CLOUD COMPUTING

Il cloud computing indica, in informatica, un paradigma di erogazione di servizi offerti su richiesta da un fornitore a un cliente finale attraverso la rete internet (come l'archiviazione, l'elaborazione o la trasmissione dati), a partire da un insieme di risorse preesistenti, configurabili e disponibili in remoto sotto forma di architettura distribuita. A seconda del livello di personalizzazione che si lascia all'utente o in generale, dal tipo di risorse che gli si lasciano gestire, si creano dei modelli diversi di cloud computing che vedremo nel dettaglio più avanti. Le risorse non vengono pienamente configurate e messe in opera dal fornitore appositamente per l'utente, ma gli sono assegnate, rapidamente e convenientemente, grazie a procedure automatizzate, a partire da un insieme di risorse condivise con altri utenti lasciando all'utente parte dell'onere della configurazione. Quando l'utente rilascia la risorsa, essa viene similmente riconfigurata nello stato iniziale e rimessa a disposizione nell'insieme condiviso delle risorse, con altrettanta velocità ed economia per il fornitore.

Il sistema del cloud computing prevede tre fattori distinti:

- **fornitore di servizi**: offre servizi di server virtuali, archiviazione, applicazioni complete (per es. base di dati) generalmente secondo un modello pay per use (PPU);
- **cliente amministratore**: sceglie e configura i servizi offerti dal fornitore, generalmente offrendo un valore aggiunto come per esempio applicazioni software;
- **cliente finale**: utilizza i servizi opportunamente configurati dal cliente amministratore.

Il termine *cloud computing* si differenzia però da *grid computing,* che è invece un paradigma orientato al calcolo distribuito e, in generale, richiede che le applicazioni siano progettate in modo specifico.

I problemi e le criticità generiche si possono raccogliere sotto quattro aspetti principali:

1. sicurezza e privacy degli utenti
2. problemi internazionali di tipo politico ed economico
3. continuità del servizio
4. difficoltà di migrazione dei dati nel caso di un cambio di gestore di servizi cloud

Service-based economy

Il trend dell'economia si sta spostando dall'acquisto di beni ai sempre più frequenti acquisti di 'servizi'. In generale il consumatore non sa (e non è interessato a sapere) come il servizio che utilizza è implementato. Il consumatore si limita spesso a decidere se utilizzare o meno il servizio sulla base del contratto che gli viene offerto. In questa ottica, quello che si evince, è che la qualità del servizio sicuramente è importante, ma non è l'unico fattore determinante.

Service Level Agreements (SLA)

I service level agreement (in italiano: accordo sul livello del servizio), in sigla SLA, sono strumenti contrattuali attraverso i quali si definiscono le metriche di servizio (es. qualità di servizio) che devono essere rispettate da un fornitore di servizi (provider) nei confronti dei propri clienti/utenti. Di fatto,

una volta stipulato il contratto, assumono il significato di obblighi contrattuali. La qualità del servizio comprende anche le garanzie offerte dai contratti e l'affidabilità del provider.

Viene scritta da tre diverse figure: avvocato specializzato nel settore, business expert e sviluppatore che indica le caratteristiche del servizio. 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 affidabilità potrebbe essere più alto. Esempi:

- Cloud Amazon S3, da nessuna parte 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, non garantisce neanche che il Cloud sia sicuro, anzi ci invita a fare regolarmente un backup dei nostri dati del Cloud su un'altra piattaforma.
- Facebook, 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.

Stima della domanda del servizio

La domanda di un servizio, cambia col tempo, ma è impossibile sapere come con esattezza. Si possono soltanto fare delle stime di mercato. Quello che può succedere con le stime, a volte porta a due estremi:

- **OVERPROVISIONING:** garantire un approvvigionamento a priori delle risorse per i picchi stimati (siano essi giornalieri o mensili o annuali) porta ad uno *spreco di risorse* anche nel caso in cui la stima sia corretta, peggio ancora nel caso in cui i picchi non sono nemmeno riscontrati;
- **UNDERPROVISIONING:** la sottostima del picco di utilizzo, può causare nel caso peggiore un rifiuto di accesso al servizio per alcuni utenti, in quanto mancano risorse per gestirli tutti insieme. Questo può tramutarsi in un danno paragonabile alla sovrastima: gli utenti rifiutati, per cominciare non portano guadagno e, oltre a questo, potrebbero decidere di non usare più il servizio e influenzare altri utenti nella scelta.

Sotto questi due grandi problemi nasce il cloud, che si impone per cercare di evitare una delle due situazioni e rendere servizi con risorse potenzialmente illimitate, facilmente scalabili e solamente su richiesta.

Definizione del NIST del 'Cloud Computing'

"... un modello per consentire ovunque, in modo comodo e *on-demand* l'accesso, attraverso la rete, ad un pool condiviso di risorse di calcolo (ad esempio, reti dedicate, server, storage, applicazioni e servizi) che possono essere rapidamente messi a disposizione con un minimo sforzo di gestione da parte dei provider."

Idee chiave:

- Pool efficiente di infrastrutture virtuali, accedibile on-demand e utilizzabile come servizio;
- Risorse virtuali scalabili dinamicamente e fornibili a più client;
- Divisione tra il servizio offerto e la tecnologia che lo implementa.

Dal punto di vista economico:

- Eliminazione dell'impegno economico iniziale da parte del consumatore (la spesa di capitale viene trasformata in spesa operativa)
- Modello Pay-per-Use
    - Molto apprezzato in generale dai consumatori
    - Anche se il pay-per-use è generalmente più costoso, il prezzo è bilanciato dall'elasticità che tale modello consente al cliente di avere.

Service Models

- **SaaS:** (*Software as a Service*) fornisce software on-demand per l'utilizzo, utilizzabile attraverso thin-client oppure APIs. Il provider SaaS gestisce l'infrastruttura, l'OS e le applicazioni, rendendo così il cliente responsabile di niente.
  Esempi: Salesforce.com, Google Drive.
- **PaaS:** (*Platform as a Service*) fornisce l'intera piattaforma come servizio, ad esempio il Sistema Operativo, varie VMs, ma sarà il cliente a essere il responsabile dell'installazione e della gestione delle applicazioni.
  Esempi: Heroku, Azure, GAE.
- **IaaS:** (*Infrastructure 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 (per esempio sistema operativo, applicazione). Il mercato del Cloud IAAS `e cresciuto del 31.3% nel 2018.
  Esempi: EC2 (Elastic Compute Cloud), S3 (Simple Storage Cloud), entrambi di AWS.

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 se succede qualcosa nei datacenter di Amazon, i nostri dati sono violati)
- **Privato**, non ha molta scalabilità ma risulta più controllato e sicuro
- **Ibrido**, sono una via di mezzo

Ostacoli all'adozione del cloud

- **Confidenzialità dei dati:** Dove vengono stoccati concretamente i dati? Come sarà garantita la privacy e l'integrità dei dati? Come potremo sapere quando ci saranno problemi?

