# APPUNTI CLOUD AND GREEN COMPUTING A.A 2019/2020

## 1°LEZIONE: intro

### Introduzione

Il Cloud è un modello che è utile approfondire per tre motivi:

- Offerta di lavoro
- Sapere cos'è per un lavoro futuro
- Business

Il mondo attuale sta puntando sempre di più sull'offrire servizi anziché beni, ad esempio, l'utilizzo del car o bike sharing invece di acquistare il mezzo di trasporto e doversi occupare della conseguente manutezione. Allo stesso modo il Cloud offre "in prestito" memoria, server, piattaforme, software ecc. senza bisogno di doverli acquistare.

### Quality of Service (QoS)

La QoS è fondamentale per qualsiasi servizio che utilizziamo, non basta che sia conveniente economicamente ma deve anche essere affidabile. Le informazioni sull'affidabilità vengono scritte nel **Contratto** che i clienti spesso (quasi sempre) ignorano e accettano senza leggere.

### Service Level Agreement (SLA)

È quella parte del contratto dove viene indicata l'affidabilità di un servizio.
Viene scritta da tre diverse figure:

- Avvocato specializzato nel settore
- Business expert
- Sviluppatore/i

Spesso l'effettiva affidabilità non viene riportata poiché è molto difficile stabilire a priori quanto un servizio sarà efficiente, ed inoltre, anche sapendo con certezza questo dato, esso dovrà competere con le altre aziende, il cui indice di affidibilità potrebbe essere più alto.

### Esempi SLA:

**Cloud Amazon S3**
https://aws.amazon.com/it/s3/sla/

**Da nessuna parte** nello SLA di Amazon S3 è riportato che abbiamo un qualche tipo di garanzia o indice di affidabilità, c'è soltanto una spiegazione su quanto "credito" l'utente può ricevere se si dovessero verificare dei disservizi. Leggendo attentamente le clausole si arriva alla conclusione che un eventuale rimborso è quasi impossibile da ricevere.

**Cloud Microsoft**
Microsoft non ci garantisce neanche che il cloud sia sicuro, anzi ci invita a fare regolarmente un backup dei nostri dati del cloud su un'altra piattaforma (e allora per cosa lo usiamo il Cloud?).

**Facebook**
Facebook ci avverte che qualsiasi contenuto condiviso pubblicamente può essere utilizzato da Facebook o essere ceduto a terzi in tutto il mondo che possono utilizzarlo a loro volta per qualsiasi scopo.

2° LEZIONE:

## Richiesta di un servizio

**Overprovisioning:** porta a sprecare risorse poiché si prevede un flusso maggiore di richieste di quelle effettive, ma anche nel caso di una previsione corretta avviene comunque una perdita economica.

**Underprovisioning**: porta a errori e crash del sistema con conseguente perdita di utenti.

È molto difficile prevedere con precisione il numero di richieste, talvolta sono prevedibili, come ad esempio l'aumento di visite su un sito di prodotti natalizi nel periodo di Natale, altre volte non sono prevedibili, ad esempio, una piccola applicazione che viene consigliata casualmente da un influencer con milioni di followers e non riesce più a reggere il flusso di utenti.

Per questo è nato il concetto di **Cloud** dove non serve fare previsioni sulle richieste poiché i server sono "illimitati"

## Cloud

Il Cloud è un **modello** per permettere un accesso via rete diffuso, conveniente e su richiesta a un insieme condiviso di risorse di calcolo configurabili (p.e. rete, server, memoria, applicazioni, servizi) che possono essere rapidamente fornite e rilasciate con un minimo costo di gestione o di interazione col fornitore del servizio.

Idee chiave:

- Raggruppare in modo efficiente e on-demand infrastrutture virtuali, offerte come servizi
- Fornire risorse dinamicamente scalabili, virtualizzate, a molti clienti attraverso Internet
- Separare la fornitura di servizi di calcolo dalla tecnologia sottostante (opaca agli utenti)

**Perché il Cloud ha rivoluzionato l'informatica?**
Per **l'attrattività economica,** infatti non serve un'infrastruttura hardware e non serve quindi un capitale iniziale per l'acquisto dell'hardware.

È molto amato dagli utenti il **Pay-per-use** (on-demand) dove paghi soltanto ciò che usi, viene preferito anche se alla fine si va a spendere di più perché garantisce **elasticità** e permette il **trasferimento dei rischi** su terzi, ma al tempo stesso ci si lega tramite l'SLA a terze parti dalle quali potrebbe essere difficile distaccarsi in un secondo momento.

Il modello di business introdotto dal cloud computing permette di trasformare alcuni costi fissi in costi variabili e permette uno shift da spese CapEx a spese OpEx.

## Modelli di servizio

**IAAS: Infrastucture as a service**
Fornisce server, memoria, rete (virtualizzati), il fornitore di servizi IaaS gestisce tutta l'infrastruttura, il cliente è responsabile di tutti gli altri aspetti del deployment (p.e. sistema operativo, applicazione).
Trasforma un server di alluminio hardware in un servizio software.
**Il mercato del Cloud IaaS è cresciuto del 31.3% nel 2018**

*Esempi: EC2, S3*

**PAAS: Platform as a service**
Fornisce un'intera piattaforma come un servizio (macchine virtuali, sistema operativo, servizi, ambiente di sviluppo) Il fornitore PaaS gestisce infrastruttura + sistema operativo + enabling software, Il cliente è responsabile di installare e gestire l'applicazione.

*Esempi: Heroku, Azure, GAE*

**SAAS: Software as a service**
Fornisce software on-demand accessibile mediante client thin o API. Il fornitore SaaS gestisce infrastruttura + sistema operativo + applicazione, il cliente non è responsabile di niente.

*Esempio: Salesforce.com, Google Drive*

Ordine di gestione da parte dell'utente: IAAS < PAAS < SAAS

La differenza tra IAAS e PAAS è lieve, se offri PAAS spesso sei obbligato a offrire anche IAAS.

## Modelli di deployment

**Pubblico**
Il vantaggio è la scalabilità, però i dati sono resi pubblici (ad esempio dati ad Amazon se si usufruisce di S3, se succede qualcosa nei loro data center i nostri dati vengono violati)

**Privato**
Non ha molta scalabilità però è più controllato e sicuro.

**Ibrido**
Sono una via di mezzo

## Ostacoli all'adozione del Cloud

Tre ostacoli principali:

**Confidenzialità dei dati:** fisicamente in quale Data center saranno memorizzati i nostri dati o i dati dei nostri clienti? È garantita la privacy? Cioè i dati sono crittografati? Qualcuno può infiltrarsi nella comunicazione in rete e modificare i nostri dati senza che ce ne accorgiamo? Se avviene un problema nel Data center e la privacy viene a mancare noi lo verremmo a sapere oppure no?

**Disponibilità dei servizi:** cosa succede se un Cloud Provider fallisce? Spesso le aziende si affidano ad un solo Cloud provider ma bisogna usarne di più. Dobbiamo anche mettere in conto che ci saranno dei fallimenti temporanei e il servizio potrebbe non essere disponibile in alcuni momenti.

**Vendor lock-in:** più vengono utilizzate funzioni del Cloud (ad esempio funzionalità di login e interfaccia senza crearla da zero) più rimaniamo "incatenati" all'interno di quello specifico Cloud perché se proviamo a spostare i nostri dati dovremmo modificare tutte le nostre applicazioni per poterle far funzionare anche su altri Cloud. È un lock-in tecnologico.

## Nuovi Modelli di Business

Sono state innovative le idee di **Dropbox, Spotify, Google**.

Dropbox ha deciso di offrire memoria gratuita a tutti, Spotify musica gratuita a tutti pagando la SIAE ogni volta che viene riprodotta una canzone, e Google un motore di ricerca gratuito finanziandosi grazie al **customized advertising**.

### Green Computing

Per costruire un computer ci vogliono **2 tonnellate** di materiale grezzo, l'ICT in generale produce più emissioni di $CO_2$ degli aerei.

Uno dei fattori che contribuisce all'inquinamento è l'obsolescenza programmata e soprattutto **l'obsolescenza percepita**, cioè il venditore cerca di spingere il cliente ad acquistare un nuovo prodotto che avrà nuove funzionalità anche se al cliente queste non servono, ma il venditore farà in modo di fargli sentire questa "mancanza". Quindi c'è un gran numero di rifiuti tecnologici.

In Arizona ci sono più di **40 Data Center** perché l'elettricità costa pochissimo, chiaramente le temperature del luogo richiedono grossi sistemi di raffreddamento che saranno sicuramente economici ma per niente ecologici.

### Perché si chiama Cloud?

Perché quando si disegnava lo schema client-server tutto ciò che stava fra i due veniva rappresentato come una nuvoletta per semplicità.

---

## 3° LEZIONE: Introduzione a IaaS con video

### Virtualizzazione

Creazione di risorse virtuali come server, desktop, sistemi operativi, file, spazio di archiviazione o network. Serve per **gestire il carico di lavoro** rendendo la computazione più scalabile. La forma più comune di virtualizzazione è quella del SO.

La virtualizzazione è un livello di astrazione, il **SO non è più legato al server/pc dove gira**, il SO è astratto dall'Hardware, non è installato direttamente sull'Hardware.

### Server Virtualization

È un **livello di virtualizzazione** tra il server fisico e il sistema operativo che useresti normalmente. Comprende le macchine virtuali dove installi effettivamente il sistema operativo e le applicazioni. Come il SO permette a tutti i programmi e software di funzionare così il livello di virtualizzazione permette di avere tante macchine virtuali.

### Hypervisor

È una funzione che **crea il livello di virtualizzazione** (che permette la virtualizzazione del server) e **contiene il Virtual Machine Manager (VMM)** per la gestione delle stesse.

Il sistema operativo non è installato direttamente sulla macchina ma è astratto dall'hardware, non è connesso direttamente al laptop, pc o server, c'è infatti un livello tra il server fisico e il sistema operativo chiamato **virtualization layer** (per esempio ESXi o ESX di VMWare). Sopra ci sono le virtual machine dove è possibile installare SO differenti.

*Esempi di hypervisor: Vmware Workstation, Virtual Servr, Hyper-V , Fusion, XenServer*

**Server virtualization**: più virtual machine usano lo stesso hardware. Permette all'OS di girare su un hypervisor.

**Virtual Host**: server fisico su cui girano le altre Virtual Machine

**Virtual Machine:** ogni SO ospite del Virtual Host

**Tipi di Hypervisor:**

- Tipo 1: caricati direttamente sull'hardware

  *Esempi: Hyper-V, ESX, ESXi, XenServer*

- Tipo 2: caricati sull'OS dell'hardware come una normale applicazione

  *Esempi: Workstation, Virtual Server, Fusion, Oracle VirtualBox*

Il tipo 1 è più performante, il tipo 2 non riesce a mantenere più VM contemporaneamente però più adatto per hardware meno potenti, come un laptop o desktop.

## Amazon

Venne fondata con il nome di "Cadabra" nel 1994 da Jeff Bezos, era una biblioteca online. Il business plan iniziale era di non aspettarsi profitto per i seguenti 4-5 anni, il primo profitto lo hanno avuto nel 2001.

Alcune statistiche:

- **300 milioni di utenti** (Dicembre 2017)
- 2 milioni di venditori Amazon (Gennaio 2018)
- 560'000 dipendenti (Gennaio 2018)
- 3'724 miliardi di dollari di entrate all'anno
- Detiene il **47.8% del mercato Cloud** (Microsoft 15%, Alibaba 7%, Google 4%)

## Amazon Elastic Compute Cloud (EC2) (IaaS) (Freemium)

È difficile prevedere quale è il numero di **server** di cui abbiamo bisogno per un'applicazione, inoltre, è un processo molto lungo e costoso. Per questo Amazon EC2 mette a disposizione dei server virtuali chiamati "istanze" in modo semplice, veloce ed economico, che si possono configurare (ad esempio con la scelta di un template Windows/Linux), si può scegliere tramite **Amazon Web Services (AWS)** management console (o librerie SDK) la potenza di calcolo delle istanze in base alle proprie necessità. Le istanze possono essere CPU, memoria, storage e GPU.

Amazon EC2 ci offre anche un buon livello di sicurezza grazie al VPC (Virtual Private Cloud) che rende la connessione più sicura all'interno del Cloud (per altre info https://aws.amazon.com/it/vpc/).

**Amazon Elastic Block Store (EBS)** ci garantisce uno storage persistente per l'uso su EC2, ci offre disponibilità e durabilità dei nostri dati. È stato pensato per le applicazioni che necessitano di big data analytics, stream processing o data warehousing.

**Autoscaling:** Amazon permette di definire delle metriche per decidere come e quando "scalare" e ridimensione l'infrastruttura per poter pagare solo per quello che effettivamente serve. Scala automaticamente all'occorrenza.

**Prezzi:**

- On demand: paghi solo per quello che utilizzi (ad esempio prezzo per GB)
- Reserved instances: pagamento mensile (di solito si risparmia rispetto a on demand), si usa quando sappiamo già quali e quante macchine mi serviranno
- Spot Instances: sono istanze non utilizzate che Amazon mette in vendita ogni 5 minuti all'asta, di solito costano il 50% in meno.
- Dedicated hosts: server fisico con istanze EC2 totalmente dedicato
- AWS Free Tier: livello freemium con limitazioni

Amazon detiene quasi il 50% del mercato Cloud.

## Amazon Simple Storage Server (S3) (IaaS) (Freemium)

Per gestire l'archiviazione dei dati della propria applicazione Amazon S3 mette a disposizione **memoria** dove poter depositare i propri dati senza il bisogno di acquistare fisicamente le unità di archiviazione (che non sappiamo quante dovranno essere). L'interfaccia è estremamente **semplice**: drag and drop in un secchiello ("bucket").
Amazon S3 per assicurarci di non perdere i nostri dati esegue 3 backup su 3 unità di memoria diverse e consente di mantenere tutte le vecchie versioni dei file in modo da poterli recuperare se cancellati erroneamente.
Si paga solo quello che si utilizza.
Ci sono diverse **classi di memorizzazione**:

- s3 standard: oggetti comuni con lettura continua
- s3 standard infrequent access: accesso poco frequente a questi dati
- amazon glacier: oggetti a lungo termine come i backup dei database

È possibile creare regole per far cambiare classe ai file automaticamente.
Gestione di accesso ai dati e sicurezza.
Crittografia sui dati.
Scalabilità automatica.

**Prezzi:** si paga per GB e in base alla classe di memorizzazione, ma anche qui si ha un AWS Free Usage Tier con 5GB gratuiti.

## Dropbox (IaaS) (Freemium)

È un servizio di file hosting che offre memorizzazione Cloud, sincrozzazione dei file e strumenti di collaborazione (Dropbox Paper). Dropbox è stato fondato nel **2007** da due studenti del MIT, avevano pensato di offrire **gratuitamente** spazio di archiviazione con limitazioni, e chi voleva poteva acquistare l'account premium per avere più capacità di memorizzazione. A Marzo 2016 Dropbox aveva **500 milioni** di utenti.

Per i primi 8 anni Dropbox ha sempre usato i server di Amazon S3, era il loro più grande cliente, poi tra il 2014 il 2016 ha costruito le proprie macchine poiché voleva poter controllare tutto al 100%. Al momento del passaggio da Amazon S3 ai propri server hanno avuto molti problemi:

- Dovevano cercare di trasferire tutto prima del rinnovo del contratto con Amazon
- Gli utenti non si sarebbero dovuti accorgere di niente, non avrebbero dovuto avere interruzioni di servizio o perdite di dati nonostante ci volesse 1 giorno intero per trasferire 4 petabyte.
- Il nuovo software "Magic Pocket" non era totalmente compatibile con il nuovo hardware, hanno dovuto fare delle modifiche rinominandolo "Mozilla Rust"
- Incidenti stradali per i camion che trasportavano i server hardware

**Come sta andando adesso Dropbox?** (dati Novembre 2019)
C'è molta competizione, è difficile farsi spazio nel mercato Cloud, al momento sembra che la situazione sia positiva.

**SLA di AWS** (https://aws.amazon.com/agreement)
Cosa ci garantisce Amazon? Niente.
The service Offering Are Provided "As is".
Non dà garanzia riguardo ai servizi offerti, garanzia contrattuale pari a zero.

