational innovation in services.

Service science is hard because it is multidisciplinary:

- Service depend critically on people, technology, organizations, and co-creaton of value
- People work together, with technology and organizations to provide value
- Shared information
- A service system is a complex socio-techno-economic system
- Service System are both designed and shaped by evolutionary forces

Need of a science of services to better understand service system. Service science aims to research services in the same scientific manner that spurred success in the manufacturing industry, and to make it easier to increas productivity through a scientific approach to modeling. At the same time also aims to make it easier for clients and providers to forecast the future effects and risk of introducing services, and to allow rational sharing of these effects and risks.

System: any set of available variables selected by an observer to identify fundamental objects, the influential attributes of the objects, and the relationships of these objects that result in a phenomena.

Service System main enablers: technology, information, people, process

Service science professional: T-shaped professional. Broad skillset with deep knowledge in at least one area. Able to analyze large service systems across various sectors of the service company. Ensure that new services consider the IT alternatives and factors such as human system.

System of engagement: improve interactions with customers, enable self-service to increase satisfaction, create new business model. IT should interact the same way consumers and employees do. The external, visible part of a system, focusing on people and products, not processes.

System of records: traditional systems that handle business transactions and enterprise records. Providing core transactional systems, database are at the center of these systems. Essential to the day-to-day operations of an organization. The internal, invisible part of a system, focusing on business processes, transactions, and accounting

System of insight: convergence of engagements and records. Consume, detect, decide, report, analyze, collect. Loop of activities:

- discover the insights that matter most;
- embed them into the software their customers and employees use to engage; and
- continuously measure and learn from the results.

Edge Computing & IoT

IoT deals with ubiquitous communication and connectivity

- Old definitions focus mostly on connectivity and sensory requirements for entities involved in typical IoT environments.
- New definitions focus give more value to the need for ubiquitous and autonomous networks of objects where identification and service integration have an important and inevitable role.

As data volume and velocity increases we need to rethink our Cloud/Data Centre architecture. Understand when and what data can be analyzed close to where it is captured: not all edge data is essential and transfer is expensive.

Edge computing: Extending computing to the edges of the network;

Fog computing and edge computing appear similar since they both involve bringing intelligence and processing closer to the creation of data. However, the key difference between the two lies in where the location of intelligence and compute power is placed. A fog environment places intelligence at the local area network (LAN). This architecture transmits data from endpoints to a gateway, where it is then transmitted to sources for processing and return transmission. Edge computing places intelligence and processing power in devices such as embedded automation controllers.

SOA & APIs

> SOA is an architectural model to provide services over the Internet. A service provides features and functions to independent clients.

> An Application Programming Interface (API) is a specification intended to be used as an interface by software components to communicate with each other. An API may include specifications for routines, data structures, object classes, and variables

Each service should:

- Provide a description to be discovered and selected
- Provide access through well-known protocols
- Support composition to deliver complex solutions
- Address client's business needs and domain requirements

Architecture: discovery agencies, service provider, service requestor.

API vs SOA:

- Consumers are Internal and External developers; Consumers are Internal (and maybe partner) developers
- Embracing of open community/social business is critical; Promote reuse within a company and sometimes with partners
- REST, leverage HTTP for Internet scale; SOAP & protocol independent headers
- Easy of use based on simplicity and readability; Interoperability and tooling consumption based on WSDL
- True 'black box' separation between Web API and consuming app: simple contract; More extensive contract between service provider and consumer in enterprise implementations

API Economy

A new as-a-service economy is emerging with value being provided and consumed through APIs.

API business model:

- Free: brand loyality, increase popularity; Facebook
- Developer pays: assets of high value to the developer; Amazon
- Developer gets paid: provide incentive for developer; Google ads
- Indirect:

REST

REST is built around the idea of simplifying agreement

- nouns are required to name the resources that can be talked about

- verbs are the operations that can be applied to named resources
- content types define which information representations are available

> Representational State Transfer. REST is an architectural style for distributed systems. A system that exhibits all defined constraints is RESTful. Systems may add additional constraints or relax existing constraints, in which case they are more or less RESTful.

> REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations.

An architectural style is:

- a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.
- an abstraction, a design pattern, a way of discussing an architecture without concern for its implementation.

Rest principles:

- Resources are defined by URIs
- Resources are manipulated through their representations
    - Multiple representations for a resource
- Messages are self-descriptive and stateless
- Application state is driven by resource maniputations
- Hypermedia is the way to control application behavior

Resources:

- any information that can be named, anything important enough to be referenced
- if you cannot name something, you cannot do anything with it
    - a popular resource type on the Web are documents
    - documents usually are a structured collection of information
- have states that may change over time
- have identifiers
- expose uniform interface
- on request a resource may transfer its representation to the client (client-server architecture)
- a client may transfer a proposed representation to a resource (manipulation through representations)
- Representations returned from the server should link to additional application state.
- Stateless interactions
    - Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server
    - Statelessness necessitates self-descriptive messages
- Uniform interface + Stateless + Self-descriptive = Cacheable
    - Layered System

Why constraints?