- **Disponibilità del servizio:** Cosa succede se il provider dei servizi cloud interrompe il servizio? Cosa succede se il servizio resta offline per un certo lasso di tempo? Dobbiamo anche mettere in conto che ci saranno dei fallimenti temporanei e il servizio potrebbe non essere disponibile in alcuni momenti.
- **Vendor lock-in:** è una condizione che si verifica quando un cliente, diventa dipendente da un certo venditore, per materiali o servizi e si trova vincolato ad esso senza possibilità di cambiare fornitore se non aggiungendo ingenti quantità di denaro.

Confronto tra IaaS, PaaS e SaaS

Quale è esattamente la differenza tra IaaS, PaaS e SaaS? La risposta è che ognuno offre un servizio cloud diverso: un ambiente per infrastruttura, degli strumenti per piattaforma o applicazioni complete. A seconda del tipo di servizio che scegli, il provider di servizi cloud gestisce diversi elementi dello stack di computing:

- ***Infrastructure as a Service (IaaS):*** per i modelli IaaS, il provider di servizi ospita, gestisce e aggiorna l'infrastruttura di backend, ad esempio computing, archiviazione, networking e virtualizzazione. Tu gestisci tutto il resto, inclusi il sistema operativo, il middleware, i dati e le applicazioni.
  Esempi di IaaS: Compute Engine, Cloud Storage.

- ***Platform as a service (PaaS):*** Come i modelli IaaS, per i modelli PaaS, il provider di servizi fornisce e gestisce l'infrastruttura di backend. Tuttavia, i modelli PaaS forniscono tutte le funzionalità e gli strumenti software necessari per lo sviluppo delle applicazioni. Sta a te scrivere il codice e gestire le app e i dati, ma non devi preoccuparti della gestione o della manutenzione della piattaforma di sviluppo software.
  Esempi di PaaS: Cloud Run, App Engine.

- ***Software as a Service (SaaS):*** Con i modelli di servizio SaaS, il provider di servizi fornisce l'intero stack di applicazioni, l'intera applicazione e tutta l'infrastruttura necessaria per la distribuzione. Come cliente, devi solo connetterti all'app tramite Internet: il provider è responsabile di tutto il resto.
  Esempi di SaaS: Google Workspace.

Iaas - (virtualizzatori di server e Hyepervisors)

IaaS (Infrastructure as a Service) è un modello di servizi cloud che offre risorse di infrastruttura on demand, come calcolo, archiviazione, networking e virtualizzazione, ad aziende e privati tramite il cloud. In particolare, **l'idea è di fornire l'infrastruttura tramite virtualizzazione delle risorse e on-demand**. Si ha quindi disponibilità on-demand di risorse di calcolo altamente scalabili come servizi su Internet. In questo modo, le aziende non hanno la necessità di procurarsi, configurare o gestire l'infrastruttura autonomamente e pagano solo per quello che utilizzano. Non c'è quindi necessità di costruire e gestire data center o server farm che sono spese onerose sia per la costruzione che per la manutenzione, ma si rimanda tutto al fornitore.

Definizione di virtualizzazione:

la virtualizzazione è un livello di astrazione nell'architettura di una macchina, in particolare:

- L'OS è astratto dall'hardware, non direttamente installato sulla macchina
- L'OS non è più residente sull'hardware

Virtualizzazione del server:

- Livello di virtualizzazione tra server fisico e l'OS normalmente installato
- Macchina virtuale: dove si installa l'OS e le applicazioni, un livello sopra la virtualizzazione del server

Definizione di Hypervisor:

Hypervisor è una funzione che crea il livello virtuale e gestisce le macchine virtuali tramite il Virtual Machine manager. Esistono due tipi di hypervisor, con alcune differenze:

- Il tipo1 gira direttamente sull'hardware, il tipo2 gira su un sistema operativo installato sull'hardware, come se fosse una normale applicazione.
- Il tipo2 ha un overhead maggiore e dunque un *consolidation ratio* inferiore (il numero di server virtuali che può girare contemporaneamente su una macchina fisica).
- Il tipo1 è impiegato nei Data Center, il tipo2 è tipico dei PC di uso comune (vedi VirtualBox).

Esempio di IaaS/compute: Amazon Elastic Compute Cloud (EC2) (per server virtuali)

- Mette a disposizione server virtuali (*istanze*) in modo semplice, veloce ed economico.
- Scelta del tipo di istanza e template da utilizzare (Windows/Linux) e scelta del numero istanze tramite *AWS Management Console* (o librerie SDK).
- Ampia gamma di istanze, con tipologie di risorse in diverse quantità (CPU, memoria, storage, GPU).
- Opzioni di pagamento diversificate:
    - on demand
    - istanze riservate (pagamento mensile, si usa quando sappiamo già quali e quante macchine mi serviranno)
    - istanze spot (sono istanze non utilizzate che Amazon mette in vendita ogni 5 minuti all'asta, di solito costano il 50% in meno).
- Sicurezza (Virtual Private Cloud - VPC) .
- Storage persistente: Amazon Elastic Block Store (EBS) (ogni volume viene replicato da Amazon) .
- Funzionalità di autoscaling

Esempio di IaaS/storage: Amazon Simple Storage Service (S3) (per stoccaggio di dati)

- Fornisce uno storage sicuro e facile da usare, senza dover sapere a priori quante unità di stoccaggio mi serviranno.
- Interfaccia semplice, modellizzata tramite un drag and drop in un "Bucket".
- 3 backup su 3 diverse unità per evitare la perdita dei dati.
- Diverse classi di memorizzazione:
    - **S3 Standard**: per dati ai quali l'accesso viene effettuato spesso
    - **S3 Standard infrequent access**: per dati ai quali si accede raramente
    - **Amazon Glacier**: per dati ai quali generalmente non si accede (come file di backup)
- Controllo configurabile accesso ai dati, in base alle classi di utenti

- Pagamento in relazione ai GB utilizzati (piano gratuito da 5GB).

La storia di Dropbox

È un servizio di file hosting che offre memorizzazione Cloud, sincronizzazione dei file e strumenti di collaborazione. Nasce con l'obiettivo di fornire gratuitamente spazio di archiviazione con limitazioni, e chi vuole può acquistare l'account premium per avere più capacità di memorizzazione. A marzo 2018 Dropbox contava più di 500 milioni di utenti.

Per 8 anni dalla sua nascita, Dropbox ha utilizzato Amazon S3 per stoccare milioni e milioni dati, ma tra il 2014 e il 2016 ha costruito la sua larga rete di computer e ha spostato il suo servizio sulle nuove macchine progettate appositamente dai propri ingegneri.

La sfida tecnica e logistica è stata notevole:

- Ogni 'Diskotech', ovvero il componente progettato dagli ingegneri di Dropbox, contiene un Petabyte.
- Nuovo codice: "Magic Pocket", che inizialmente non funzionò sul nuovo hardware.
- Spostamento di una grande quantità di dati: in un giorno si potevano trasferire circa 4 Petabyte tramite internet, .
- Necessità di installare 40-50 rack ogni giorno in diverse zone del paese
- Necessità di completare l'esodo entro un data specifica, per evitare il rinnovo del contratto con Amazon S3
- Cercare di non causare disagi e malfunzionamenti agli utenti durante la migrazione.