**SLA di Dropbox**
Uguale, nessun diritto se ci sono perdite di business, profitto o dati.

---

**Come suddividere un'applicazione three-tier su un Cloud ibrido?**
Web server (presentation tier) – App server (business logic tier) – DB server (data tier)

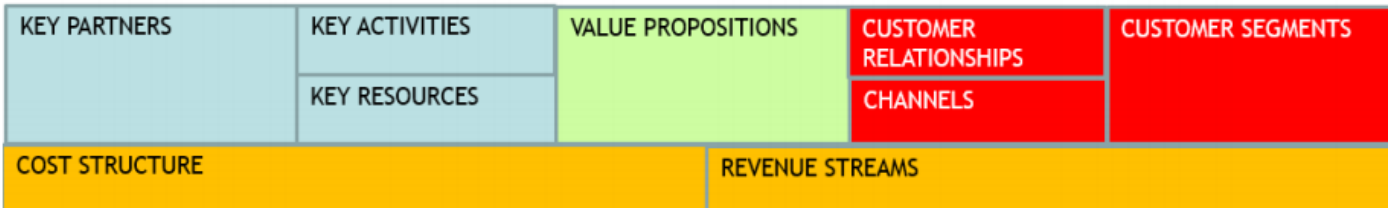
In generale qualsiasi soluzione va bene, solitamente si mettono almeno 2 parti sul Cloud pubblico, in alcuni casi anche tutto. Più raramente viene usato solo il Cloud privato.

Più info https://www.slideshare.net/AmazonWebServices/aws-reinvent-2016-hybrid-architecture-design-connecting-your-onpremises-workloads-to-the-cloud-gpsisv4?from_action=save

---

# 4° LEZIONE: Business Model

**Cos'è un Business Model**
Descrive come un'azienda crea, consegna e cattura valore.
Ma come si fa a dire queste cose di un'azienda?
Attraverso i **Business Model Canvas,** possiamo utilizzarli in molti modi, possiamo prendere un canvas vuoto e un'azienda esistente e provare e riempirlo oppure costruirne uno per una startup.



*Business Model Canvas*

**Customer Segment:** il target di mercato, quali tipi di clienti vogliamo attirare.

**Value Proposition:** Qual è il valore del nostro servizio? Cosa offriamo al cliente? Per cosa dovrebbero venire da noi?

**Channels:** I canali con cui il servizio arriva ai clienti (Shop online, negozio fisico ecc.)

**Customer Relationship:** Sembra opzionale ma in realtà il rapporto col cliente è molto importante, ci sono aziende che si focalizzano molto su questo aspetto sfruttando la "fidelizzazione" attraverso dei punti per il cliente.

**Revenue Streams:** I flussi da cui arrivano le entrate, ne descrive anche il tipo.

**Key Resources:** risorse chiave, le risorse che eroghiamo e che sono fondamentali.

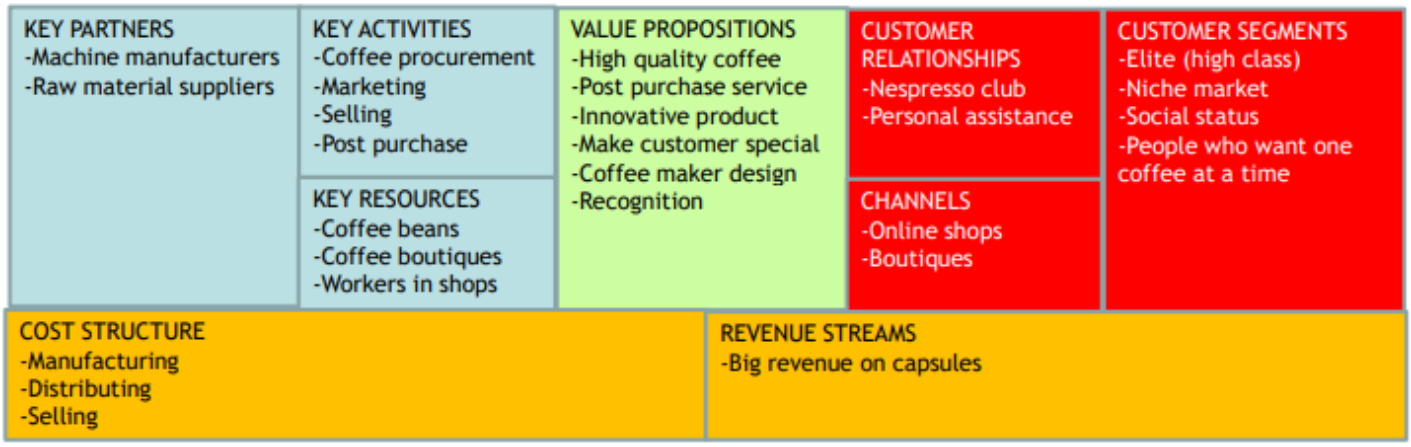
**Key Activites:** L'attività principale della nostra azienda.

**Key Partnership:** È bene cercare dei partner, come Dropbox aveva Amazon, però se il partner fallisce rischia di cadere tutto.

**Cost structures:** Indica quali sono le uscite.

## ESEMPI DI BUSINESS MODEL

### Nespresso




Nel 1976 Nestlè dominava il market con Nescafè il caffè solubile più venduto.
Le macchinette Nespresso furono ideate nel 1976 quando nessun altro le faceva, erano pronte ma non riuscivano ad entrare nel mercato.
Nel 1988 il nuovo CEO cambiò il business model, cambiò il **Customer Segment** poichè le macchinette non dovevano avere lo stesso mercato del Nescafè ma dovevano attrarre impiegati di alto livello e in generale famiglie benestanti.

CONSEGUENZE:
Pubblicità: L'idea è che sei fig* se bevi Nescafè (vedi George Clooney)
Canali di acquisto: online shop, Club Nespresso, oppure vai nelle boutique solo di capsule della Nespresso, solitamente nel centro della città vicino a Armani, Gucci ecc.

Club Nespresso: una delle cose più importanti di un'azienda è sapere il comportamento dei propri clienti per riuscire a tenerli. Analisi dei dati, Nespresso traccia il 100% delle attività dei clienti poichè ci sono solo 2 canali. Ogni volta che fai l'ordine col club nespresso vedono tutto quello che hai comprato in modo da poterti offrire tramite newsletter e news i prodotti più adatti.

CRITICHE:
Dal punto di vista della sostenibilità ambientale Nespresso produce una grandissima quantità di materiale difficilmente riciclabile che attualmente viene smaltito male.
Come anche la commercializzazione del latte in polvere, grande successo di Nestlè, ma ha avuto effetti secondari: nei paesi in via di sviluppo l'acqua con cui veniva sciolto il latte era contaminata e molti bambini sono morti. Hanno conquistato comunque una fetta di mercato importante.

### Zara

Zara ha deciso di produrre abiti in base a cosa la gente acquista, non è il cliente a seguire la moda ma è Zara a seguire il cliente. Rinnova continuamente. **No warehousing: no magazzino, viene spedito quello che serve ad ogni negozio.**

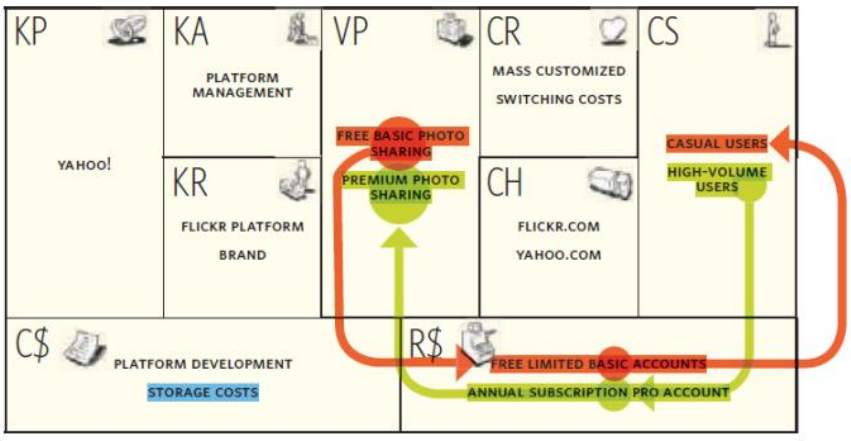
ESEMPIO TICKETRESTAURANT MANCA

## FREEMIUM COME BUSINESS MODEL

Consiste in una suddivisione dei servizi: una parte Free che comprende i servizi cosiddetti “di base” e una parte Premiuim a pagamento che invece offre servizi aggiuntivi e/o migliorati.

Chi adotta il modello Freemium deve porre molta attenzione al costo che comportano gli utenti “free” e alla percentuale di conversione degli utenti che decidono di passare a Premium.
I prossimi esempi metteranno in luce quali sono i modi in cui guadagnano le aziende che seguono questo modello.

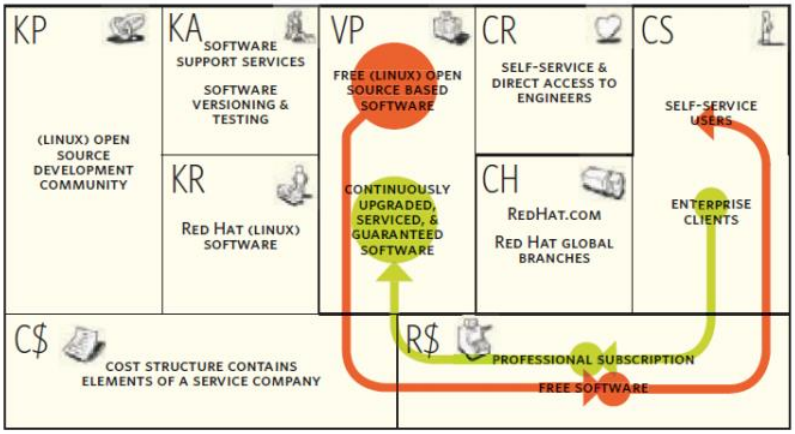
### Flickr




Se paghi puoi avere premium photo sharing, non puoi solo mettere le foto nelle cartelle ma anche avere dei servizi addizionali.

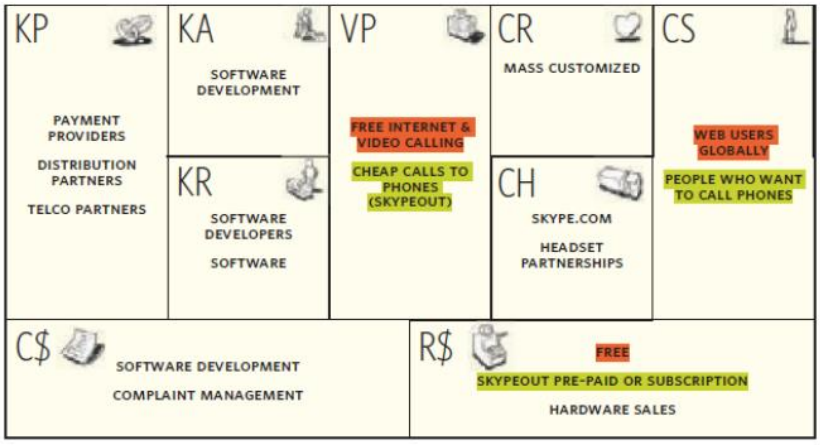
È durato 3 anni, dal 2002 al 2005, acquistato da Yahoo per 25 milioni di dollari che a sua volta è stato acquistato da Verizon per 4 miliardi nel 2017 dopo aver rifiutato nel 2009 44 miliardi da Microsoft.

## RedHat




Chi usava un Opensource e offriva un servizio aveva paura che l'Opensource fallisse, RedHat si metteva nel mezzo e garantiva stabilità, veniva pagato e manteneva l'Opensource sempre aggiornato.
Venduto per 34 miliardi di dollari a IBM.
(Opensoruce non significa che il software sia gratuito ma che è possibile modificarne il sorgente)

## Skype




Tutti gli utenti web possono fare chiamate gratuite e anche videochiamate. Pagando si ha la possibilità di chiamare anche i cellulari a prezzi competitivi.
Nato nel 2003, comprato da Ebay nel 2005 e poi nel 2011 da Microsoft.

## Dropbox

Il servizio Freemium di Dropbox è così suddiviso:

**Per i privati**
Free: 2 GB
Plus: 1 TB per 9,99€ al mese
Professional: 2 TB per 19,99€ al mese

**Per le aziende**
Standard: 3TB per 10,00€ al mese
Adavanced: no limit per 15,00€ al mese
Enterprise: <contact us>

Dropbox ha **500 milioni** di utenti di cui **12 milioni** paganti.

**Netflix**
Netflix usa attualmente il **15% del traffico internet** in download nel mondo. Netflix paga i diritti dei film e guadagna per il "noleggio" verso gli utenti.

Come già detto precedentemente Dropbox ha iniziato appoggiandosi ad Amazon AWS e poi si è distaccato, Netflix invece, nonostante avesse costruito già i data center, ha iniziato la partnership con Amazon. L'arrivo dei Container ha creato dei problemi al reparto di sviluppo di Netflix e hanno dovuto mettere da parte l'idea di usare i propri data center chiudendoli e continuando ad appoggiarsi ad Amazon.

---

## 5° LEZIONE: Ancora Business Model

Esistono diversi tipi di strategie per i Business Model, al giorno d'oggi la cosa più importante è mettere al centro il cliente e il rapporto che si crea con esso. Bisogna chiedersi di che cosa ha bisogno, come preferisce essere contattato, a quale tipo di valore sono interessati i clienti e per cosa sono diposti a pagare.

Per **innovare il business model si può...**

- Partire dalle risorse che ha già l'azienda **(Resource Driven)**
  Come ad esempio **Amazon Fulfillment** che consiste nell'offrire a terze parti la possibilità di vendere e far arrivare i propri beni ai clienti, prima di iniziare con questa pratica Amazon aveva già tutto pronto (shop online, catena organizzativa).

- Basarsi sull'offerta (**Offer Driven)**
  Ad esempio: **Cemex** che in un momento in cui il cemento era garantito in 48h di tempo è riuscita a farlo in 4h.

- Basarsi sui clienti **(Customer Driven)**
  **23andMe** offriva test per dna ai singoli clienti privati

- Basarsi sull'aspetto finanziario (**Finance Driven)**
  Per esempio **Xerox** che introdusse un modello in cui il costo delle fotocopie veniva offerto in leasing e venivano inoltre garantite 2000 copie gratuite

- Epicentri multipli (Multiple Epicenter Driven)

Sono tutti modi in cui si può ridefinire un Business Model innovativo partendo da angolature distinte.
Le idee delle aziende che hanno fatto più successo (Google, Spotify, Dropbox ecc.) si sono basate su domande del tipo "Cosa succederebbe se offrissi musica gratis a tutti?" e sono riuscite a trovare una strategia vincente.

### Statistiche startup

**Airbnb**: valore di 35 miliardi di dollari. 2 Milioni di persone a notte dormono in una struttura prenotata su Airbnb.

**Epic Games**: Fortnite ha 250 Milioni di giocatori e guadagna centinaia di milioni di dollari al mese grazie alla vendita delle skin.

**Facebook**: 2.45 miliardi di persone ogni mese.

**Instagram**: acquistato da Facebook dopo 2 anni per un miliardo di dollari.

**Il 90% delle startup fallisce** per 4 motivi principali: incompetenza, incapacità di gestire il budget, mancanza di esperienza, problemi personali.

### Spotify

Il nome è nato da un misunderstanding.
Gli stakeholder di spotify sono:

- Freeusers
- Royalty holder: quelli che vengono pagati ogni volta che viene riprodotto un brano (SIAE in Italia)
- Subscriber: utenti premium
- Advertiser: coloro che pagano Spotify per venire sponsorizzati

Gli utenti free e gli utenti abbonati devono bilanciarsi, ma quando iniziamo con un modello Freemium non abbiamo idea di quale sarà la percentuale tra utenti free e premium, è difficile da stabilire. Spotify per pagare meno di royalty ha ideato una strategia che permette di scegliere dove reperire il brano in base alla localizzazione dell'utente poiché i costi variano in base al paese da dove l'utente richiede l'ascolto.
Se gli utenti Premium ascoltano troppo rischiamo di andare in perdita poiché è più quello che paghiamo per le royalties che quello che riceviamo dagli utenti.