- “There are irreconcilable differences between local and distributed computing”
- Constrained systems emphasize simplicity and awareness of the system’s context
- Constraints are the realization of design goals
- Unconstrained distributed systems are difficult to use, brittle, unreliable, and of marginal utility

Rest is standard:

- correct and complete use of the HTTP protocol
- lightweight and layerd mechanisms for data and service integration
- distributed, hypermedia-driven application platform

Methods:

- OPTIONS (idempotent): represents a request for information about the communication options available

- GET (cache, safe): means retrieve whatever information (in the form of an entity) is identified by the RequestURI

- HEAD (cache, safe): identical to GET except that the server MUST NOT return a message-body in the response

- POST ( ): is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified

- PUT (idempotent): requests that the enclosed entity be stored under the supplied Request-URI

- DELETE (idempotent): requests that the origin server delete the resource identified by the Request-URI

- TRACE (idempotente): is used to invoke a remote, application-layer loop- back of the request message

- Safe: methods SHOULD NOT have the significance of taking an action other than retrieval

- Idempotent: the side-effects of N > 0 identical requests is the same as for a single request (aside from error or expiration issues).

> Put vs POST
>
> When creating new resources:
>
> - Use POST if the server chooses the URI: when creating, POST does “something” and returns “something”
> - Use PUT if the client chooses the URI : when updating

Web technologies MUST work in a RESTful way (if they work differently they are not web technologies):

- resources are identified by URIs and can be navigated
- resources have a state and can be accessed using HTTP
  - State is represented as part of the content being transferred
  - clients can simply store the representation to save the state: it is possible to switch between servers for different interactions
- state transitions are modelled as resource accesses
- Representations should (not must) be machine-processable

Hypermedia control: HATEOAS - Hypermedia as the engine of application state (Hypermedia = links and forms)

- The server "guides" the client to new states by providing links inside hypertext representations
- Consumer requires only a single URI to bootstrap: the hypermedia representation is the interface

Containers & Docker

Apps + libs + tools should be shipped together to simplify management and provide consistent behaviour: use containers.

> Containers: A standard way to package an application and all its dependencies so that it can be moved between environments and run without changes. Containers work by isolating the differences between applications inside the container so that everything outside the container can be standardized.

Containers vs VM: VM run on hypervisor and are "heavy", OS on top of OF. Containers are light and run on an engine and they share the underlying kernel.

Docker uses:

- development and build faster, efficient and lightweight
- stand-alone services across multiple environments
- create isolated test instances
- building and testing complex applications on a local host
- building multi-users PaaS and SaaS
- lightweight and stand-alone sandbox environment for developing, testing and teaching technologies

Docker Basics:

- Image, read only snapshot of a container saved in a registry
  - Often, an image is based on another image, with some additional customization.
  - To build an image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it.
  - When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt.
  - A Docker image is made up of filesystems layered over each other.

    - It uses the union mount technique that allows several filesystems to be mounted at one time but appear to be one filesystem.
    - Copy on Write pattern: When Docker first starts a container, the initial read-write layer is empty. As changes occur, they are applied to this layer; for example, if you want to change a file, then that file will be copied from the read-only layer below into the readwrite layer. The read-only version of the file will still exist but is now hidden underneath the copy.
    - created with docker commit (manual) and docker build (dockerfile)
- Container: standard running unit in which the application services resides
    - Runnable instance of an image
    - a container is relatively well isolated from other containers and its host machine
    - A container is defined by its image as well as any configuration options you provide to it when you create or run it
    - When a container stops, any changes to its state that are not stored in persistent storage disappears
- Registry: stores and distributes images
- Docker Engine: create, ships and runs containers, runs on any platform, client communicates with engine to execute commands
    - client-server application: server/daemon + API + CLI
    - The engines manages Docker images, containers, networks, volumes, …

> Docker Compose allows you to run stacks of containers to represent application stacks. It is an open source basic Docker orchestration tool.

> Swarm: Multi-container, multi-machine applications are made possible by joining multiple machines into a “Dockerized” cluster called a swarm. A swarm is a group of machines that are running Docker and have been joined into a cluster. The machines in swarm, they are referred to as nodes. Docker commands are executed on a cluster by a swarm manager, which can use several strategies to run containers.