Secondo Forbes, tuttavia, la scelta era giustificata e le entrate di Dropbox per il 2022 sono viste al rialzo. Tra i pro della migrazione ci sono sicuramente il vantaggio di non dover più pagare Amazon per i servizi, avere molta più autonomia e offrire servizi di supporto più veloci ai clienti. Tra i contro c'è la gestione della infrastruttura per lo stoccaggio dei dati, i costi dello sviluppo del proprio hardware e software e i rischi dovuti alla migrazione dei dati.

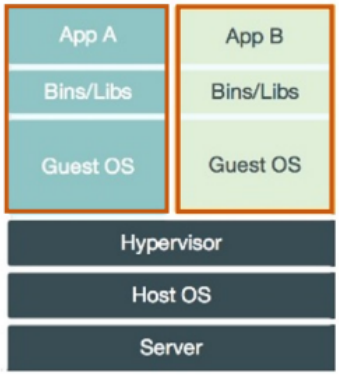
Containers e il loro utilizzo nella virtualizzazione

Come visto per IaaS, all'interno delle macchine virtuali si possono avere pi`u applicazioni in esecuzione all'interno dello stesso server fisico e per ogni macchina virtuale, il server alloca le risorse in termini di CPU, memoria, spazio di storage e l'installazione del sistema operativo ospite. Il nucleo dei sistemi operativi permette di avere più istanze isolate, che vengono chiamate Cointainer.

I container sono pacchetti di software che contengono tutti gli elementi necessari per l'esecuzione in qualsiasi ambiente. In questo modo, i container virtualizzano il sistema operativo e sono eseguibili ovunque, da un data center privato al cloud pubblico o anche sul laptop di uno sviluppatore.

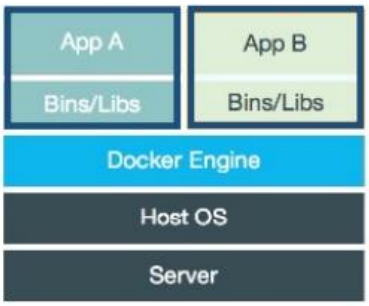
Docker: è una piattaforma che consente di runnare applicazioni in un ambiente isolato. Docker consente quindi di sviluppare applicazioni e far girare applicazioni di tipo "portable" sfruttando i containers. Quello che succede che al posto dell'hypervisor viene montato il cosiddetto *Docker engine*. Quest'ultimo permette di creare i Container che contengono le applicazioni e le sue dipendenze.

Virtual machines



Applications run in Virtual Machines
Various applications can run on the same physical server
Each VM requires
- resource allocation (CPU, memory, storage)
- installing a Guest OS

Container



The kernel of the operating system allows the existence of multiple isolated user-space instances. Such instances are called ***containers.***

Facendo una comparazione veloce, si può dire che in generale:

- I containers sono PIU' leggeri (dal punto di vista di risorse hardware richieste)
- I containers sono PIU' veloci nella fase di avvio
- I containers sono PIU' semplici da implementare
- I containers sono MENO sicuri rispetto alle macchine virtuali (hanno meno isolamento)

La piattaforma Docker, definitivamente lanciata nel 2013, si compone di due elementi:

- Docker Engine: per creare e runnare i containers
- Docher Hub: per la distribuzione dei containers

Docker sfrutta quindi la virtualizzazione basata sui containers per runnare istanze multiple e isolate tra di loro di ospiti sullo stesso sistema operativo. I Container sono volatili ovvero non tengono traccia dei dati una volta terminata l'esecuzione, per questo motivo vi è un altro concetto di *volume*, che è il meccanismo utilizzato da Docker per memorizzare i dati in modo persistente. Le applicazioni

sono impacchettate in “immagini”, che sono dei file di sola lettura utilizzate per creare e far girare containers.

Docker Images:

- Sono templates di sola lettura usati per creare containers
- Salvate in un registro (pubblico o privato) di Docker:
    - Registro strutturato in repository
    - Ogni repository contiene un set di immagini per differenti versioni di uno stesso software

Le immagini vengono identificate dalle coppie {repository , tag}, e sono strutturate in livelli: a livello più basso vi è il *base image*. Il Container andrà in esecuzione in cima a questa pila di livelli, e a seconda del tipo di applicazione potrà effettuare delle modifiche che potranno a loro volta essere ‘committate’ in una nuova immagine. I comandi principali di Docker sono:

- **pull**, scarica un’immagine da un Docker registry
- **run**, manda in esecuzione il Container definito dall’immagine
- **commit**, per creare ed effettuare delle modifiche che possono essere salvate in una nuova immagine
- **build**, permette di specificare la costruzione dell’immagine in un file chiamato *dockerfile*

Modalità Swarm

La modalità Swarm è una funzione di Docker che fornisce funzionalità di orchestrazione dei contenitori predefinite, inclusi il clustering nativo di host Docker e la pianificazione dei carichi di lavoro dei contenitori. Un gruppo di host Docker forma un cluster "swarm" quando i relativi motori Docker vengono eseguiti insieme nella "modalità swarm".

Uno swarm è costituito da due tipi di host contenitore:

- **nodi di gestione (manager nodes),** che delegano e distribuiscono task
- **nodi del ruolo di lavoro (workers),** che eseguono soltanto

Ogni swarm viene inizializzato mediante un nodo di gestione e tutti i comandi dell'interfaccia della riga di comando di Docker per il controllo e il monitoraggio di uno swarm devono essere eseguiti dai uno dei relativi nodi di gestione. I nodi di gestione possono essere considerati come dei "guardiani" dello stato Swarm; insieme formano un gruppo di consenso che **mantiene il riconoscimento dello stato dei servizi in esecuzione nello swarm e il relativo compito è quello di garantire che lo stato effettivo dello swarm corrisponda sempre a quello previsto che viene definito dallo sviluppatore** o dall'amministratore.

PaaS

Platform-as-a-Service (PaaS) è un servizio di cloud computing in cui la piattaforma software applicativa viene fornita da terze parti. Pensato principalmente per sviluppatori e programmatori, il servizio PaaS offre una piattaforma su cui l'utente può sviluppare, eseguire e gestire applicazioni senza dover creare e gestire l'infrastruttura o la piattaforma normalmente associate a tali processi.

Le piattaforme PaaS possono essere eseguite nel cloud o su infrastrutture on premise. Nelle offerte gestite, **il servizio PaaS prevede che hardware e software siano ospitati nell'infrastruttura del provider**, che distribuisce la piattaforma all'utente come soluzione integrata, stack di soluzioni o servizio erogato tramite una connessione Internet.

I provider di terze parti gestiscono quindi hardware e software-tools per lo sviluppo delle applicazioni, mentre l'utente provvede solo a gestire i dati e l'applicazione (codice).