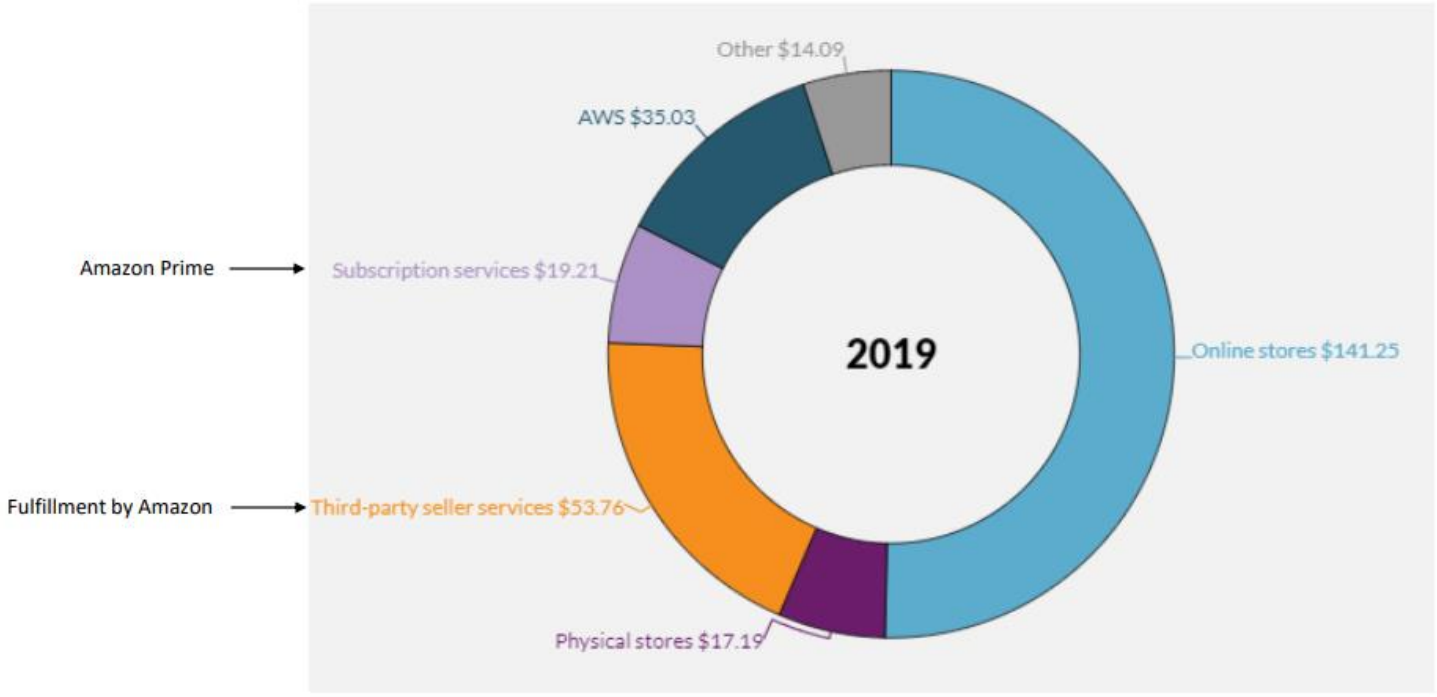
Statistiche:
**217 milioni di utenti attivi**
**100 milioni di questi sono membri Premium**

---

## 6° LEZIONE: Amazon e Google

### Le entrate di Amazon (Amazon revenue)

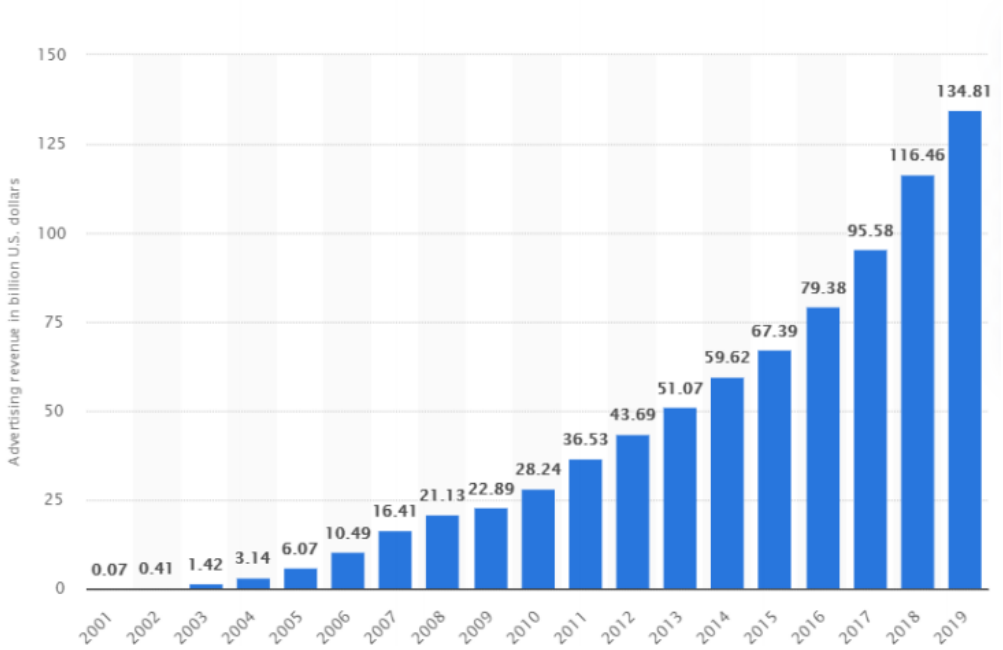
Nel 2019 le entrate sono state pari a **280 miliardi di dollari**




1. Store Online
2. Venditore per altri (Amazon Fulfillment)
3. Amazon Web Services (Cloud)

## Le entrate di Google

Google è un servizio gratuito, come fa a guadagnare? Grazie al **Customized Advertising,** la nostra "attenzione" verso le pubblicità permette a Google di farsi pagare.




*Crescita del guadagno di Google grazie al Customized Advertising*

### 5 Riflessioni sui motori di ricerca

1. Per la prima volta nella storia tutti posiamo facilmente generare informazioni accessibili a tutti. Ci sono più di **4 miliardi di utenti su Internet e quasi 2 miliardi di siti Web**. Quasi tutte le ricerche di informazioni passano attraverso un unico punto di accesso: Google. Azienda che domina il mondo dei motori di ricerca con più di **3.5 miliardi di ricerche** giornaliere. Durante una ricerca **i primi 3 risultati ottengono il 75% dei click** e solitamente se non troviamo niente nei primi 3 risultati cambiamo query di ricerca. I siti Web cercano di comparire nei primi 3 risultati perché altrimenti non vengono cliccati. Dobbiamo sempre tenere a mente che la prima pagina dei risultati di Google è una parte molto piccola di tutte le informazioni esistenti.

2. **Google non controlla le fonti e né se un'informazione è vera o no**. Se facciamo una ricerca oggi otteniamo, nei primi 3 risultati, determinati siti web, se riprovassimo a cercare la stessa cosa dopo un mese potremmo avere dei risultati diversi.

3. Google può **tenere traccia** delle nostre attività poiché possiede molti servizi diversi: Youtube, Google Maps, Google Chrome, Google Shopping, Google Calendar, Waze, Google Photos ecc.

4. Cosa succederebbe se Google manipolasse i risultati delle ricerche? È stato fatto un esperimento ed è stato visto che è possibile spostare del **20%** le preferenze di voto degli elettori indecisi mostrando degli opportuni risultati di ricerca in modo trasparente per gli utenti.

5. Quali sono gli effetti di Google per la nostra memoria?
   Se sappiamo che possiamo avere accesso a certe informazioni in qualsiasi momento non le ricordiamo, ma ci limitiamo a ricordare dove possiamo cercarle o trovarle. Questa cosa si chiama memoria transattiva, è sempre esistita, cioè sappiamo che ci sono cose che non ricordiamo ma sappiamo che altre persone invece le sanno. **Internet diventa una memoria transattiva.**

---

## 7° LEZIONE: PaaS

### Platform as a Service (PaaS)

I PaaS aumentano ancora di più i servizi offerti dallo IaaS offrendoci un intero ambiente di sviluppo e gestione dell'applicazione.
Benefici del PaaS:
- Può creare prototipi in **pochi minuti**
- Può creare nuove versioni o deploy del codice più **rapidamente**
- **Self-assemble service**, quindi le varie parti dell'applicazione si assemblano da sole
- **Scala automaticamente**
- Non bisogna preoccuparci della sicurezza
- Il PaaS si occupa delle strategia di Backup e Recovery

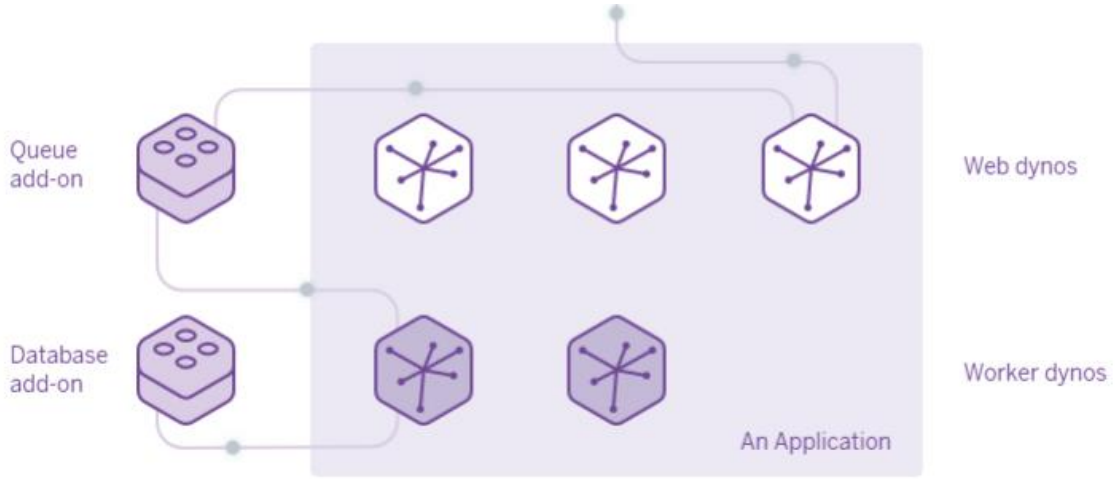
Il PaaS guadagna meno rispetto a IaaS(2°) e SaaS (1°) ma comunque vale molto, circa 19 Miliardi di dollari.

### Heroku (Paas) (Freemium)

È un PaaS. Piattaforma Cloud che fornice una serie di servizi integrati e un intero sistema che ci permette non **solo di fare il deployment delle applicazioni ma anche di completarle, mandarle in esecuzione e gestirle**. Una delle caratteristiche di Heroku è che è basata su un sistema di **Container.**

È nato nel 2007 poi nel 2010 è cresciuto ed è stato acquistato da Salesforce (che si occupa di Saas soprattutto). Comprende diversi linguaggi per lo sviluppo del codice (Node, Java, Php, Python, Go, Scala).
I Container di Heroku si chiamano **Heroku Dynos.**
Gli Heroku Dynos sono una virtualizzazione, sono ambienti Linux abbastanza leggeri, isolati che permettono di "incapsulare" un'applicazione e tutte le sue dipendenze, librerie ecc. in un grande "container". Questo ci permette di **spostare la nostra applicazione** senza dover dividere le varie parti, una volta giunto a destinazione viene prelevata l'immagine del container e ricostruita l'applicazione.
I Dynos sono utilizzati anche per lo **scaling**, potrebbe volerci un solo Dynos inizialmente ma poi l'applicazione ingrandendosi avrà bisogno di più spazio e vengono semplicemente aggiunti più container la cui **gestione è molto semplice**, non dobbiamo più preoccuparci dell'aspetto scalabilità.

**Web Dynos**

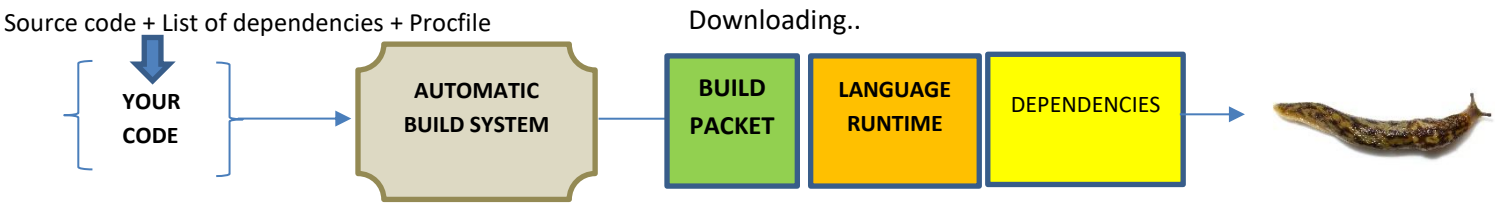



L'applicazione riceve la richiesta, la richiesta viene inviata a uno degli **Web Dynos**, viene analizzata e messa in una coda asincrona (che è ottima per la **scalabilità orizzontale**), poi il **Worker Dyno** prende la richiesta e la gestisce. Se si vuole si può far persistere i risultati **in un database che è sempre parte di Heroku**.
In questo caso la **scalabilità permette di aumentare il numero di Web Dynos** in modo da poter gestire un elevato numero di richieste ricevute contemporaneamente.

**Buildtime**
Heroku ha bisogno di tre cose per costruire un'applicazione:

- Codice sorgente
- Lista di dipendenze (Quali sono le cose di cui ha bisogno per funzionare)
- Un "Procfile" cioè un file di testo che indica qual è il comando per far partire il codice

Queste 3 cose fanno parte del Buildtime, cioè la fase di costruzione, una volta date in input parte il **sistema automatico di built**: dopo aver ricevuto il codice scarica il **Build Packet** (linguaggio, dipendenze, librerie ecc.), produce uno **Slug** e lo mette in esecuzione in un Dyno. Il componente finale per eseguire l'applicazione è il sistema operativo (Ubuntu) che è aggiunto da Heroku e che si chiama "**stack**".



**Runtime**
Una volta preparato tutto Heroku crea uno o più Dynos con lo stack e lo slug e fa partire l'applicazione, a quel punto ci abilita a modificare l'app con i vari tipi di Dyno (free, standard, performance, web, workers)

**Add-ons**
Heroku ha più di **150 Add-ons** (servizi esterni) da aggiungere alla propria applicazione per estenderne le funzionalità come servizio di autenticazione, log, monitorning, data stores ecc.
Tutto questo attrae l'utente perché ha la possibilità di integrare nuove funzionalità in un tempo molto breve e ci facilitano lo sviluppo poiché non dobbiamo costruire da zero ciò che vogliamo. L'aspetto negativo è che questo ci lega all'utilizzo della piattaforma e instaura una forma di **vendor lock-in** rendendo la nostra applicazione più **difficilmente portabile**. Infatti se volessimo spostare la nostra applicazione su un nuovo servizio cloud saremo costretti a rivedere tutto il codice perché gli Add-ons non funzionerebbero.

## Microsoft Azure (PaaS e anche IaaS)

Cosa offre Azure:

- Scalabilità **verticale**
- Supporta **diversi linguaggi** e ha un'ampia varietà di strumenti per sviluppatori
- SQL database
- Machine Learning per analisi predittive
- Servizi media per supportare video live e on demand
- Meccanismi di sicurezza
- Data storage (IaaS*)

*Ai servizi PaaS è spesso richiesto di offrire anche IaaS.

## Cloud Foundry (PaaS)

Concepita da VMware nel 2009, è un Open source.
Nel 2013 trasferita a Pivotal Software e nel 2015 a Cloud Foundry Foundation.
La piattaform Cloud Foundry si presenta come un insieme di servizi distribuiti, alcuni sono:

- **Cloud Foundry Bosh** che permette di utilizzare tutta la struttura sottostante, cioè i servizi IaaS offerti da provider diversi (compreso Amazon WS).
- L'**Health Manager** monitora la qualità del servizio e ci può aiutare per capire come sta crescendo e funzionando la nostra applicazione. Riporta eventuali discrepanze al controllore.
- **Dynamic Router** ha il compito di instradare tutto il traffico che arriva dalla rete esterna
- Il **Cloud Controller** mantiene e gestisce tuti i sistemi di controllo della piattaforma compresa l'interfaccia con i clienti.
- **User Authorization and Authentication** gestisce l'autenticazione e autorizzazione
- Il **Service Brokers** permette di accedere ad altri servizi esterni o nativi.
- Messagging (NATS) Fast internal messaging bus with pub-sub mechanism
- User Provided Service Instances Store metadata in service broker (to permit connecting to services not managed by CF)

La dipendenza debole tra i servizi consente di resistere meglio a possibili guasti.