Copy on write:

- Copy-on-write is a strategy of sharing and copying files for maximum efficiency. If a file or directory exists in a lower layer within the image, and another layer (including the writable layer) needs read access to it, it just uses the existing file. The first time another layer needs to modify the file (when building the image or running the container), the file is copied into that layer and modified. This minimizes I/O and the size of each of the subsequent layers.
- When an existing file in a container is modified, the storage driver performs a copy-on-write operation. The specifics steps involved depend on the specific storage driver. For the aufs, overlay, and overlay2 drivers, the copy-on-write operation follows this rough sequence:
    - Search through the image layers for the file to update. The process starts at the newest layer and works down to the base layer one layer at a time. When results are found, they are added to a cache to speed future operations.
    - Perform a copy_up operation on the first copy of the file that is found, to copy the file to the container’s writable layer.

    - Any modifications are made to this copy of the file, and the container cannot see the read-only copy of the file that exists in the lower layer.

Benefits of Docker:

- ease of use: quickly build and test portable applications.
- speed: containers are lightweight and fast, taking up fewer resources
- docker hub: app store for docker images
- modularity and scalability: break application functionality into individual containers

Microservices

A change is needed in systems design, deployment and maintenance:

- Traditional systems are single systems developed top down by decomposition
    - tight coupling: a single organization maintain the system
- Contemporary distributed systems need to include thirdparty subsystems
    - loose coupling: each component is maintained independently from the others
    - serve different systems in different ways

A Software Ecosystem consists of the set of software solutions that enable, support and automate the activities and transactions by the actors in the associated social or business ecosystem and the organizations that provide these solutions.

The approach is to develop:

- components as microservices, and
- containers as deployment components

No more interfaces: move away from WSDL to use communication protocols. Use REST and HTTP.

Services tend to be large and monolithic, microservices are small, focused, autonomous and independent.

Microservices are small enough to be handled by a single cross-functional team and not feel “big”

Advantanges:

- Composability
- Replaceability
- Deployment of smaller pieces with lower risk
- Selective scaling
- Resilience against failure
- Maintainability: alignment with the organization structure

A microservices architecture

- puts each element of functionality into a separate service

- makes each service an independent unit of deploy
- uses software-engineering practices to drive the architecture

Overview:

- Products not projects: development team takes full responsibility for the software in production
- Coordination not control: decentralized governance and data management
- Infrastructure automation: dev-ops
- evolutionary design: getting boundaries right for ease of refactoring
- Microservices is about people, minimize communication and coordination reducing the scope and the risk of the change

Issues in monolithic software:

- difficult to maintain
- suffer from the dependency hell
- require rebooting for any change in one module
- expensive or sub-optimal deployment with respect to its individual modules
- technology lock-in for developers
- suffer from reliability problems: a bug in any module can bring down the whole infrastructure

Microservices architecture:

- set of smaller interconnected services
- each microservice is a mini application that would expose an API and a WebUI
- At runtime each instance is often a cloud VM or a container
- Communication is mediated by an API Gateway: responsible for tasks such as load balancing, caching, access control, API metering, ….
- designed according to single responsibility principle
- scaling: running multiple copies of an application, splitting the application into different services, each server is responsible for only a subset of the data
- single responsibility principle: every module or class should have responsibility over a single part of the functionality
- tackles the problem of complexity: the application has been broken up into manageable chunks or services, each service has a well-defined boundary in the form of an RPC- or message-driven API.
- enables each service to be developed independently by a team that is focused on that service.
- enables each microservice to be deployed independently
- enables each service to be scaled independently

Drawbacks of microservices:

- The term microservice places excessive emphasis on service size.
- the complexity that arises from the fact that a microservices application is a distributed system (handle partial failure, sync, etc)
- partitioned database architecture: you end up having to use an eventual consistency based approach, which is more challenging for developers
- testing a microservices application is also much more complex.

- hard to implement changes that span multiple services

Microservices deployment:

- Deploying a microservices-based application is also much more complex
- A service discovery mechanism is needed to enable a service to discover the locations
- Successfully deploying a microservices application requires greater control of deployment methods by developers, and a high level of automation
- One approach to automation is to use an off-the-shelf PaaS such as Cloud Foundry.
- Another way to automate the deployment of microservices is to develop what is essentially your own PaaS.

Hidden dividends:

- Permissionless innovation is about “the ability of others to create new things on top of the communications constructs that we create”.
- Enable Failure: failure of individual services should be expected, whereas the cascading failure of all services should be impossible.
- Disrupt Trust: Microservices can provide an effective model for evolving organizations that scale far beyond the limits of personal contact.
- You Build It, You Own It
- Accelerate Deprecations
- End Centralized Metadata: Consumers should not know or care about how data persists behind a set of APIs on which they depend, and indeed it should be possible to swap out one persistence mechanism for another without consumers noticing or needing to be notified
- Concentrate The Pain: As microservices proliferate, it can be possible to ensure the most severe burden of compliance is concentrated in a very small number of services, releasing the remaining services to have a higher rate of innovation, comparatively unburdened by such concerns
- Test Differently: The adoption of practices such as continuous deployment, smoke tests, and phased deployment can lead to tests with higher fidelity and lower time-to-repair when a problem is discovered in production.