Vantaggi:

- Management dell'infrastruttura ridotto o nullo, così come della sicurezza
- Manutenzione automatica (a carico del provider)
- Caricamento del lavoro, scalabilità e distribuzione più semplici, perché infrastruttura è gestita dal provider
- Adozione di nuove offerte e tecnologie più rapida e semplice
- Creazione e sviluppo di prototipi molto veloce

Svantaggi:

- Se il servizio che mantiene l'infrastruttura va giù, non sarò in grado neppure io di fornire il mio servizio
- Dipendenza dal provider (vendor lock-in)
- Cambi interni al PaaS

Esempio di PaaS: Heroku

Heroku è una piattaforma cloud basata su un sistema gestito con containers, con servizi dati integrati e un potente ecosistema, per la distribuzione e l'esecuzione di applicazioni moderne.

La piattaforma Heroku utilizza dei containers detti "dynos", per runnare e scalare tutte le applicazioni. I Dynos sono containers Linux isolati e virtualizzati, progettati per eseguire codice in base a comandi specifici dell'utente. Il fatto di essere basata sui container risolve il grosso problema della portabilità: una volta giunti a destinazione, viene prelevata l'immagine del Container e ricostruita l'applicazione.

Alcune spiegazioni: https://www.fastweb.it/fastweb-plus/digital-magazine/cos-e-e-come-funziona-heroku-piattaforma-cloud-per-app/

I Dynos permettono una scalabilità molto flessibile, quindi a seconda delle richieste e delle risorse, l'applicazione può scalare arbitrariamente il numero di Dynos.

Funzionamento: applicazione riceve la richiesta che viene inviata a uno dei Web Dynos; la richiesta viene analizzata e messa in coda. Il Worker Dynos prende la richiesta e la soddisfa, se è necessario può memorizzare in maniera persistente il risultato in un database. 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.

Per sviluppare un'applicazione, al buildtime, Heroku ha quindi bisogno di tre soli elementi da parte del programmatore:

1. Codice sorgente

2. Una lista di dipendenze (quali sono le cose di cui ha bisogno per funzionare, per esempio delle librerie)
3. Un *'procfile'*: un file di testo che indica quale è il comando per l'inizializzazione dell'app

Il sistema di build automatico riceve il codice, recupera un buildpack (linguaggio, dipendenze, librerie, ...) e produce uno "slug", ovvero un insieme di risorse iniettato in un dyno che poi esegue l'applicazione. Il componente finale per eseguire l'applicazione è il sistema operativo che su Heroku è chiamato "stack" ed è un'immagine di Ubuntu mantenuta da Heroku.

Al runtime, quando si sviluppa una app, Heroku creerà automaticamente uno o più dynos, ognuno inizializzato con lo stesso stack e slug che rappresentano l'applicazione sviluppata. Infine, il dyno manager di Heroku lancia il comando contenuto nel procfile e farà così partire l'esecuzione della applicazione su Heroku. A seconda della potenza di calcolo richiesta, si possono comprare e gestire più dyno di tipologie diverse, che chiaramente hanno prezzi mensili a salire all'aumentare della loro potenza di calcolo e di RAM dedicata.

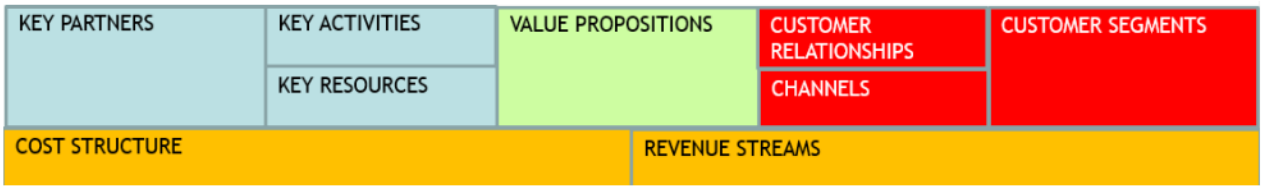
Heroku ha pi`u di 150 add-ons che gli sviluppatori possono utilizzare per estendere l'applicazione. Per esempio, servizio di autenticazione, log, monitoring, data stores, ... Tutto questo attrae l'utente perché ha la possibilità di integrare nuove funzionalità in un tempo molto breve. L'aspetto negativo è che questo ci lega all'utilizzo della piattaforma e instaura una forma di vendor lock-in rendendo la nostra applicazione pi`u difficilmente portabile. Infatti, se vorremo spostare la nostra applicazione su un nuovo servizio Cloud saremo costretti a rivedere tutto il codice perché gli add-ons non funzionerebbero.

Top 10 PaaS in 2022

- Amazon Web Services (AWS) Elastic
- Beanstalk
- Oracle Cloud Platform (OCP)
- Google App Engine
- Microsoft Azure
- Salesforce aPaaS
- Red Hat OpenShift PaaS
- Mendix aPaaS
- IBM Cloud Platform
- SAP Cloud Platform
- Engine Yard

Business Models

Un business model descrive in modo razionale come una organizzazione crea, distribuisce e cattura il valore. In questa ottica si è imposto un modello semplificativo ma di impatto per descrivere un modello di business: il business model canvas:



CUSTOMER SEGMENTS defines the different groups of people or organizations an enterprise aims to reach and serve

VALUE PROPOSITIONS describes the bundle of products and services that create value for a specific customer segment

CHANNELS describes how a company communicates with and reaches its customer segments to deliver a value proposition

CUSTOMER RELATIONSHIPS describes the types of relationships a company establishes with specific customer segments

REVENUE STREAMS represents the cash a company generates from each customer segment
(costs must be subtracted from revenues to create earnings)

KEY RESOURCES describes the most important assets required to make a business model work

KEY ACTIVITIES describes the most important things a company must do to make its business model work

KEY PARTNERS describes the network of suppliers and partners that make the business model work

COST STRUCTURE describes all costs incurred to operate a business model

Studiamo alcuni esempi di business model canvas

- Nespresso nel 1976 Nestlé dominava il mercato mondiale del caffè istantaneo con un prodotto che si chiamava Nescafé. Le macchinette Nespresso furono ideate nel 1976 quando nessun altro le faceva, ma non riuscivano ad entrare nel mercato con questo prodotto. Nel 1988 il nuovo CEO cambio il business model in particolare il *customer segment* poiché le macchinette non dovevano avere lo stesso mercato del Nescafé ma dovevano attrarre impiegati di alto livello e in generale famiglie benestanti.
  I canali di acquisto sono due: shop online o dalle boutique: solitamente nel centro della città vi è il Club Nespresso che, oltre a fornire dei vantaggi ai clienti, permette di tracciare le preferenze e l'attività dei clienti.
  Critiche: dal punto di vista della sostenibilità ambientale Nespresso produce una grandissima quantità di materiale difficilmente riciclabile
- Zara (no nelle slide) ha deciso di produrre abiti in base alla preferenza dei clienti effettuando un monitoraggio costante dei consumi degli utenti. Per questo motivo vi è un rinnovo costante della linea dei prodotti. Zara fornisce settimanalmente quello di cui ha bisogno un certo punto vendita, e così facendo nessun punto vendita utilizza un magazzino dove avere le scorte dei prodotti. I prezzi sono molto pi`u convenienti rispetto ai negozi di alta moda, ma i punti vendita vengono collocati vicini ad esse.