**Caso di studio: CSAA Insurance**
Assicurazioni Auto in America.
Erano in una situazione poco stabile, non avevano automazione, avevano un cosiddetto "debito tecnico" cioè avevano molte tecnologie obsolete, troppi "**silos**" cioè blocchi grandi di applicazioni che invece sarebbero dovute stare separate e poca capacità di fare analisi sui dati.
Hanno deciso poi di avvalersi del servizio fornito da **Pivotal Cloud Foundry** e la situazione funzionò così passarano ad usarlo nella produzione vera e propria e sono passati al Cloud pubblico.
Hanno fatto 3 passaggi in pochi giorni quando invece ci sarebbero voluti dei mesi, hanno così ottenuto un'automazione robusta per poi decidersi di passare definitivamente ad un **Cloud Privato**.
I risultati sono stati un aumento del **200% di produttività** e un aumento di deployment frequency del **1400%.**

Non tutti saranno stati contenti del cambiamento e qualsiasi errore o problematica futura attribuiranno la colpa al Cloud, ma è un "rischio" da correre per migliorare l'azienda.

**Caso di studio: Australian Government**
Aveva da gestire più di 1500 siti web di agenzie locali e l'obiettivo era soprattutto diminuire la burocrazia e riuscire a compattare e coordinare questi servizi sparsi. Si voleva anche ridurre il tempo di deployment delle applicazioni e ridurre il downtime a zero.
Hanno creato quindi un'agenzia digitale per semplificare l'interazione con gli utenti per i servizi governativi e poi hanno progettato il passaggio su **Cloud Foundry** grazie ad un team cross-functional (cioè ogni componente del gruppo è esperto in qualcosa di diverso dagli altri).

Quali sono stati i risultati?
- Le prenotazioni di appuntamenti per i cittadini sono passate da **un'attesa di 92 min a 2 minuti**
- Semplificata la richiesta per i permessi di soggiorno
- Digital Marketplace per appalti e gare
- Possibilità di avere una dashboard, cioè poter visualizzare le performance di tutti i servizi

La scelta che hanno fatto è stata quella di iniziare a **modificare i servizi più utilizzati** anziché partire da quelli meno utilizzati perché così l'impatto è stato maggiore, hanno organizzato il lavoro in team piccoli (circa 8 persone) con specialisti diversi e hanno aumentato il numero di test sulle funzionalità.

**Open Shift (PaaS)**
**(RedHat)**
Il video mostra come si possono "assemblare" varie parti per costruire una piccola applicazione che simuli un sistema di voto. Il video commerciale sembra molto più semplice di questo, in realtà è abbastanza macchinoso collegare i vari pezzi di funzionalità e linguaggi diversi tra loro.

**Quale PaaS usare? Find your PaaS**
In un mercato così variegato è difficile fare la scelta giusta, fortunatamente ci sono dei siti a disposizione che ci permettono di trovare il PaaS adatto alle nostre esigenze semplicemente facendoci scegliere le funzionalità di cui abbiamo bisogno.

---

9° LEZIONE: Microservizi

## MICROSERVIZI

Tecnologia molto importante nello sviluppo del software, soprattutto utilizzando Cloud.
Da qualche anno si parla in modo insistente dei microservizi, è una moda o una realtà? Sono metodologiche davvero importanti per la gestione del software?

Amazon, Netflix, Spotify, Google, Ebay, Facebook, Twitter, LinkedIn usano i microservizi. Prima conclusione: è una buzzword ma non è staccata dalla realtà perché ci sono una serie di servizi informatici interamente basati sui microservizi.

## Perché si usano?

Per due motivi principali:

- Riducono il **Lead Time**

  Il Lead Time è il tempo che passa da un tempo A ad un tempo B. Tempo A: il progettista dell'applicazione si immagina su una lavagna una nuova feature o aggiornamento. Tempo B: momento in cui il primo utente fa un click sulla nuova feature o aggiorna l'applicazione. Questo è importante perché la fase di rebuilt e redeployment è una fase molto costosa, aggiungere una feature significa rifare il rebuilt e il redeployment, grazie ai microservizi viene fatta velocemente.

- La necessità di **scalare** (orizzontalmente)

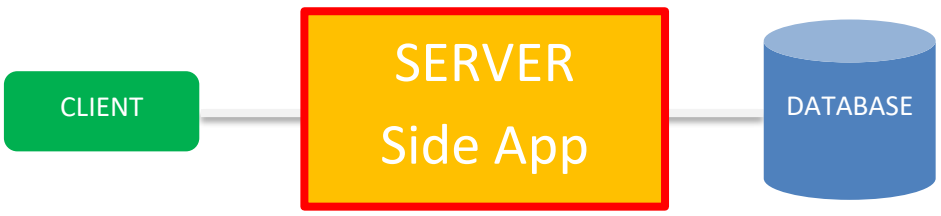
  Cioè quindi aumentare la capacità di risposta dell'applicazione. Un modo è creare delle repliche della nostra applicazione per rispondere al doppio dei clienti. Ma se la criticità dell'aumento del numero di richieste non è per tutta l'app ma solo per alcune funzionalità non è necessario replicare interamente tutto, i microservizi ci aiutano a scalare soltanto quella parte. La capacità di scalabilità replicando un singolo microservizio è **orizzontale**, mentre replicare tutta l'app è **verticale**.

  **Esempio:**
  Spotify ha 75 Milioni di utenti attivi ogni mese
  La media di una sessione è 23 minuti
  Il numero delle playlist create dagli utenti è di 2 Miliardi
  73 milioni di playlist di Spotify

  Quindi ha tantissime richieste e ha bisogno di scalare solo alcune parti.

## Monoliti

Prima dei microservizi si usavano i cosiddetti Monoliti, cioè grandi blocchi di applicazioni, in particolare la tipica interfaccia era:




Serve side App è un monolite (singola logica eseguibile)

Questo portava a molti lati negativi:

- Fare cambiamenti a piccole parti dell'applicazione richiedevano rebuilding e redeploying di **tutto il monolite**
- Scalare significava **replicare tutta l'applicazione**

## Essenza dei microservizi

1. **Service-orientation**
   La nostra applicazione consiste in una serie di servizi, ognuno su un container diverso, che comunicano fra loro con meccanismi d**i comunicazione lightweight, leggeri**, ovvero attraverso protocolli RESTish, richieste-risposte di 2 tipi:
   - HTTP con API
   - dumb message bus, che non garantisce proprietà aggiuntive, code asincrone
   Inoltre supportano diversi linguaggi (poliglottismo)

   Cosa non si usa più:
   ~~ESBs~~ (smart endpoints and dumb pipes) nella prima versione dei microservizi venivano usati questi ESBS, connettori ricchi di funzionalità che permettevano ai servizi di collaborare tra loro, non erano code asincrone ma facevano tante cose. **Parte della logica dell'app finiva per essere messa in ESBs**

   ~~Standard WS:~~ standard della sicurezza, sono tantissimi, i microservizi hanno detto no basta andiamo col minimo di standard richiesti.

2. **Organizzare i servizi intorno a delle business capabilities**
   Legge di Conway 1968: le organizzazioni che progettano dei sistemi finiscono con l'essere vincolati nel produrre delle progettazioni che riflettono la struttura delle proprie organizzazioni.
   Esempio: se nella nostra azienda abbiamo 3 divisioni uno che si occupa del client, uno server uno db molto probabilmente il nostro sistema informatica finirà per avere tre pezzi. I microservizi invece dicono di passare a un modello totalmente diverso, avere dei **cross-functional teams** dove in ogni gruppo c'è una persona esperta di cose diverse.

3. **Decentralizzazione della gestione dei dati**
   Permettere a ogni servizio di avere e **gestire il proprio Database**, non dobbiamo mai avere un unico DB, proprio per eliminare quelle "corde" tra team diversi, ognuno gestisce il proprio db e non deve chiedere agli altri.
   Le transazioni distribuite tra processi indipendenti comunicanti sono una cosa molto complessa, se devo fare 4 operazioni su 4 db diversi e alla 4° ho un fallimento dovrei fare un rollback su servizi separati, un'operazione molto costosa e molto complessa e magari irreversibile.
   I microservizi invece di eseguire queste transazioni distribuite ma utilizzano la tecnica dell'**eventual consistency**, invece di mantenere i database sempre consistenti accettare che per un periodo non lo siano ma poi lo saranno. Se ho un'operazione su 2 db aggiorno il mio db e l'altro se non posso aggiornarlo ora non importa, **lo aggiornerò dopo**, andiamo avanti. Ci sarà un incaricato che cercherà di portare a termine l'aggiornamento. **Sacrifichiamo la consistenza per migliorare l'efficienza**.

4. **Independently deployable service**
   Ogni servizio è indipendente, può essere lanciato senza lanciare gli altri servizi, si fa il rebuilt solo di una parte non di tutta l'applicazione.

5. **Scalabilità orizzontale**
   Vorremmo riuscire ad ampliare solo quella di cui ho bisogno, replicare più istanze del servizio critico.

6. **Resistenza ai fallimenti**

Tondini: microservizi
Puntini: messaggi
Flash: errori

Il servizio più a sinistra è l'API Gateway, se c'è un fallimento quindi potrebbe causare un fallimento a cascata che è arrivato a rendere inaccessibile l'applicazione.
Con i microservizi bisogna accettare che ci saranno dei fallimenti, il problema da risolvere è che chi ha fatto la richiesta deve **rispondere al fallimento nel modo migliore possibile**.
Il problema è che se ci sono chiamate sincrone:
30 chiamate sincrone tra servizi e ogni servizio sta su 99.99%
La probabilità che tutti non falliscano scende a 99.7% e in un mese abbiamo circa 2 ore di downtime che è tanto.
Conclusione: i microservizi cercano di progettare tenendo conto che il fallimento avverrà, inserendo circuit breaker **per evitare il fallimento a cascata**. Viene messo un proxy che spezza il circuito, se prima C chiamava S ora C chiama proxy che la manda a S. Fa da intermediario, se dovessero aumentare i tempi di risposta o non riuscisse a ad ottenerla il circuit breaker dà comunque una risposta anche se S non gliel'ha data.

Il **servizio di ricerca** di Spotify è quello che ha avuto più problemi, quindi cosa ha fatto spotify? Se un utente cerca qualcosa e nel caso di fallimento semplicemente non riceve niente, viene restituita la cella bianca, l'utente rimette la query e ha successo. Non è un fallimento se l'utente non se ne accorge.

**Chaos Monkey**: Netflix
Test coraggiosi, strumento che viene usato in fase di sviluppo pe abbattere in maniere pseudocasuale dei container, delle istanze ecc.. in fase di produzione per vedere se tutto va bene.

**I microservizi si chiamano micro perché sono piccoli?**
La dimensione non è importante, in realtà non è importante che siano piccoli in linee di codice, è più importante la dimensione del team, è piccolo il team che sta progettando il microservizio.
Ogni microservizio è gestito singolarmente da un team separato.

**Perché è così importante?**
Una "vecchia" struttura è composta da silos collegati dalle corde, ogni corda rappresenta un ritardo nel nostro processo. Se il team 1 ha bisogno di una modifica dove sta lavorando il team 2, viene mandata una richiesta che valuteranno e diranno se si può fare. Nello sviluppo di applicazioni avere team che devono comunicare tra loro introduce dei ritardi in ordine di giorni e settimane.
Con i microservizi andiamo ad evitare proprio questa situazione.

**Esempio di Team full-stack nella vita reale: London's Royal Free Hospital**
Per curare l'ebola hanno fatto un gruppo fullstack di 30 persone con 12 clinici e 21 non clinici, una specie di mini ospedale autonomo. Gruppi di lavoro in grado di gestire tutte le parti.

**DevOps**
Nei primi 50 anni di informtica veniva fatto il development (DEV), poi un muro rosso e il tutto veniva passato agli operatori che mantenevano l'applicazione (OPS) e questa differenza si percepiva anche nell'applicazione stessa. Adesso con Dev0ps non facciamo progetti ma facciamo prodotti. Tu lo hai costruito e tu lo mandi in esecuzione.

**PRO e CONTRO dei microservizi**

**PRO**

- Minore lead time
- Scalabilità

**CONTRO**

- Eccessiva comunicazione (tra servizi)
- Complessità
- Sbagliare a “tagliare” un microservizio
- È più difficile identificare la causa di un abbassamento delle performance
- Difficoltà nell’evitare la duplicazione dei dati mantenendo i microservizi isolati
- Un team deve essere preparato e in grado di gestire e creare dei buoni microservizi

**Conclusione:**
I microservizi non vanno presi in considerazione se l'applicazione funziona in modo semplice e non è troppo complessa come monolite.

---

## 10° LEZIONE: Architettural Smell e soluzioni

L'applicazione che ho fatto con i microservizi rispetta i principi dei microservizi? E se non li rispetta come faccio a renderla davvero un'app con i microservizi?

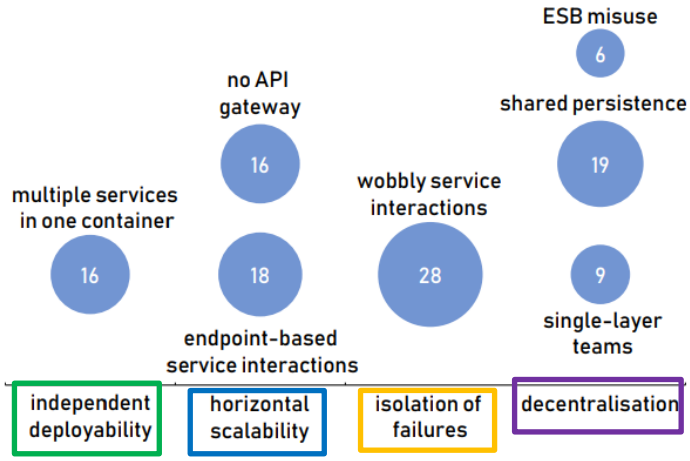
**Architectural smell**
Cattivo odore architetturale, un po' meno brutto di antipattern, un'architettura che molto probabilmente non dovrebbe essere così.

Domanda: Come è possibile risolvere degli architectuale smell che possono affliggere il design dei micoservizi?
Ci sono un milione di possibili risposte. Review di articoli scientifici e blog aziendali:

**Design principles**

- Microservizi Indipendenti
- Scalabilità Orizzontale
- Decentralizzazione
- Isolare i fallimenti

**ARCHITECTURAL SMELLS**




*Studio su 41 applicazioni*

1. **Multiple service in one container**
   Se devo usare un servizio singolo è difficile perché sono tutti nello stesso container

2. **Endpoint based service interaction**
   Potenzialmente viola scalabilità orizzontale. Se ho un servizio che ne invoca un altro se poi questo altro lo faccio scalare creando altre 2 istanze il chiamante continua a invocare sempre lo stesso di prima.
   **Refactoring:**
   Le soluzioni sono quelle di usare un "intermediario" che invoca l'istanza migliore, oppure usare una coda in cui viene messa la richiesta e poi dopo si consuma il messaggio dalla coda.
   Message broker (message queue), Message router (load balancer), Service discovery.

3. **No API Gateway**
   Con un API gateway il cliente non chiama direttamente i servizi interni ma chiama l'API gateway che gestisce la sua richiesta.
   **Refactoring:**
   Se un'architettura non ce l'ha questo deve essere aggiunto in modo che i clienti non possano fare invocazioni all'interno della nostra architettura. L'API gateway viene utilizzato anche per l'autenticazione.

4. **Shared persistence**
   Più servizi accedono e condividono lo stesso DB
   **Refactoring:**
   Soluzione 1: dividere il database se possibile, a volte ci sono doppioni e quindi bisogna aggiornare più database.
   Soluzione 2: Introdurre un **data manager**, s1 e s2 invocano le operazioni offerte dal data manager, così quando dobbiamo modificare il database s1 e s2 non lo sanno. Il prezzo da pagare è il data manager, ci sono 2 chiamate, una per dm e una per db.
   Soluzione 3: se s1 e s2 condividono lo stesso database può essere che s1 e s2 siano lo stesso microservizio e debbano essere uniti.