Il modello Freemium

Modello che si è diffuso negli ultimi anni per cui il servizio offerto è (di base) gratuito. Ci sono però dei vantaggi o dei servizi aggiuntivi/miglioramenti sbloccabili tramite il pagamento di una certa quota di denaro (mensile, annuale o unica). In genere circa il 10% degli utenti è disposto a pagare per avere le funzionalità aggiuntive.

Vediamo degli esempi:

- **Flickr** è una piattaforma che permette di caricare e condividere immagini. Gli utenti free hanno uno spazio di memorizzazione e funzioni limitate. Gli utenti premium, pagando una quota mensilmente, hanno tutte le funzionalità del servizio. Uno degli obiettivi delle startup non è di diventare una multinazionale, ma di riuscire a farsi acquisire da qualche azienda più grande. Infatti, Flickr nato nel 2002, viene acquistato nel 2005 da Yahoo! per 25 milioni di dollari che a sua volta è stato acquistato da Verizon per 4 miliardi nel 2017. Yahoo! nel 2009 aveva rifiutato 44 miliardi da parte di Microsoft. Nel 2018 Flickr è diventato parte di SmugMug.
- **RedHat** le aziende che volevano usare un software open source, avevano il timore che esso poteva fallire o che nessuno potesse offrire una assistenza in caso di problemi. Quindi RedHat metteva a disposizioni per gli utenti free, software gratuiti open source. Per gli utenti premium, RedHat offriva aggiornamento dei software, sicurezza, manutenzione, eccetera. RedHat è stata acquisita per 34 miliardi di dollari da IBM nel 2019.
- **Skype** consente agli utenti web di fare chiamate e videochiamate gratuite. Pagando, al consumo o con un abbonamento, 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.
- **Netflix** usa attualmente il 15% del traffico di internet in download nel mondo. Paga i diritti dei film e guadagna per il ”noleggio” verso gli utenti. A differenza di Dropbox, Netflix ha adottato una strategia completamente diversa: nonostante avesse già costruito i data center, ha chiuso la propria rete, affidandosi completamente a Amazon AWS.

Generazione di un modello di business

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 è interessato e per cosa è disposto a pagare.

- Mettersi dal punto di vista del cliente e non dal nostro che offriamo il servizio. Cercare di capire di che cosa ha bisogno o cosa vuole il cliente
- Non concentrarsi sul passato o sulla concorrenza
- Testare vari scenari (... what if?)
- Individuare l’epicentro del nostro business: (velocità del servizio, qualità del servizio, personalizzazione...)
- Studio di alcuni modelli di business simili al nostro

Vediamo gli epicentri per l’innovazione di un business:

- **Resource Driven**, partire dalle risorse che ha già l’azienda. Come, ad esempio, Amazon Fulfillment che consiste nell’offrire a terze parti la possibilità di vendere e far arrivare i propri beni ai clienti.
- **Offer Driven**, basarsi sull’offerta. Come, ad esempio, Cemex che in un momento in cui il cemento era garantito in 48 ore è riuscita a farlo in 4 ore.
- **Customer Driven**, basarsi sui clienti. Come, ad esempio, 23andMe offriva test per DNA ai singoli clienti privati.

- **Finance Driven**, basarsi sull'aspetto finanziario. Come, ad esempio, Xerox che introdusse un modello in cui il costo delle fotocopie veniva offerto su base mensile e si garantivano 2000 copie gratuite.

Casi di studio:

- **Amazon:** Venne fondata da Jeff Bezos con il nome di *Cadabra* nel 1994, era una biblioteca online. Il business plan era di non aspettarsi profitto per i seguenti 4-5 anni, il primo profitto lo hanno avuto nel 2001. Amazon cerca continuamente di essere innovativa facendo partire dei progetti come:
    - Amazon Go, ovvero dei negozi fisici senza casse
    - Amazon prime air, dove una volta ordinato un prodotto, ti viene ricapitato entro 30 minuti da un drone. Per quest'ultimo progetto Amazon ha dei problemi soprattutto per la vendita in Europa perché le leggi impongono la "navigazione a vista", cioè in ogni momento chi pilota il drone deve poterlo vedere. Amazon da anni continua a pagare un brevetto chiamato amazon airship patent che prevede di avere un dirigibile con all'interno un certo numero di prodotti e dipendenti. Il dirigibile `e una stazione di droni, per cui può essere allocata sopra o al margine di una citta, e gli ordini che vengono effettuati dagli utenti della città possono essere serviti direttamente dal dirigibile attraverso i droni. Questo supererebbe il vincolo della navigazione a vista.

  Alcune statistiche di Amazon:

    - 300 milioni di utenti (Dicembre 2017)
    - 2 milioni di venditori di Amazon (Gennaio 2018)
    - 560 000 dipendenti (Gennaio 2018)
    - 3 724 miliardi di dollari di entrate all'anno
    - Detiene il 47,8% del mercato Cloud

- **Google:** Inizia la sua storia di successo con il famoso motore di ricerca gratuito. Le aziende pagano Google per mostrare la propria pubblicità attraverso il motore di ricerca. Google è riuscito a dominare il mercato della pubblicità grazie al *customized advertising* dove Google riesce a garantire di mostrare la pubblicità ai soli clienti potenzialmente interessati ad un certo prodotto. In ogni caso Google offre una moltitudine di servizi oltre al motore di ricerca.

FaaS

Per FaaS (Function as a Service) si intende un modello di cloud computing in cui l'elemento distintivo è l'evento. Eventi, gestiti da applicazioni, che vengono eseguiti in container "stateless" ovvero in cui qualsiasi dato applicativo è archiviato all'esterno del container stesso.

I container applicativi sono, in un certo senso, l'evoluzione delle Virtual Machine (VM). Si tratta di ambienti di sviluppo in cui si propone una nuova architettura per l'impacchettamento e la distribuzione del codice attraverso moduli indipendenti. Alla base del paradigma DevOps per lo sviluppo delle applicazioni, i container migliorano le applicazioni in termini di modularità e scalabilità. Questo approccio comporta la velocizzazione dello sviluppo applicativo grazie alla condivisione pubblica dei container, tipicamente residenti su cloud, e quindi al loro riutilizzo. Altri vantaggi riguardano la riduzione del time-to-market, l'incremento della qualità della applicazione, quindi dei rischi di sicurezza, e ne favoriscono la scalabilità.

Un modello FaaS (Function as a Service) permette agli sviluppatori di creare, eseguire e gestire i pacchetti applicativi come se fossero funzioni, senza preoccuparsi della loro architettura. Occuparsi dello sviluppo senza pensare al resto è un vantaggio enorme. Un vantaggio che si traduce, come detto, in un time-to-market più rapido, in un'ottimizzazione dei processi, in una maggiore focalizzazione sul business. E, non ultimo, in un risparmio di costi e risorse.