5. **ESB misuse**
   Alcune applicazioni utilizzano questo ESB (Enterprise bus), parte dell'intelligenza e della business logic va a finire in questo punto centralizzato violando l'idea dei microservizi di endpoint e dumb message.

6. **Single-layer teams**
   Ci devono essere esperti di cose diverse in ogni team.

7. **Wobbly service interaction**
   Quando un'interazione tra m1 e m2è traballante(wobbly)? Quando un fallimento di m1 può scatenare un fallimento su m2.
   **Refactoring:**
   Soluzione 1: aggiungere un **circuit breaker** , una specie di proxy, il proxy chiude il circuito in modo che m2 non fallisca.
   Soluzione 2: **message broker** cioè coda asincrona, dove i produttori mandano messaggi alla coda e i consumatori sono staccati, ma si limitano a leggere dalla coda, un fallimento del poduttore non genera un fallimento del chiamante. Il servizio *A* chiama il message broker e mette la richiesta e poi *B* prende la richiesta dal Message broker.

Soluzione 3: **timeouts**, meccanismo in cui il chiamante ha una chiamata timed e se non arriva una risposta può interrompere l'attesa e non fallire
Soluzione 4: **Bulkhead**, paratie, cerca di definire il confine più grande all'esterno del quale non ci deve essere condivisione dei dati e chiamate singole dei servizi. All'interno della paratia condivido dati ma quando l'attraverso non devo più farlo per avere un isolamento dei fallimenti. Prevedere che non ci siano chiamate sincrone tra servizi.

Tutto questo introduce un costo, se metto un message broker se prima avevo 2 messaggi ora sono 4 ecc..
Aumenta la comunicazione.

**uFreshener (microFreshener)**
Ci offre una visione grafica di queste soluzioni.
Ci permette anche di avere una visione teambased nell'ultimo aggiornamento.
Possiamo scegliere di vedere solo alcuni microservizi, per esempio qulli più critici o importanti.

**Conclusioni:**
Uno smell non viola per forza dei principi
Una volta individuati gli smell ci sono soluzioni diverse che poi vengono decise dal manager.
Molti "amen", se c'è un db condiviso tra 3 servizi si lascia così perché può avere un costo cambiarlo

FINE PRIMO COMPITINO ----------

Info utili compitino:

il message queue è un tipo di message broker

0,9 * 0,9 percentuale uptime di 2 microservizi indipendenti al 90% uptime l'uno

---

## 1° LEZIONE: Aziende e microservizi (SPOTIFY)

**Spotify e microservizi**

Perché Spotify ha deciso di passare ai microservizi?

**Deve ridurre il Lead Time**

- Spotify ha la necessita di innovare il proprio servizio e reagire in maniera rapida per essere competitivo in un **mercato che si muove velocemente**, se non fa niente rischia di perdere parte di questo mercato, se gli altri introducono nuove funzionalità anche Spotify deve farlo, e deve farlo rapidamente.
- Spotify deve essere supportato su molte piattaforme, per ogni nuova funzionalità aggiunta essa deve essere supportata su ogni dispositivo.
- Inoltre versioni vecchie del servizio devono essere funzionanti.

**Deve scalare**

- Spotify ha più di 75 Milioni di utenti attivi con una sessione media di 23 minuti. Deve fronteggiare un grandissimo numero di richieste.

**Quando e come è avvenuto il passaggio ai microservizi?**

All'aumentare del numero di richieste l'interazione con il **Database** di Spotify ha cominciato a rallentare e non scalava. La soluzione iniziale è stata quella di replicare le istanze del Database (cioè far scalare il back end del DB) e hanno resistito.
All'aumentare nuovamente degli utenti il Database resisteva ma il **Server** no, allora hanno scalato il Server replicandolo (lato business logic) mettendo davanti dei Load Balance per sostenere il carico.
Il problema è che aumentando ancora di più gli utenti e avendo la necessità di modificare l'applicazione con **tanti device e con tante versioni** questa architettura non permetteva di raggiungere entrambi gli obiettivi e allora sono passati ad un'organizzazione a **microservizi** e hanno riorganizzato anche i propri teams (**Full-stack autonomous team**).
Questo passaggio ha comportato operazioni molto complicate, tra cui anche la riorganizzazione stessa dei teams.

**Vantaggi dei microservizi secondo Spotify**

- **Facile scalare**
- Più facile fare i **test**
- Più facile **il deploy**
- **Più facile monitorare**, non bisogna fare un monitor di tutta l'applicazione ma possiamo anche solo monitorare parti diverse e più piccole
- **Versionato in maniera indipendente**
  uno dei problemi di Spotify è che io posso comprarmi una lampada intelligente e mettere le credenziali di spotify e usarlo sulla lampada intelligente. Il business model della lampada intelligente è molto diverso da quello di Spotify: viene venduta come un bene non come un servizio. La lampada non verrà mai più aggiornata, con le nuove versioni di spotify se la lampada un funzionerà il cliente si lamenterà con spotify non con la lampada. Quindi è molto importante poter mantenere tante versioni della propria applicazione.

- **Meno suscettibile ai grandi fallimenti**
  con un'organizzazione a microservizi sono riusciti a isolare il fallimento perché appunto sono indipendenti, come per esempio la ricerca.

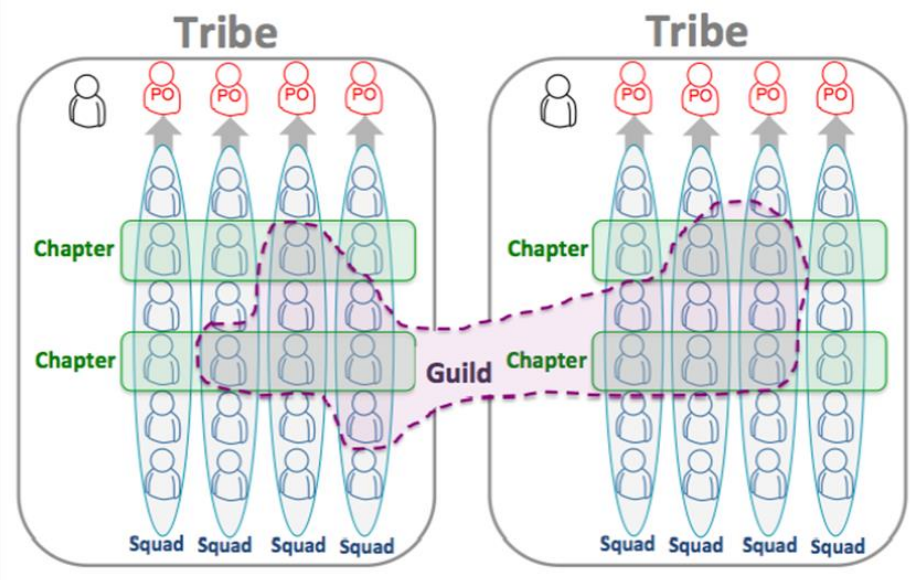
**Svantaggi dei microservizi secondo Spotify**

- Se i servizi sono tanti e le istanze dei servizi ancora di più diventano **difficili da monitorare**, è vero che possiamo trovare metriche diverse per servizi diversi ma la **quantità aumenta**
- Rivedere su slide(?) Need good documentation/discovery tools
- Latenza aumenta, si generano più comunicazioni e **aumentano i ritardi**

**Alcuni dati:**

90+ teams
600 sviluppatori
5 uffici di sviluppo
1 prodotto
**810 servizi attivi**
10 servizi per squad

**La struttura dei team** di lavoro di Spotify si basa sull'**Agile Software Engineering**.
Descrizione delle varie parti:




Una **squadra** (squad) ha un **Product Owner**. Le squadre hanno tutte le competenze e gli strumenti necessari per progettare, sviluppare, testare e rilasciare funzionalità, essendo un **team autonomo** e auto-organizzato con un esperto per ogni area di lavoro.

Una **tribù** (tribe) è una raccolta di squadre all'interno della stessa area commerciale (ad es. Mobile). Le squadre all'interno di una tribù risiedono nella stessa area. Spotify implementa incontri condivisi per creare interazioni tra squadre e eventi informali dove le squadre condividono ciò su cui stanno lavorando. Il leader della tribù è responsabile di fornire e creare un ambiente ideale per tutte le squadre.

I **capitoli** (Chapters) sono membri del team che lavorano in un'area speciale (ad es. Sviluppatori di front office, sviluppatori di back office, amministratori di database, tester). I Chapters si scambiano idee per promuovere l'innovazione e "l'impollinazione incrociata" tra i team. Si occupano soprattutto della review dell codice, quando una squad vuole produrre una nuova versione di un servizio chiede una review ad un paio di esperti dell'altro team che fanno parte del Chapters.

Una **gilda** (Guild) è una comunità di membri all'interno dell'organizzazione con interessi condivisi (ad es. Tecnologia web, automazione dei test), che vogliono condividere conoscenze, codice degli strumenti e tecniche.

[VIDEO SPOTIFY]

**Usa principi dell'agile software engineering**

Quindi teams organizzati come sopra, release frequenti e piccoli, focalizzandosi sulla motivazione, community e fiducia.

**Fail fast->Learn fast->Improve fast:** certamente ci saranno dei fallimenti, dopo i quali si fanno delle analisi e si cerca di imparare qualcosa, meglio fallire che evitare i fallimenti. Si lavora con il cosiddetto **Limited blast radius**, cioè un raggio di danno limitato: il problema di Spotify è che avendo milioni di utenti quando si sviluppa un nuovo servizio potremmo causare problemi a tutti, ma loro lo rilasciano gradualmente prima solo ad alcuni utenti in modo da limitare eventuali malfunzionamenti.

**Approccio allo sviluppo:** principi di **Lean Startup**. L'Idea è cercare di dare più imporanza all'**impatto** che alla velocità dell'innovazione e inoltre anche dare più importanza all'**Innovazione** che alla Prevedibilità, 100% prevedibilità = 0% innovation.

Delivery value > Plan fullfillment

**Hack time**: si vuole che i dipendenti dedichino 10% del tempo in Hack Time dove possono dedicarsi a fare cose innovative per favorire la creatività.

**Waste-repellent culture:** ciò che non ci interessa si butta, tutto ciò che non crea valore (time report, meeting inutili, task estimates ecc..). Viene utilizzata la Lavagna dei miglioramenti,dove scrivere quello che vorremmo migliorare del nostro prodotto.

Minimizzare il bisogno di "Big projects" perché aumentano i rischi, meglio dividere in piccoli progetti in modo da minimizzare anche la burocrazia e la documentazione.

**Health culture**: creare un ambiente sano e piacevole per i dipendenti

---

## 2° LEZIONE: Aziende e microservizi (NETFLIX + altri)

### Netflix e microservizi

Perché Netflix ha deciso di passare ai microservizi?

**Deve ridurre il Lead Time**

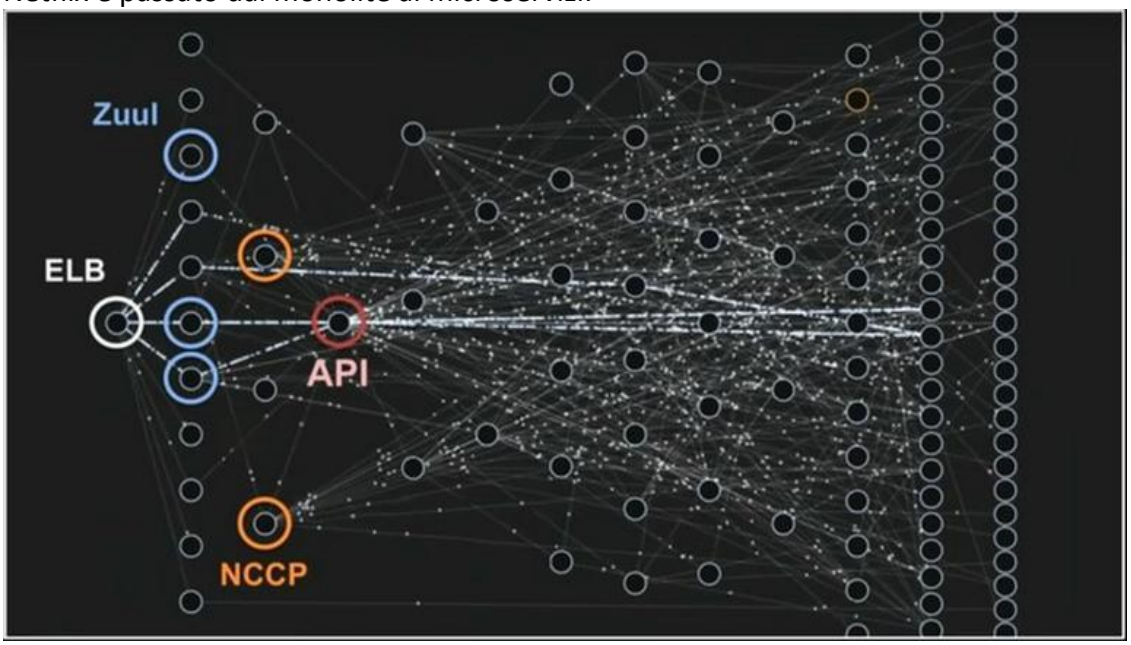
- Ha bisogno di automatizzare e velocizzare gli aggiornamenti
- Deve supportare più di 1000 tipi di device differenti

**Deve scalare**

- Più di **86 milioni di membri, 150 Milioni di ore** di streaming
- **15% di tutto il traffico in downstream di internet**

[VIDEO NETFLIX]
Netflix utilizza **100'000 istanze su Amazon Web Services**, dove si sono spostati nel periodo **2008-2016 chiudendo** tutti i loro data center, possiede circa **500 microservizi** ed è uno dei più grandi distributori di open source (come Simian Army dove c'è Chaos Monkey).
Per Netflix l'introduzione dei container ha creato problemi al supporto tecnologico, dopo aver provato a risolvere da soli il problema dei container (con Fenzo) hanno deciso di utilizzare AWS Blox di Amazon.

Netflix è passato dal monolite ai microservizi.




I cerchietti neri sono i microservizi di Netflix, le linee rappresentano lo scambio di messaggi e come si può notare l'architettura è molto complessa, sono più di 500 microservizi che interagiscono tra loro.

Per facilitare e gestire al meglio questa struttura sono stati introdotti:

- **ELB:** Elastic Load Balancing di AWS per instradare il traffico attraverso il primo livello
- **Zuul**: fa da proxy per le richieste che vengono dal Load Balancing, effettua un routing dinamico fino ad arrivare all'API di Netflix
- L'**API** di Netflix chiama tutti gli altri servizi
- **NCCP**: mantiene le vecchie versioni, legacy tier

**Quali sono i problemi di Netflix con i microservizi:**

- **Richieste tra servizi diversi**

il fatto di avere un'architettura con tanti microservizi è che il servizio A chiama il servizio B, questo viene descritto come attraversare un "abisso", può causare fallimenti o congestione della rete, se cadono tutti i servizi a catena fino all'API Gateway è finita.
Cosa ha fatto Netflix?
Ha sviluppato **Hystrix** che permette di introdurre tecniche come Timeouts and retries, Fallback (permettere al cliente di continuare a usare il servizio) e Pool e circuiti di thread isolati.

Come facciamo a saper se la nostra applicazione funzionerà su scala?
Si fa un'introduzione di fallimenti tramite chaos engineering.

Testare visione complessiva del sistema, critical microservices, 2 livelli di testing, il livello con critical microservices invece di testare tutti i microservizi fare solo quelli che ci servono.(??? too fast prof)

- **Librerie clienti**

Se usiamo troppe librerie clienti rischiamo di rendere l'architettura un monolite, soluzione:
cercare di tenere le librerie clienti solo se sono davvero utili, ma con moderazione e mantenendole molto semplici.

- **Persistenza**

**CAP theorem:** In un sistema informatico **non** possiamo garantire contemporaneamente:

- Coerenza/Consistenza (cioè che tutti i nodi vedano gli stessi dati nello stesso momento)
- Disponibilità/aivalability (la garanzia che ogni richiesta riceva una risposta su ciò che è riuscito o fallito)
- Tolleranza di partizione [partitioning] (il sistema continua a funzionare nonostante arbitrarie perdite di messaggi o malfunzionamenti)

Se abbiamo 3 istanze di un database B, C e D può succedere che arrivino 3 richieste e che la richiesta sul database C fallisca, quindi non viene aggiornato.
Dobbiamo quindi accettare che non sia consistente, chi usa database B e D non avrà problemi, il database C invece non sarà aggiornato. Se invece vogliamo la consistenza dei dati dovremo fare un rollback e non aggiornare né B, né C né D.

Cosa ha fatto netflix per risolvere questo sistema?
Ha scelto la strada dell'**eventual consistency** (letteralmente “consistenza prima o poi”) utilizzando Cassandra, la strategia è: scrivi dove puoi scrivere, prima o poi dovrai aggiornare anche i database che non sono stati aggiornati, viene data priorità all'aivalability.
C'è un quorum minimo, cioè almeno un tot di database devono essere stati aggiornati, sennò non si può applicare l’eventual consistency.

- **Infrastructure**

I server Amazon sono andati down la notte del 24 dicembre 2012 causando la non disponibilità di Netflix, questo perché avevano una sola regione di server, ora Netflix, per evitare avvenimenti simili, usa una strategia multiregioni su più regioni di Amazon.

- **Scalare servizi senza stato**

Un servizio senza stato è un tipo di comunicazione in cui le richieste tra client e server sono indipendenti e non memorizzano uno “stato” (un carrello pieno su amazon è uno stato).
In questo caso la perdita di un nodo è un "non evento", per i servizi che non portano a uno stato la strategia di Netflix è replicare le istanze e poi testare con Chaos Monkey.

- **Scalare servizi con stato**

La cosa più brutta è quando va giù un servizio che ha uno stato.
Come si risolve? Quando deve essere fatta la scrittura di un dato viene fatta in zone multiple, la read cerca di leggere dalla zona più vicina il dato, ma nel caso di fallimento, la lettura viene ripetuta sulle copie anche se si trovano in zone di availability diverse, se c'è un disastro bisogna recuperare i dati da un clone.

30 milioni di richieste al secondo
2 trilioni di richieste al giorno
tantissimi oggetti e istanze

La latenza è dell'ordine dei millisecondi Ottimo scaling! 😊

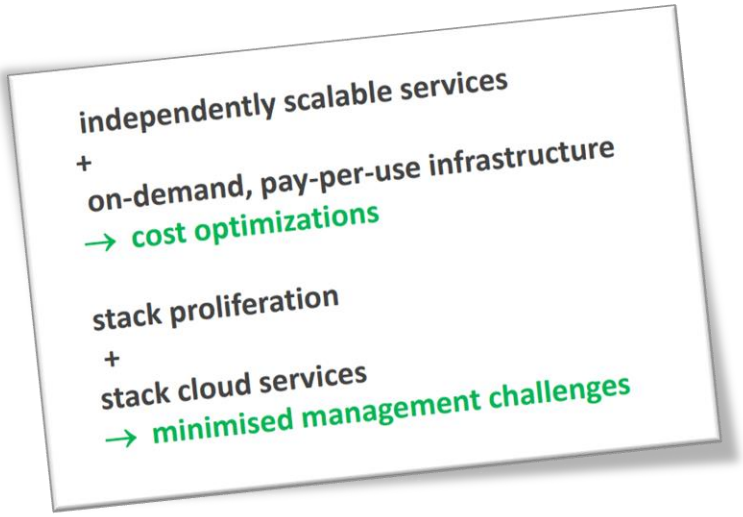
**Many tools**
Netflix utilizza tanti software ma ha anche i propri open source.
[VIDEO NETFLIX]

**comparethemarket.com e i microservizi**
È un servizio molto popolare nel Regno Unito che permette di comparare i prezzi delle assicurazioni. Utilizza i microservizi perché ha bisogno di rimanere aggiornato in tempo reale con i cambiamenti dei prezzi e perché ha circa 10 milioni di visite mensili.

**Uber e i microservizi**
Uber prima non scalava, adesso sì.
78 milioni di visite al mese e tanti servizi diversi (Uber eats, Uber Bike ecc.)

Chi altro usa i microservizi?
Amazon, ebay, Google, linkedin, facebook, twitter

**Che collegamento c'è tra Microservizi e Cloud?**
Io potrei avere la mia applicazione a microservizi e non usare il cloud, ma molto spesso i microservizi vanno a braccetto con cloud, viene deployata sul cloud.
Perché?




1. I microservizi mi permettono di avere parte della mia applicazione indipendente e scalabile e se posso usare il Cloud in modalità on demand e pay-per-use ottengo un'ottimizzazione dei costi.
   La scalabilità è teoricamente illimitata con un pagamento proporzionale all'uso. Ottimo 😊

2. Il poliglottismo porta alla proliferazione degli stack, questo vuol dire che posso rendere più efficiente e flessibile la mia applicazione, e dall'altra parte devo però gestire molteplici **stack** ☹.
   Se posso usare servizi Cloud che si occupano autonomamente di gestire gli stack allora questo mi porta molti benefici. Ottimo 😊

---

3° LEZIONE: FaaS e Amazon Lambda

## FaaS: Function as a Service
Function as a Service, è nato con un servizio offerto da AWS che si chiama Lambda.
L'idea è semplice, nel caso di IaaS si doveva far partire la macchina virtuale, i server ecc.
Qui è serverless

## Amazon Lambda (Faas) (Freemium)

Il cliente deve mandare in esecuzione il codice:

- può farlo senza dover gestire un server quindi zero amministrazione, non deve neanche sapere cos'è una macchina virtuale.
- Basta uplodare il codice e poi Lambda pensa alla scalabilità
- Puoi invocare il codice con dei trigger oppure invocarlo direttamente da mobile o web
- Paghi a seconda del tempo di calcolo consumato

[VIDEO AWS LAMBDA]

Possiamo o caricare una funzione che già abbiamo, oppure possiamo scriverla o prenderla da degli esempi già pronti.
Una volta passata la funzione Lambda la manda in esecuzione quando avviene il trigger specificato e si occupa di tutta l'infrastruttura per il codice (macchina virtuale, scalabilità, far partire il processo, metriche e monitoring) tutto con un costo basso secondo Amazon

**Costi**
Per richiesta: 0.2 $ per 1 Milione di richieste
Per durata: 0.0000166667 dollari per ogni GB-second (dipende da quanta memoria allochiamo per la funzione)

**Esempio:**
512MB per la funzione usata 3 milioni di volte in un mese con durata di 1 secondo ogni volta

3,000,000 / 1,000,000 * 0,20 $ = 0.60$ per richiesta
+
3,000,000 * 512/1024(circa mezzo GB) * 0,0000166667$ = 25$
=
25,60 $ totale

È un modello **Freemium,** si ha un usage tier con **1 milione di richieste gratis e 400,000 Gb-seconds di compute time al mese.**

**Esempio Freemium:**
512MB per la funzione usata 3 milioni di volte in un mese con durata di 1 secondo ogni volta

(3,000,000 - 1,000,000) / 1,000,000 * 0,2$ = 0.40 $
+
((3,000,000*1) * (512/1024) – 400,000 ) * 0.0000166667
=
18.33 $ totale

**Come si usa?**
[video random number]
Ci fa vedere come configurare Lambda per creare le funzioni.
Per testare una funzione col trigger bisogna associare il codice del trigger, può essere uno degli eventi dei servizi di amazon (tipo quando si modificano cose in Amazon S3)
vediamo esempio su come usare API Gateway per renderla utilizzabile all'esterno
[video API gateway]

molto semplice, possiamo decidere con cosa far partire il nostro codice.

## Quale FaaS usare?

Ci sono diversi FaaS
Attraverso uno studio sono state comparate 10 piattaforme (tra cui Amazon, Google e Microsoft).
2 punti di vista: **Business View** e **Technical View**

### Licenza

Tutte quelle open source usano licenze permissive mentre quelle commerciali usano quelle proprietarie, a parte Azure che usa anche alcune open source.

### Installazione

Cioè i server FaaS possono essere usati on-premises (attraverso l'installazione) oppure as a service. Quasi tutte quelle opensource sono disponibili on-premises, mentre quelle commerciali as-a-service fatta eccezione per Azure che può anche essere installato.

### Source code

Questo aspetto è di interesse sia per l'aspetto tecnico che del business. Nel caso degli open source il codice è accessibile direttamente su GitHub e come implementazioni sono in Go.
Per quelle commerciali non è disponibile il codice sorgente tranne che per alcune cose di Azure.
**La scelta di un FaaS Open source riduce i problemi di vendor lock-in.**

### Release

Chiaramente tutte le piattaforme commerciali sono aggiornare e in produzione. Mentre per gli open source 4/7 candidate a release, 1/7 stable release e 1/7 ready to market.
Leggermente indietro rispetto alle piattaforme commerciali.

### Interface

CLI tutti (Command Line Interface)
API 7/10
GUI 6/10 interfaccia grafica
tutte le piattaforme supportano le operazioni di base, le piattaforme open source sono disomogenee in come ci permettono di amministrare la nostra applicazione.

### Community

Qual è quello più utilizzato? su GitHub dove ci sono solo gli open source sono più popolari quelli sugli opensource
Su stackoverflow quasi tutto su AWS Lambda

### Documentation

Che tipo di documentazione abbiamo? Tutte le piattaforme forniscono la documentazione su come sviluppare applicazioni e sia su come possiamo utilizzare la piattaforma.
Nel caso dell'open source cerchiamo altra documentazione su come è strutturata l'architettura e sviluppata, solo alcuni documentano.

### Quotas

Se vengono imposte quote di utilizzo, nelle piattaforme commerciali bisogna cambiare sottoscrizione per modificarla. Mentre in quelle Open source non ci sono quotas

### Development

Quali linguaggi di programmazione vengono supportati? Su queste 10 **non ce n'è uno che è supportato da tutti.** Quelli più supportati sono Python e NodeJS. Anche Docker images.

### Versioning

Quelli Open Source usano meccanismi Impliciti, mentre quelli commerciali utilizzano meccanismi Dedicati.

### Event sources

(Non importa saperlo)

### Function orchestration

Avremo nel nostro codice più funzioni combinate tra loro. Può esserci oppure no un orchestratore di funzioni dedicato, **la maggioranza delle piattaforme considerate lo offre**, la metà di questi usano dei DSL specifici per gli workflow, diagramma di flusso con composizioni sequenziali, parallele ecc..

### Testing & debugging

6/10 offrono testing
debugging 8/10
Quelli open source offrono il **testing solo tramite function invocation**

### Observability

Monitorare la nostra applicazione
Ci sono tanti strumenti diversi, nel caso di piattaforme commerciali si usano strumenti dello stesso provider. Gli open source integrano **servizi di terze parti.**

### Application delivery

La quasi totalità usa approccio dichiarativo al deployment, cioè permette all'utente di dichiarare qual è lo **stato che vuole raggiungere** senza dichiarare i passi, tipo vorrei che soddisfacesse questa proprietà e la piattaforma lo fa.

### Code reuse

Lambda e Azure ci offrono tante funzioni già pronte.
Tutte le piattaforme hanno degli esempi di funzioni, ma **queste due hanno più cose.**

### Access management

Quelle commerciali **supportano autenticazioni alle risorse** mentre quelle Open Source si appoggiano a **servizi esterni.**

**Conclusione:**
Diversità delle piattaforme esistenti, non c'è una piattaforma migliore, dipende da quali sono i bisogni dell'azienda o di chi sviluppa l'app. C'è il **FaaStener Prototype** che ci permette di scegliere il FaaS più adatto.

**Rischio di Lock-in**
*"Lambda è una delle peggiori forme di lock-in mai viste."*
cit. Alex Polvi CEO CoreOS

Il fatto di poter scrivere così facilmente le funzioni e renderle collegabili è un grosso vantaggio in termini di velocità di sviluppo e riduce la quantità di testing. Di contro una volta sviluppata tutta la mia applicazione se voglio cambiare provider devo aprire il codice sorgente e modificarlo tutto. Ci sono comunque delle strategie per uscire o quantomeno ridurre Lock-in. Se non si vuole rischiare si può sempre usare l'open source.

*"More than 20% of global enterprises will have deployed serverless computing technologies by 2020"*
*cit.* Gartner, Dec 2018

---

## 4° LEZIONE: Vendor Lock-In

Il lock in è una cosa che i venditori vorrebbero. Il Vendor lock-in rende un cliente dipendente ad un prodotto o servizio, il cliente è impossibilitato ad usare un provider diverso senza sostanziali costi relativi al cambio.

### Vendor lock-in nel mondo del Cloud

Concetti presi da interviste:
Nel 2013 **4/5** degli intervistati pensavano che firmare con un Cloud significava fare un lock-in.
La portabilità era un "sogno", le compagnie si sveglieranno.

**Data gravity**: più è grande la quantità dei dati più è difficile spostarli, i dati ti "ancorano" a terra

Lambda è una delle peggiori forme di lock-in

Impariamo dagli ultimi 20 anni: è molto probabile che vogliamo passare da un venditore ad un altro nel tempo, i mercati sono molto dinamici e in evoluzione, in quel momento potremmo avere dei problemi.

Il lock-in è solitamente nel contratto e viene esplicitato, ma nel caso del cloud **il lock-in non è specificato** perché non è un lock-in a pagamento ma è vincolante a livello tecnologico.

**Livelli di lock-in**
Ovviamente Lock-in FAAS>>PAAS>>IAAS

**Esempio di vendor lock-in:**
**Migsolv:**
Hanno deciso di usare un Cloud conosciuto ma avevano avuto 2 problemi: il costo era aumentato e anche il creare reports era diventato difficile.
E allora ad un certo punto hanno detto: cambiamo provider!
Poblemi:

1. I dati erano memorizzati in modo diverso tra i due Cloud, per riuscire a trasportarli dovevano prima manipolare questi dati per poterli poi trasferire. **Ogni dato andava trasformato in un nuovo formato**.

2. Il vecchio provider **non aveva funzionalità di export**

Alla fine sono riusciti a fare tutto ma ci hanno messo alcune settimane per copiare i dati.
La lezione che hanno imparato è: **prima di scegliere un fornitore dobbiamo cercare di capire meglio come vengono salvati e gestiti i dati.**

**Esempio di Platform lock-in: (Tesi di uno studente)**
Tesi: Sviluppare un'applicazione più semplice possibile per vedere se c'è un tipo di lock-in.

Lo studente ha usato il Python e un add-on di Google per creare un login, poi è voluto passare a Microsoft Azure, ma non supportava il Python, i dati del database andavano estratti uno per uno e salvati di nuovo e anche il funzionamento dell'applicazione andava modificato, è stato necessario riscrivere uno script per supportarlo.

**Altri tipi di lock-in**:

- La tua applicazione usa **servizi di terze parti** supportate solo da questo provider
- Non sei il **proprietario** dell'applicazione che gestisce i tuoi dati
- La tua applicazione usa **linguaggi proprietari**
- I **dati** devono essere riconvertiti per essere spostati
- Non sei proprietario delle "**side information**" (come i log files)
- Non controlli **l'operating system platform**

**Ma perché dovrei cambiare fornitore?**

- Il provider fallisce
- aumento dei costi del provider
- diminuisce la qualità del servizio (response time per esempio)
- possono cambiare i termini di servizio cioè le funzionalità che posso usare col piano che ho
- troppi **outages (interruzioni)**
- non supporta alcune features che vorrei

## Outages 2019

Salesforce dopo aver mandato in esecuzione uno script ha avuto 20 ore di disservizi
Amazon ha avuto un calo di energia e il 7.5% delle istanze non funzionavano più e si sono danneggiati permanentemente
Apple a luglio non gli funzionava l'iCloud
Microsoft ha avuto per più di un'ora problemi di connettività
Anche Google, Instagram e Whatsapp hanno avuto interruzioni.

## Qual è la soluzione?

- Ci rassegniamo, amen, accettazione.
- Scegliere con prudenza il fornitore ("uscite" con i fornitori ma non li "sposate"), usare con attenzione gli add-on e le funzionalità
- Pianificare una exit strategy
- Usare più Cloud
- DevOps: usare architetture blandamente accoppiate come i microservizi, Container, API/REST
- Usare open-source o meccanismi unificati, cioè dove posso usare gli stessi API per esempio

## STANDARD CLOUD: sono standard pubblici e dicono come i Cloud devono essere

Il processo di standardizzazione tradizionale è che prima le cose avvengono e poi dopo vengono scritti gli standard, poi in realtà ognuno fa come vuole, come Amazon che dice "siamo noi lo standard".