Da wikipedia: Function as a service (FaaS) è una categoria di servizi di cloud computing che fornisce una piattaforma che consente ai clienti di sviluppare, eseguire e gestire le funzionalità delle applicazioni senza la complessità della creazione e della manutenzione dell'infrastruttura tipicamente associata allo sviluppo e all'avvio di un'app. La creazione di un'applicazione seguendo questo modello è un modo per ottenere un'architettura " serverless " e viene in genere utilizzata durante la creazione di applicazioni di microservizi . I casi d'uso per FaaS sono associati alla funzionalità "on-demand" che consente lo spegnimento dell'infrastruttura di supporto e il mancato addebito quando non è in uso. Gli esempi includono l'elaborazione dei dati (ad es. elaborazione batch , elaborazione flusso , estrazione-trasformazione-carico (ETL)), servizi Internet delle cose (IoT) per dispositivi connessi a Internet, applicazioni mobili e applicazioni Web .

Alcuni esempi di FaaS molto diffusi sono:

- IBM Cloud Functions
- Amazon AWS Lambda
- Google Cloud Functions
- Microsoft Azure Functions (open source)
- OpenFaaS (open source).

Confronto con i servizi di hosting di applicazioni PaaS

I servizi di hosting di applicazioni Platform as a Service (PaaS) sono simili a FaaS in quanto nascondono anche i "server" agli sviluppatori. Tuttavia, tali servizi di hosting in genere hanno sempre almeno un processo server in esecuzione che riceve richieste esterne. Il ridimensionamento si ottiene avviando più processi server, per i quali lo sviluppatore viene in genere addebitato direttamente. Di conseguenza, la scalabilità rimane visibile allo sviluppatore.

Al contrario, FaaS non richiede che alcun processo server sia costantemente eseguito. Sebbene una richiesta iniziale possa richiedere più tempo per essere gestita rispetto a una piattaforma di hosting di applicazioni (fino a diversi secondi), la memorizzazione nella cache può consentire la gestione delle richieste successive in pochi millisecondi. Poiché gli sviluppatori pagano solo per il tempo di esecuzione delle funzioni (e nessun tempo di inattività del processo), è possibile ottenere costi inferiori con una maggiore scalabilità (a scapito della latenza).

Amazon AWS Lambda

FaaS è nato con un servizio offerto da AWS che si chiama **Lambda**. Permette agli utenti di mandare in esecuzione del codice (funzione in Lambda) senza gestire in alcun modo i server. L'utente semplicemente carica il codice sulla piattaforma e a questo punto Lambda si prende l'incarico di mandare in esecuzione il codice e di scalarlo opportunamente. Quindi l'utente non si preoccupa nemmeno dell'esistenza delle macchine virtuali o meccanismi di virtualizzazione. Un'altra

funzionalità è quella di associare automaticamente al codice dei trigger, ovvero, degli eventi che scatenano l'esecuzione della funzione. In AWS Lambda paghi soltanto il tempo di calcolo necessario per eseguire la funzione caricata dall'utente. Vi sono tre modi per mandare in esecuzione la funzione:

- caricare una funzione che l'utente ha già definito
- usare l'IDE di AWS per scrivere la funzione
- Lambda fornisce un insieme di template già definiti

Una volta passata la funzione, Lambda la manda in esecuzione automaticamente ogni volta che avviene il trigger specificato e si preoccupa di tutta la gestione dell'infrastruttura: scalabilità, amministrazione, fornisce metriche e logs per monitorare le prestazioni. Tutto questo ad un costo di servizio calcolato in base alla durata dell'esecuzione della funzione e alla memoria utilizzata. Il nome deriva dal lambda calcolo.

I prezzi sono vantaggiosi: Amazon fa pagare $0.20 ogni milione di richieste che riceve la funzione. Il prezzo è una combinazione sia del numero di richieste che della durata dell'esecuzione della funzione, in particolare, costa $0.0000166667 per ogni GB/second.

Esempio: Si assume che un utente allochi 512MB di memoria per la funzione che viene usata 3 milioni di volte in un mese con una durata di 1 secondo ogni volta:

per richieste 3,000,000/1,000,000 × $0.20 = $0.60 +

per durata (3,000,000 × 1) × 512/1024 × $0.0000166667 = $25 =

totale $25,60.

C'è anche un modello Freemium, in cui si ha usage tier con 1 milione di richieste gratis e 400,000GB/s di compute time al mese, quindi nell'esempio di prima:

per richieste (3,000,000 − 1,000,000)/1,000,000 × $0.20 = $0.40 +

per durata (3,000,000 × (512/1024) - 400,000) × $0.0000166667 = $18.33 =

totale $18,73

Scelta del FaaS più adeguato

Le piattaforme FaaS si distinguono in due tipologie: Commerciali e Open Source. Attraverso uno studio, si sono catalogate le 10 più importanti:

Commerciali	Open source
AWS Lambda	Apache Openwhisk
Google Cloud Functions	Fission
MS Azure Functions	Fn
	Knative
	Kubeless
	Nuclio
	OpenFaaS

Per scegliere la più adatta, si deve avere una visione specifica, divisibile in due punti di vista:

- ***Business view:***
    - **Licenza:** le piattaforme open source usano una licenza permissiva (generalmente Apache 2.0) mentre le commerciali usano licenze di tipo proprietario.
    - **Installazione:** tra le piattaforme commerciali solo Azure Functions ha alcune parti installabili.
    - **Source Code:** tutte le piattaforme open source sono hostate su GitHub e molte sono già implementate per partire, delle commerciali solo Azure Functions è parzialmente open source.
    - **Interfaccia:** le piattaforme open source possono variare in modo considerevole l'una dall'altra nei modi in cui sono amministrate. Tutte dispongono di CLI, ma non tutte hanno anche API e GUI.
    - **Community:** le piattaforme open source hanno una community più disponibile e larga, considerando la documentazione su GitHub e su forum simili a StackOverflow, dove molte persone discutono di problemi rilevati e propongono soluzioni.
    - **Documentazione**: le piattaforme commerciali hanno garanzia di avere una documentazione sviluppata e dettagliata sull'utilizzo della piattaforma. Nell'open source può non essere sempre così.

- ***Technical view:***
    - **Development:** Java, Node.Js, Python sono I più comuni supportati. Docker è sempre più usato per il runtime, mentre le piattaforme commerciali offrono spesso dei plugin di IDE e text editors.
    - **Versioning:** la maggior parte delle piattaforme open source implementano il cosiddetto *Implicit Versioning*, al contrario delle piattaforme commerciali che spesso hanno un meccanismo di *Dedicate Versioning*.
    - **Event sources:** tutte le piattaforme supportano sincronismo, invocazione di funzioni basato su HTTP, mentre l'invocazione asincrona degli eventi è supportata da poche. Scheduler e messaggi sono anche spesso supportati. Più della metà delle piattaforme open source non supporta eventi basati sul data storing, ma in generale molte piattaforme supportano l'integrazione di documenti per eventi personalizzati.
    - **Orchestrazione delle funzioni:** nel codice ci possono essere più funzioni combinate tra loro. La maggioranza delle piattaforme offre un "orchestratore" di funzioni dedicato, la meta delle piattaforme usano dei DSL specifici per i workflow, diagrammi di flusso, ...
    - **Testing e debugging:** la maggior parte supporta testing e debugging delle funzioni, ma le piattaforme commerciali offrono spesso più funzionalità sofisticate, mentre le open source si limitano a chiamate di test e debugging log-based.
    - **Osservabilità:** le commerciali offrono strumenti dedicati per il monitoraggio e il logging delle funzionalità, mentre spesso le open source utilizzano software di terze parti.
    - **Distribuzione delle applicazioni:** la maggior parte delle piattaforme utilizza un approccio dichiarativo per automatizzare la distribuzione delle app. Le commerciali

hanno un supporto nativo CI/CD attraverso tool dedicati, mentre le open source non documentano l'integrazione con CI/CD pipelines.

- **Riutilizzo del codice:** soltanto AWS Lambda e MS Azure Functions sono dotate di un *Function Marketplace.*
- **Gestione degli accessi:** le commerciali hanno in maniera nativa processi di autenticazione e controllo degli accessi alle risorse, le open source tipicamente rimandano queste funzionalità all'ambiente di hosting.

Non c'`e una piattaforma migliore, dipende da quali sono i bisogni dell'azienda o di chi sviluppa l'app. *FaaStener Prototype* permette di scegliere il FaaS più adatto. Il fatto di poter scrivere così facilmente le funzioni e renderle collegabili è un grosso vantaggio in termini di velocita di sviluppo e riduce la quantità di testing. Di contro una volta sviluppata tutta l'applicazione, se si vuole cambiare provider, si deve aprire il codice sorgente e modificarlo. Ci sono comunque delle strategie per evitare o quantomeno ridurre il lock-in. Se non si vuole rischiare si può sempre usare l'open source.

GREEN COMPUTING

Informatica Verde (dall'inglese *green computing* o *green IT*), si riferisce ad un'informatica ecologicamente sostenibile. Si occupa dello studio e della messa in pratica di tecniche di progettazione e realizzazione di computer, server, e sistemi connessi come ad esempio monitor, stampanti, dispositivi di archiviazione, reti e sistemi di comunicazione efficienti con impatti ambientali più o meno limitati (ma non completamente "nulli"). La green IT si pone un duplice obiettivo: il raggiungimento di un tornaconto economico e di buone prestazioni tecnologiche, rispettando le responsabilità sociali, ambientali ed etiche. Quindi comprende la sostenibilità ambientale, l'efficienza energetica, il costo totale di proprietà, che comprende il costo di smaltimento e riciclaggio. La green IT è lo studio e l'utilizzo di tecnologie informatiche in modo efficiente.

Per affrontare in modo efficace e completo l'impatto dei computer sull'ambiente, dobbiamo adottare un approccio olistico e rendere l'intero ciclo di vita più ecologico affrontando il problema della sostenibilità ambientale, sulla base dei seguenti quattro percorsi complementari:

- Utilizzo Verde — ridurre il consumo di energia da parte dei computer e degli altri sistemi informatici e utilizzarli in modo ecologicamente corretto.
- Smaltimento Verde — revisionare e riutilizzare i vecchi computer, riciclare tutti i dispositivi elettronici non reimpiegabili.
- Progettazione Verde — la progettazione a basso consumo energetico e componenti dell'ambiente, computer, server, apparati per il raffreddamento, e data center.
- Fabbricazione Verde — realizzare componenti elettronici, computer e altri sottosistemi con un minimo impatto ambientale.

In particolare, questa branca dell'informatica nasce dalla sempre crescente delineazione energivora che sta assumendo l'informatica e in generale l'utilizzo delle ICT *(Information and Communication Technologies*): basti pensare che per la fabbricazione di un singolo PC sono necessari 1800 Kg di materia prima, l'utilizzo delle ICT contribuisce per il 4% all'emissione totale di CO_2 e per il 9% al consumo totale di energia in Europa. E le proiezioni indicano che nel 2030 si arriverà addirittura al 20%.

Impatto dei datacenter

PUE è l'indice che cerca di fare un rapporto tra l'energia necessaria a far funzionare un datacenter, diviso, l'energia necessaria per la parte di IT. Il PUE misura l'efficienza energetica ma non misura il grado di utilizzo delle energie rinnovabili. Con energia rinnovabile si intendono le fonti di energia che si rinnovano in maniera naturale nel corso della vita di un essere umano (onde, vento, sole, pioggia, ecc...).

Obsolescenza

- **Programmata:** si ha quando un prodotto è deliberatamente concepito per avere una durata prestabilita del ciclo di vita. Esistono autorità che controllano abusi su questo tipo di obsolescenza.
- **Percepita:** situazione per la quale l'utente si convince che il suo prodotto ha bisogno di un update, anche se il suo ciclo di vita non è terminato.

L'obsolescenza contribuisce all'aumento dell'impatto ambientale in quanto tende a diminuire sempre di più lo span di durata di vita di un prodotto.

In questa ottica si introduce il termine *E-waste* che sta ad indicare tutta la spazzatura che deriva da prodotti elettronici/tecnologici e che spesso si presta ad un commercio illegale per il recupero dei materiali preziosi che sono contenuti nei vari dispositivi elettronici, con un conseguente smaltimento non adeguato alla restante parte del prodotto.

Energia rinnovabile e clicking clean

Il progetto di alcune associazioni ambientaliste come Greenpeace è quello di arrivare ad ottenere una struttura informatica capace di attingere per il 100% ad energia derivante da fonti rinnovabili. L'idea è stata anche sposata da alcuni colossi dell'informatica come Google, Apple ed altri, che stanno cercando di portare al 100% l'utilizzo delle energie rinnovabili, per ridurre i costi dell'energia utilizzata e per andare incontro ai sentimenti ambientalisti che crescono nei propri dipendenti.

Esistono tuttavia dei limiti al raggiungimento del 100% di energie rinnovabili, tra i quali si ha **una bassa trasparenza da parte delle aziende** nel fornire i dati sull'energia che consumano e soprattutto, dall'**impossibilità per alcune aziende di avere accesso a energie rinnovabili**.

Se da una parte i grandi colossi fanno a gara per dimostrare il loro utilizzo di energia pulita, dando l'esempio, altrettanti grandi colosso si rifiutano di fornire i dati necessari per comprendere se siano verso una strada giusta per completare la transizione verso l'utilizzo di energie rinnovabili.