**CDMI:** definisce come deve essere fatta un'interfaccia REST per gestire i dati

**OCCI:** Rest per API e IaaS

**CIMI:** Standardizzato il modo in cui fornitori e consumatori di un servizio possono interagire con L'IaaS

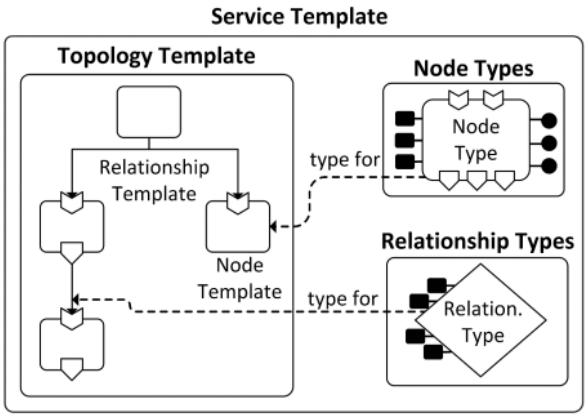
**OVF**: Chi adotta questo formato favorisce la portabilità tra piattaforme diverse

**CAMP**: standard di come deve essere fatta una API di un gestore PaaS in modo che si possa fare la portabilità

**TOSCA:** Standardizzare un modo per descrivere le applicazioni con l'obiettivo di ottenere applicazioni portabili, associo alla mia applicazione una descrizione con delle caratteristiche che deve fare in modo di potermela far trasferire senza costi aggiuntivi associandole un grafo dei requisiti e le relazioni tra i componenti. Questo ci permette poi di fare il deployment e il management di tutta l'applicazione. Alla fine viene tutto impacchettato in un file CSAR con tutti gli script necessari.
Lo standard TOSCA viene seguito da molte aziende (CISCO, Huawei ecc.), permette di ridurre il vendor lock-in e di automatizzare il management e il deployment delle applicazioni cloud.

Obiettivi:

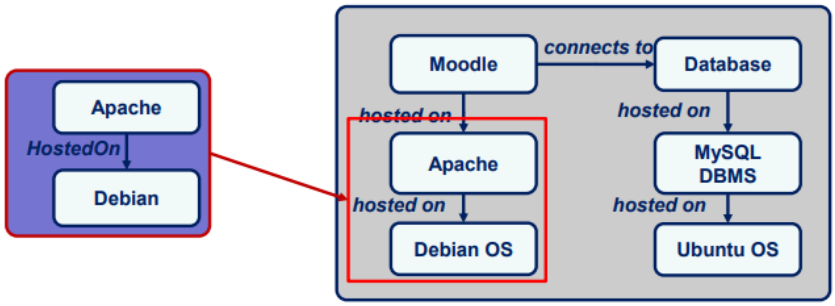
1. **Portabilità**




```yaml
tosca_definitions_versions: tosca_simple_yaml_1_0

imports:
  - tosca-normative-types: 1.0

topology_template:
  node_templates:
    apache:
      type: tosca.nodes.WebServer
      capabilities:
        host: {}
      requirements:
        host:
          node: debian
          capability: host
          relationship: apache_hosted_on_debian

    debian:
      type: tosca.nodes.Compute
      capabilities:
        host:
          num_cpus: 4
          disk_size: 1 TB
          mem_size: 4 GB
        os:
          distribution: debian
          version: 8.2

  relationship_templates:
    apache_hosted_on_debian:
      type: tosca.nodes.HostedOn
```

Associamo alla nostra applicazione un file che la descrive.

2. **Riusabilità dei componenti**




I pezzi possono essere usati per fare descrizioni più grandi assemblandole

3. **Automatizzare il deployment**
   Nella mia descrizione posso dichiarare uno script, ogni nodo TOSCA ha delle operazioni standard che possiamo eseguire, ognuna di queste può essere implementata semplicemente allegando uno script eseguibile

```yaml
node_templates:
  ...

  apache:
    ...
    interfaces:
      Standard:
        create:
            implementation: apache-install.sh
```

---

## 5° LEZIONE: Containers

**Docker**

Docker è una piattaforma che ci permette di mandare in esecuzione app in ambienti isolati, sfrutta un meccanismo di virtualizzazione chiamato Container.

**Container**

A differenza delle macchine virtuali, che richiedono memoria, cpu ecc.. i container richiedono meno risorse, sono più veloci ad avviarsi e più facili da costruire, sopra il sistema operativo viene installato il **Docker Engine** e poi direttamente i container. Lo svantaggio è che devono condividere tutti lo stesso

sistema operativo e condividono più risorse del SO rispetto alle macchine virtuali, questo significa che potrebbero essere più vulnerabili, meno sicuri. Un altro vantaggio dei container è che se voglio eseguirne uno sulla mia macchina non devo scaricare librerie, tools ecc. ma mi basta scaricare il container che è un piccolo ambiente indipendente, inoltre il suo funzionamento non dipende dalla macchina che uso.

## Docker

### Storia dei container

L'idea dei container esiste da circa da 10-15 anni ma solo dal 2013 ha preso ancora più piede grazie a **Docker**, perché ha aggiunto due pezzi che hanno reso questa tecnologia migliore: **immagini portabili** e **un'interfaccia utente più amichevole**.
La piattaforma Docker contiene:

- **Docker Engine** (per creare e far partire i container)
- **Docker Hub** (è uno dei repository, qui troviamo le immagini)

### Come funziona Docker

I software sono impacchettati come **immagini** che vengono usate per creare e eseguire i **container.**
**Volum**i: volumi esterni che possono essere montati per assicurare la persistenza dei dati.

### Docker images:

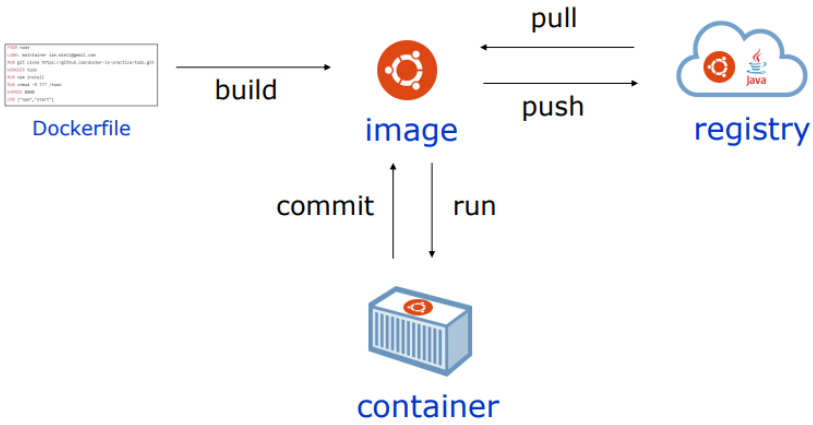
Sono read-only e sono **template** usati per creare i container
sono dentro i registri, ad esempio abbiamo una cartella per Ubuntu e abbiamo più immagini per versioni diverse di Ubuntu.
Le immagini sono strutturate in livelli, il più basso è **base image**, possiamo costruire un DockerFile aggiungendo più livelli ad una base image.

### DockerFile:

Text file con una lista di steps per ceare l'immagine corrispondente, ad esempio le configurazioni necessarie ecc. Non si deve partire da zero ma si utilizzano le immagini prese direttamente da Docker Hub.
I DockerFile iniziano sempre con FROM e poi viene specificata l'immagine.

### Docker commands:

Permette di mandare in esecuzione le immagini




registry--pull--->ottengo immagine(con tutte le info dell'app, dipendenze, librerie, ecc.)--run---> manda in esecuzione in container
Se vogliamo salvare le modifiche possiamo eseguire un commit e creiamo una nuova immagine.
Per creare l'immagine si può anche usare un Dockerfile dove specifichiamo come vogliamo la nostra immagine.

[VIDEO DOCKER]
Dato che il container è costruito da un'immagine read only allora se io modifico qualcosa non vedo i risultati, devo rebuildare tutta la cartella, **per questo esistono i volumi che ci permettono di avere persistenza condividendo le cartelle tra l'host e il container.**
Il container può essere fermato con CTRL C oppure quando il processo finisce il container si ferma, per questo **è meglio mettere un processo solo in un container** così si ha un singolo ciclo di vita e non più cicli di vita insieme.

**Docker compose:**
[VIDEO]Comando per orchestrare più microservizi insieme, ci servono più file: api.py, index.php, requirements.txt, Dockerfile, docker-compose.yml dove specificare le varie configurazioni.

**Docker Swarm mode:**
Gestisce un **cluster** (= insieme di computer connessi tra loro), si mandano in esecuzione istanze diverse su host diversi, è simile a Heroku (workers e manager).
Puoi definire lo stato dei servizi e dire quanti ne vuoi, ad esempio se vogliamo 10 repliche di un container e 2 di queste dovessero crashare lo Swarm manager ne crea subito altre 2 e le assegna agli Workers liberi.

**Kubernetes** [VIDEO]
Non è in competizione con DockerSwarm, Docker ha permesso di far usare Kubernetes per **l'orchestrazione e la coordinazione dei container**, si occupa solo di questo ed è adatto in caso di cluster medio grandi con applicazioni complesse.
Anche qui l'utente può specificare qual è il suo goal.

**Pod:** un gruppo di uno o più containers, con storage/network condivisi e una specificazione su come eseguire i containers. Il contenuto di un pod è sempre co-locato e co-schedulato, e viene eseguito in un ambiente condiviso. Contiene una o più applicazioni che sono relativamente connesse tra loro.

**Kubelet**: il node agent primario che esegue ogni codice. Lavora in termini di PodSpec(oggetto che descrive un pod). Prende set di PodSpec che gli vengono forniti da vari meccanismi e assicura che i containers descritti in quesi PodSpecs siano in esecuzione senza problemi. I kubelet non gestiscono i container che non sono stati creati in kubernetes.

**DockerSwarm vs Kubernetes VIDEO UTILE EXTRA MIO**

| Docker Swarm | Kubernetes |
|---|---|
| ▪ Simpler to install<br>▪ Softer learning curve | ▪ Features auto-scaling<br>▪ Higher fault tolerance<br>▪ Huge community<br>▪ Backed by Cloud Native Computing Foundation (CNCF) |
| Preferred in environments where simplicity and fast development is favored | Preferred for environments where medium to large clusters are running complex applications |

## 6° LEZIONE incontro azienda: ION GROUP, Cloud AWS Essentials

(Non ho preso appunti, è tratto tutto dalle slide)

*[ancora mancante]*

---

## 7° LEZIONE: Datacenter

Entriamo dentro i datacenter

**SAP datacenter [VIDEO]**

I datacenter sono strutture abbastanza complesse i cui server rappresentano solo una piccola parte. Ci sono generatori di energia, **batterie** in caso di interruzioni di energia, unità di raffreddamento, connessioni internet, sezioni per emergenze come incendi con compartimenti tagliafuoco (isolati), telecamere di sicurezza, porte di sicurezza, backup dei dati in un'altra sede.
I datacenter di SAP **si preoccupano anche della salvaguardia dell'ambiente** adottando strategie per ridurre l'inquinamento.

**Google datacenter [VIDEO]**

Spazi per il divertimento dei dipendenti, diversi livelli di sicurezza tipo scanner dell'iride o laser sotto i pavimenti.
Energia distribuita in modo "Overhead" (dall'alto)
Tecnologia di raffreddamento: dentro c'erano 27 gradi, più alta degli altri, di solito si tiene più bassa, hanno un sistema di raffreddamento con delle torri e dei tubi di rame in cui scorre l'acqua. **Anche Google cerca di ridurre l'emissione di CO2.**
Quando cambiano i drive, perché hanno smesso di funzionare, dopo averli sostituiti li distruggono.

**Datacenter Ateneo**

Il Datacenter di ateneo è a San Piero a Grado e a Pisa ci sono 3 nodi DCI + 1 IT
**100 gbit/sec** velocità ateneo
Il traffico viene chiamato east-west (quello che rimane dentro la rete datacenter) e north-south all'esterno.
**No Single Point of Failure**: per il traffico east-west usa una Spine-leaf topology, cioè una topologia a due livelli, ridondante per ridurre al minimo i punti di fallimento.
La tecnologia per il raffreddamento è quello **adiabatico,** permette di avere un indice **PUE<1.3** , è uno degli indici per misurare l'efficienza energetica, deve essere sempre maggiore di 1 ma più vicino possibile a 1.

PUE = Total Facility Power/IT Equipment Power

**Adiabatic cooling [VIDEO]**

Meccanismo alternativo per ridurre il consumo energetico

**DCIM Data center infrastructure management**

Le azioni principali per gestire un datacenter sono il planning, quindi meeting, organizzazione ecc., installare e gestire le "pile" di server, gestire le connessioni e mantenere il **cablaggio pulito**. Per monitorare i datacenter si ha anche una **visione grafica (DCIM)** dove controllare l'energia, sicurezza, ambiente e raffreddamento attraverso grafici e reports con notifiche immediate in caso di errori o fallimenti.

Un altro aspetto importante è quello della **documentazione**, spesso non si ha una documentazione in linea, bisogna invece mantenere una documentazione costante.

**NOT to do in data center:**
- Gestione disordinata del cabling
- Non introdurre cibo o bevande all'interno datacenter
- Non tralasciare la documentazione
- Non lasciare la porta aperta del datacenter tra un accesso e l'altro, neanche per il trasporto di oggetti all'interno
- Prevenire click accidentali di interruttori

**Cosa fa il personale del datacenter? Come reagisce di fronte ad un problema?**
1. Renditi pronto immediatamente
2. Gestisci il panico
3. Segui la **checklist**.

Le checklist servono sia all'operatore che al gestore, altrimenti ognuno farebbe delle cose diverse.

**Business continuity & disaster recovery:**
1. Avere una copia degli stessi dati LONTANA dall'altra
2. Il piano di evacuazione/incendio va prima testato
5. Avere personale preparato
6. Avere piani di recupero nel caso il primo piano fallisca

**I più strani disastri mai accaduti:**
Un fulmine ha colpito il datacenter di Amazon e Microsoft
Uragano Sandy
Un camion ha sfondato un datacenter
Degli scoiattoli hanno mangiato i fili
Una nave ha calato un'ancora sui fili comunicanti col datacenter sotto il livello del mare

**Hyper-converged**

La struttura **tradizionale** di un datacenter è di avere server, networking e storage separati tra loro e poi montati assieme, questo può causare incompatibilità tra le componenti.
La struttura **"converged"** invece dà subito un insieme di componenti pre-testate e validate con un sistema di gestione integrato con cui poter eseguire le operazioni principali.
La struttura **hyper-converged** invece sfrutta la virtualizzazione: combina la virtualizzazione del server con il networking e lo storage in una singola scatola, quindi riduce di molto i costi hardware ma ha alcune limitazioni (fino al 2016): non si può aumentare lo storage senza aumentare anche la potenza di calcolo, non sono supportate tutte le applicazioni, e i relativi software non sono economici.

---

## 8° LEZIONE: Incontro con azienda TD GROUP [NON SISTEMATI]

Microservizi:
Scelti per ridurre tempi di sviluppo, anche piccole aziende possono averli.
Progettare a microservizi:
Quanto piccoli? Da riscrivere in 2 sett. se nò lo risuddivido con 3 strategie:
- Casi d'uso
- buonded context: funzionalità autocontenute
- srp come i comandi unix una cosa sola

REST API per far comunicare i DB e i microservizi
Per mantenere i dati integri devo farlo a livello di applicazione non direttamente in sql

Docker swarm per la scalabilità, gestire ogni microservizio dentro container
l'immagine docker è un minisistema operativo, piccolo linux, messi poi sul server del cloud privato.
Team diversi, full stack developer
node.js, mongodb, angular 2
sviluppatore scrive codice->lo carica su gitlab->Jenkins scarica e compila il codice sorg.->crea un'immagine docker->lo deploya su docker swarm

Spedire le PEC
Uno che guarda se ci sono delle pec inviate, un microservizio che controlla il nome all'anagrafe nel DB, un altro microserv. che invece scarica gli allegati mongoDB

---

## 9° LEZIONE: Introduzione al Green Computing

L'ICT genera inquinamento e gas serra, ci vogliono **1800 kg per produrre un pc,** circa 2 tonnellate.
Il **9% di consumo elettrico in Europa** per l'ICT e il **4% per le emissioni CO2**

Il consumo energetico Datacenter è dato maggiormente dal meccanismo di raffreddamento che rappresenta il **40%.**

### Il problema del PUE

PUE = potenza necessaria per far funzionare tutto / potenza solo per L'it
Il PUE misura l'efficienza energetica ma non misura il grado di utilizzo delle energie rinnovabili.
Con energie rinnovabili si intendono le fonti di energia che si rinnovano nel corso della vita di un essere umano (onde, vento, sole, pioggia ecc.)

### Obsolescenza programmata

Si definisce Obsolescenza programmata quando un prodotto è designato per avere un ciclo di vita breve. Il ciclo di vita del prodotto non deve essere troppo breve, sennò il cliente pensa che non sia buono, bisogna fare in modo che duri abbastanza affinchè il cliente sia contento e che pensi che sia di buona qualità. **È una delle cose che produce di più e-waste.**

Questo concetto è nato nel 1925 con i produttori di **lampadine**, i quali si sono messi d'accordo per abbassare la vita di una lampadina da 2500 a 1000 ore, così che tutti i produttori di lampadine guadagnassero di più.
Un altro esempio sono le **cartucce** che vengono rilevate finite dalla stampante quando non lo sono ancora.
Ma anche le **batterie non removibili** così che quando non funziona più la batteria si cambia tutto.
**Entrate plug diverse**: se si cambia il plug dobbiamo cambiare anche i caricabatterie.
**Update** 2018 Apple e Samsung che una volta installati creavano problemi di batteria e funzionamento appositamente.

### Obsolescenza percepita:

Con obsoloscenze percepita ci si riferisce a quando un cliente è convinto che ha bisogno di un nuovo prodotto anche se il prodotto che ha sta funzionando bene. La nuova versione del prodotto deve essere **visibilmente differente**, si deve vedere che io ho un telefono vecchio e l'altro ha un telefono nuovo.

Gli ambiti in cui si sente molto l'obsolescenza percepita è il mondo degli **smartphone** e delle **automobili**, si cerca di spingere il cliente a comprare di più convincendolo che "più compri più sei figo", "I shop so I exist".

### e-waste

Ogni anno si producono **50 milioni di tonnellate di e-waste** = 770 milioni di lavatrici
il traffico di e-waste è più grande del traffico di droga, l'esportazione di e-waste è stata resa illegale da 30

anni ma si continua a praticare, danneggiando fisicamente le popolazioni più povere che lavorano in condizioni ambientali pessime respirando fumi potenzialmente cancerogeni.

## Report di Greenpeace: clicking clean

Report in cui GreenPeace mette in evidenza la sostenibilità ambientale ICT e mette a paragone arrivare a 100% energia rinnovabile oppure vedere dei pericolosi cambiamenti ambientali, vorrebbe spingere le più grandi aziende ad arrivare al 100% delle energie rinnovabili come Apple Facebook e Google, che hanno deciso di aderire al progetto. Perché?
Perché se prima l'energia rinnovabile costava tanto **ora costa meno**, l'altro motivo sono i clienti, se ai clienti non fregasse niente dell'ambiente non lo avrebbero fatto ma se invece **i clienti sono interessati** allora si.

**Ci sono dei problemi:**

- mancanza di trasparenza: GreenPeace fa un report e i dati vengono forniti dalle aziende, ci sono aziende che lo fanno in modo trasparente altre molto meno dando dei dati non veritieri o incompleti, come Amazon che ha aumentato il numero di datacenter in Virginia.
- mancanza di accesso alle energie rinnovabili (RE): in cina solo 5% RE, corea del sud 1.1%, Virginia Stati Uniti con molti datacenter ancora meno 1%.

il report di GP valuta le energie rinnovabili al contrario del PUE.

**Conclusioni di questo studio:**
**Apple: leader delle energie rinnovabili**
**Apple e Google**: sono i migliori delle energie rinnovabili
Amazon: **non trasparenti**, non veritieri, grande crescita in Virginia con tantissimi nuovi datacenter.

Dati parzialmente positivi perché quelle poco trasparenti sono grandi aziende.

## Le 3 azioni principali da portare avanti

### Minimizzare il consumo di elettricità:

- Nel proprio piccolo è bene limitare il numero di dispositivi in standby in casa.
- Per i datacenters migliorare il **raffreddamento** è l'obiettivo principale, il migliore è il raffreddamento liquido.
- Fortunatamente ci sono diverse iniziative per promuovere il risparmio energetico a livello globale.

### Rendere più efficiente lo sfruttamento dell'energia elettrica dei dispositivi

- Cercare di designare **Hardware efficienti**, magari con le nanotecnologie.
- Per la parte **Software** cosa si può fare?
  Introdurre l'energia come un **requisito importante** nella produzione di software, ma anche nel testing, progettazione, manutenzione. **Minimizzare la comunicazione** tra processi/microservizi, i microservizi vanno contro a questo. Eliminare controlli, loop non necessari perché se **semplifico la struttura del mio software** ho meno da far lavorare però mi riduce il funzionamento del software.
  Avere applicazioni che si **adattano all'ambiente** se hanno meno energia a disposizione tipo luminosità che si abbassa.
  Soluzioni **approssimate** euristiche anziché una soluzione più lunga e precisa.

### Ridurre l'e-waste:

l'e-waste continua a crescere, continua il traffico e continua la generazione, ci sono però sempre più iniziative da parte di paesi e aziende per ridurlo.

## Conclusioni

Il Greencomputing è un po' in contrasto con gli obiettivi dell'ICT, se nel nostro software riduciamo il codice o la crittografia per far lavorare meno la macchina rischiamo di creare un'applicazione poco sicura, anche la scalabilità e la replicazione di istanze consuma più energia. Inoltre applicare tecniche per il risparmio energetico non conviene alle aziende a livello economico.

**GLOBAL CLIMATE PREDICTIONS**
**200 milioni di migranti** per problemi ambientali nel 2050
green policies: nel 2016 Parigi trattato per ridurre il riscaldamento globale, ma gli Stati Uniti nel 2020 a novembre **usciranno dall'accordo.**

---

## 10° LEZIONE: Internet of Things & Fog Computing [NON SISTEMATI]

VALUTARE L'IMPATTO AMBIENTALE DELL'IOT
Internet of things, 3 livelli sensore, come un misuratore della qualità dell'aria.
Ci saranno un sacco di datacenter in più.
Stimare l'impatto di un IoT comparandola a una soluzione idale, perché quando chiedi i consumi di qualcosa ti dicono solo "consumerai tot. di e-waste" ma non sai se è buono o meno.

Proposta green:
tutta l'energia generata da un pannello solare e una batteria
perché meno rifiuti quando dismessi.
I rifiuti vengono pesati in kg ma a volte ci sono cose tossiche che vanno valutate.

I circiuiti e ls schede rappresentano il 3% dei rifiuti
_____-

**Fog Computing**
IoT pervasivo, l'internet delle cose è ormai entrato a far parte del nostro mondo a pieno, applicazioni di domotica (automazione della casa).
orologi indossabili, frigoriferi, termostati, droni, guida autonoma, impianti produzione di energia, smart city come rescatame project.
E' in costante espansione.
**26 miliardi di dispositivi attivi nel 2019 con una crescita di circa 100 nuovi dispositivi al secondo.**
Molti dati da IoT sul Cloud, questo introduce la necessità di avere un'intermdiario tra il Cloud e l'IoT, **uno dei problemi principali è la latenza**, se c'è latenza in alcuni sistemi non succede niente ma ad esempio in una guida autonoma è importante.

**Così tanti dati?** Vediamo un esempio, compro una pianta e dei sensori per misurare terreno, acqua, temperatura, alcuni frequenza ogni min altri ogni ora, ci metto anche una videocamera.
transfer di 150 gb al mese circa e un cloud storage di 440 gb al mese

Deployment models, 2 modelli:

1. **IoT + Edge**
   (più vecchio) sulla stessa rete locale, non c'è latenza ma problemi di capabilitis perché non abbiamo potenza infinita del cloud, possiamo fare cose finchè non colmiamo capacità server. Inoltre c'è difficoltà nella condivisione dei dati fra applicazioni.
2. **Iot + Cloud**
   Opposto, potenza di calcolo illimitata ma connettività a internet obbligatoria.

**Fog Computing:**
Infrastruttura che sta in mezzo tra IoT e Cloud per migliorare la latenza e la bandwidth hungry. Proprio per supportare il tipo di applicazioni in cui la latenza è importante o dove c'è bisogno di avere sufficiente banda.

[VIDEO FOG COMPUTING]

Ci dà un'idea di quante applicazioni esistono che hanno a che fare col fog computing, dl controllo ambientale ai droni ecc.. Controllo a metà strada tra IoT e Cloud.
Diversi problemi: Privacy, Come fare il deployment delle app, Problemi di risorse e energia, problemi con l'infrastruttura e aspetti di business.

**Prendiamo il deployment**
E' difficile mandare in esecuzioni applicazioni composte in modo continuo sul Cloud tramite Fog.
Fare il matching tra i requisiti di un'appl. e l'infrastruttura Fog è difficile, perché eterogenea e non costante (tipo la rete può esserci di meno in un momento, poi di più ecc), **np-hard problem**

Esempio di un sistema di un edificio:
DataStorage: hardware large
Dashboard: hardware small
le frecce sono i vincoli di latenza e banda che vorremmo soddisfatte.
Infrstruttura fog:
Casino per effettuare il matching corretto ed efficiente, un'altra dimensione di complessità è che la rete è dinamica, quindi la qualità del servizio in termini di latenza e banda non è costante nel tempo, abbiamo un comportamente che possiamo esprimenere in maniere probabilistica, sappiamo che per le maggiorpart del tempo la altenza media sarà di tot millisecondi ma sappiamo che per altre cose la latenza sarà inferiore.

Come faccio a decidere qual è il miglior modo di posizionare le componenti e i nodi ecc.?Il problema è complicato, servono degli strumenti, sarà un problema di ricerca operativa con molti vincoli: coso, Qualità del servizio, sicurezza, consumo risorse fog.

Una soluzione a questo problema è fogtorchPI.
algoritmi + Monte Carlo simulation per gestire la variabilità nel tempo, per generare simulazioni diverse. + modello di costo.
Grafico 3d
Per sapere anche se mi conviene fare l'upgrade della rete, riduco i costi o riduco il consumo di risorse? Senza un modello che lo calcola sarebbe difficile.

Conclusioni:
Grande hype di interesse per il fog computing

---

ULTIMA LEZIONE [NON SISTEMATI]

L'ultima volta abbiamo introdotto il Fog computing.
Cioè un continuo di infrastruttur tra il cloud e l'internet delle cose.

**PROLOG**
Modo dichiarativo per coreFogTorchPI
Dichiarativo in prolog, insieme di regole e di fatti
questo programma posso interrogarlo, posso chiedere se è vero nice(W), cioò istanziare W in modo che sia conseguenza logica di quei fatti.
Albero che prolog utilizza per fare questa dimostrazione, sottolineato perché viene istanziata la prima regola. da nice va a honest, da honest guarda il primo gentle, poi dà risposta e torna indietro e fa quello che c'è dopo cioò gentle di barbara e fine.

Infrastruttura = insieme di fatti, requisiti infrastr
%node Nome del nodo, software capabilities, quanti hardware capabilities, quali dispositivi IoT raggiunti)
node (fog1 è il nome, (linux, php..), 2, (sensore fuoco))
…
%link latenza end to end tra ogni possibile coppia di nodo
link(fog1, fog2, 15) la latenza che vogliamo tra i due fog è di 15

applicazione, requisiti appl
Nome dell'appl + servizi (stesse cose di prima)
Anche qui latenza

Placing Services
placement(App, Placement) se app è un'app che contiene servizi e se placemente è un serviceplacement per questi servizi e flowsOK se rispetta i miei vincoli.

in grassetto: Quando un servizio S può essre piazzato su un nodo N
se service è fatto così (va a prendere il primo serviz.)
poi va a prendere il primo nodo e poi possiamo piazzare s su N se i requisiti SW sono soddisfatti dalle capability e anche HW e dall'IoT.
Per gli HW se un nodo offre linux possiamo piazzare anche 2 servizi con linux.
subset: è un sottoinsieme
Va a cercare tutti i possibili piazzamenti

pag dopo
oneServicePlacement se ha successo piazza S su N  costruisce il  secondo parametro del servicePlacement.

flowsOK dato un placement prende tutti i srvizi e controlla se la latenza tra i servizi rispetta i vincoli, se sono sullo stesso nodo la latenza è 0 quindi non si controlla. Se la FeaturedLatency =< latenzaRichiesta allora va bene.

Ora possiamo fare un'interrogazione:
Come piazzare smartHome su P?
Viene messo dashboard su cloud1, datastorage su cloud1.

Aggiungere probabilità
ProbLog, vengono associate probabilità ai fatti, diciamo che il primo profilo del fog1 è quello che succederà nel 90% dei casi, il secondo, quello meno potente dove non si può accedere alla videocamera, accadrà nel 10%.
Allo stesso modo con la latenza

Ci restituisce una probabilità
Per il primo placement nel 70% dei casi i vincoli di latenza saranno soddisfatti.

Usati algoritmi genetici, su fogtorch generiamo prima due insiemi di deployment, uno per la rete nella miglior possibile situazione e una pggiore. Poi si simulano con monteCarlo.

----------FINE PROLOG----------

PROGETTO Giò che il prof sta seguendo
Tester per il Fog per sviluppare attività di ricerca e insegnamento.
Obiettivo di poter fare ricerca e sperimentare suoluzioni SW e HW diverse per smart ambiens tipo domotica ma per ambiente di lavoro e uni, con sostenibilità ambientale.
Giò Plant servizio che fa monitoraggio su una pianta ee riesce a rilevare umidità, °C, e la irriga quando serve.
GiòShader per aumentare e diminuire la luce in base anche alla temperatura per impattare meno sull'ambiente.
GiòRoom orchestra i due.
GiòMap mappa dipartimento digitale, ci dice come raggiungere un punto o una persona.
Alexa e Google Mini Home per interagire col sistema.
---

---

AZIENDA EXTRA RED [NON SISTEMATI]

Anche loro usano Jenkins, codice sorgente in pacchetti e(container) e poi su Docker.
Ansible: configurazioni + container si occupa deel deployment.
Flusso automatizzato

Servono strumenti flessibili e scalabili, ci vuole un'infrastruttura Cloud, se è tutto all'interno di un Cloud meglio.

Sul srver hardware ci sono diversi livelli di virtualiz.
Il primo è OpenStack che mette risorse astratte esposte ai livelli superiori. In un Cloud separato c'è il backup e monitoring.
Tutto questo ci dà istanze (macchine virtuali, RAM)
+ OpenShift che costruisce un altro strato di astrazione, hanno aggiunto a Kubrnetes delle funzionalità e ci hanno fatto OpenShift
Cosa fa OpenShift?
Offre una serie di servizi pr orschestrare containr, semplificano la vita.
POD = contenitore di container, permett di creare all'interno dello stesso POD dei container che si spartiscono il lavoro, opnShift permette di scalare questi POD.
Un'altra caratteristica di OpenShift è la sicurezza in modo che i container non si disturbino tra loro, tra cluster diversi.
CLUSTER: insieme di Nodi, ogni Nodo macchina virtuale.

artefatto deploybile: un eseguibile + il container che lo impacchetta.

Si possono lanciare i microsrvizi in maniera isolata, l'output della prima pipline è un artefatto eseguibile. La seconda pipeline è quella in cui dato un repository con tutti gli artefatti finiti è una fase in cui li mettiamo tutti nello stesso ambiente di produzione e facciamo i test più complessi.
Software usati per fare Continuos Integration Pipeline:
Git ->Jenkins(+Maven+sonarqube)->Repository (Jfrog Artifactory eQuay)
Software usati per seconda fase:
Jenkins: è in grado di accettare script (da Git) con step successivi e si interfaccia con Ansible per deploy automatico dei container.
Ansibl riesce a gestire installazione applicazione su un grande numero di host.

Siamo riusciti a minimizzare le dipendenze? Se tante persone usano Jenkins contemporaneamente possono crearsi dei problemi con l configurazioni, i plugin possono andare in conflitto e va reinstallato. Dobbiamo trovare un sistema pr rendere i team autonomi.
Soluzion:
OpnShift riesce a creare spazi di lavoro separati, ognuno coi propri strumenti. Ci sono docker interni a openshift e altri esterni.

Si cerca di automatizzare la creazione di questi spazi di lavoro.