Green directions

L'obiettivo è di spostarsi in tre direzioni per ridurre l'impatto ambientale, da sviluppare parallelamente:

- **Minimizzare il consumo di energia:** obiettivo raggiungibile tramite
  - Responsabilizzazione del consumatore
  - Riduzione impatto dei Data Center tramite scalabilità dei server e miglioramento delle tecniche di raffreddamento
  - Iniziative per regolamentazione e sensibilizzazione sul risparmio energetico
- **Progettazione di dispositivi efficienti energeticamente:**
  - Implementazione delle nanotecnologie
  - Sviluppo di software più efficienti

    - Design di prodotti che tenda a minimizzare l'energia consumata (opzioni di stand-by, meno led di segnalazione...)
- **Riduzione di e-waste:**
    - Responsabilizzazione dell'utente finale
    - Iniziative rivolte al riciclaggio
    - Regolamentazioni degli smaltimenti

In conclusione: Il green computing è un po' in contrasto con gli obiettivi dell'IC: 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 pi`u energia. Inoltre, applicare tecniche per il risparmio energico non conviene a livello economico. Nel 2016 a Parigi ci fu un trattato per ridurre il riscaldamento globale. Nel 2017 Trump ha deciso di far uscire gli USA dal trattato, ma nel 2021 Biden ha firmato il reingresso nel trattato.

Global climate predictions: nel 2050 ci saranno 200 milioni di migranti per problemi ambientali.

Calcolo dell'impatto ambientale dell'IoT (Internet of Things)

- Sviluppo di un modello analitico per l'analisi e la stima dell'impatto ambientale per impianti di IoT alimentati tramite installazioni di impianti ad energia solare.
- Stima dell'impatto sull'energia e sui rifiuti generati durante e dopo il ciclo di vita del prodotto (produzione, uso, manutenzione, smaltimento).
- Confronto tra la normale implementazione e una realizzazione del sistema IoT completamente alimentata da energia rinnovabile con un minimo impianto di raccolta di energia e un sottosistema di batterie per lo stoccaggio.

Nozioni di base sull'ingegneria del software sostenibile

1. **CO_2:** è il principale gas serra presente nell'atmosfera terrestre; pertanto, è importante averne una produzione controllata; è inoltre molto importante per la fotosintesi clorofilliana.
2. **CO_2-eq:** Esprime l'impatto sul riscaldamento globale di una certa quantità di gas serra rispetto alla stessa quantità di anidride carbonica.
3. **Joule [J]:** unità di misura dell'energia.
4. **Watt [W]:** unità di misura della potenza, corrisponde a J/s.
5. **Watt-ora:** unità di misura per energia (1 Wh=3600 J).
6. **Efficienza ε:** $\#\ di\ calcoli/elettricità$. (non corrisponde alla correttezza).

Definizione di *Sostenibilità*

«Lo sviluppo sostenibile è lo sviluppo in grado di soddisfare i bisogni del presente senza compromettere la possibilità delle future generazioni di soddisfare i propri».

Digitale e impronta ecologica

Il digitale ha una propria impronta digitale importante e non sottovalutabile; tuttavia, può venirci incontro nella gestione delle altre risorse, cosa che consentirebbe di gestirle in maniera più efficiente. Per questo è importante sviluppare l'ingegneria del software sostenibile e l'idea che fa nascere questa scienza.

Ingegneria del software

Un ingegnere del software è una persona che applica i principi dell'ingegneria del software per progettare, sviluppare, mantenere, testare e valutare il software informatico. Gli Ingegneri del software devono perseguire obiettivi contrastanti continuamente: tipicamente costo, tempo e qualità. Devono inoltre soddisfare requisiti funzionali e non-funzionali del loro prodotto.

Modello GREENSOFT

Un modello olistico per lo sviluppo, utilizzo e dismissione di software sostenibile.

«Green and Sustainable Software Engineering […] is the art of defining and developing software products in a way, so that the negative and positive impacts on sustainable development that result and/or are expected to result from the software product over its whole life cycle are continuously assessed, documented, and used for a further optimization of the software product»

Caso di studio: aggiornamento di un SO

In generale, in questa situazione si ha da un lato l'utente finale che beneficia e che richiede che il suo SO venga aggiornato, dall'altra il team di persone che contribuisce alla costruzione dell'aggiornamento, tra cui esistono chiaramente diversi ruoli. Alcuni di questi ruoli possono anche essere ricoperti da una stessa persona. Tra i principali si evidenziano: Sviluppatore, Operatore e Architetto.

Prospettiva Utente

- Costo elettrico dell'aggiornamento trascurabile (non saranno interessati a ridurlo).
- Costo in termini di tempo non trascurabile (saranno interessati a ridurlo).

In generale l'utente, dal punto di vista dell'impatto ambientale può:

- Essere consapevole della propria impronta ambientale.
- Richiedere software di qualità maggiore per evitare aggiornamenti frequenti.
- Richiedere che la sostenibilità sfrutti i requisiti esistenti.

Prospettiva architetto

L'architetto è colui che si occupa di determinare i requisiti funzionali e no funzionali del sistema e che progetta le interazioni tra i componenti. I requisiti funzionali parlano di un particolare risultato del sistema quando un'attività viene eseguita su di essi dall'utente. D'altra parte, il requisito non

funzionale fornisce il comportamento complessivo del sistema o dei suoi componenti e non sulla funzione.

Gli architetti possono includere la sostenibilità nei requisiti non funzionali ad esempio migliorando:

- Scalabilità - aumentando quindi la possibilità di adattare il sistema alla domanda, ad esempio scomponendo i servizi in unità scalabili (calcolo, rete, storage) attraverso CDN o dividendo in microservizi.
- Performance – aumentando il tempo di risposta nel Service Level Agreement.

CDN (*Content Distribution Network*)

È un termine coniato sul finire degli anni '90 per descrivere un sistema di computer collegati in rete attraverso Internet, che collaborano in maniera trasparente, sotto forma di sistema distribuito, per ripartire contenuti (specialmente contenuti multimediali di grandi dimensioni in termini di banda, come l'IPTV) agli utenti finali ed erogare servizi di streaming audio e video. Grazie alla CDN si riducono notevolmente i tempi di caricamento di una pagina perché quando un contenuto viene richiesto a rispondere è il server più vicino geograficamente, il che si ripercuote positivamente sulle prestazioni del sito.

Utilizzando questo sistema anche per mantenere i dati degli aggiornamenti, i vantaggi sono quindi notevoli:

- Minore distanza per raggiungere i dispositivi dell'utente finale (più veloce, consumo ridotto)
- Minore utilizzo di banda (meno nodi infrastrutturali)
- Migliore utilizzo della rete (miglioramento prestazioni)

Il problema dell'utilizzo di questa soluzione, si presenta sotto la veste del cosiddetto problema del *Bin Packing*, ovvero il problema dello zaino, che consiste nell'inserire N oggetti di taglie diverse in K contenitori di capacità fissata C, minimizzando il numero di contenitori utilizzati.

- È un problema NP-HARD.
- Corrisponde a minimizzare il numero di server per far girare i servizi che compongono il sistema.
- L'architetto può contribuire progettando componenti più piccole, risolvendo il problema del bin packing con meno frammentazione possibile.
- La soluzione del problema verrà poi delegata a livelli più bassi o ad altri ruoli (sviluppatori, operatori, gestori dell'infrastruttura...).

Ottimizzazioni consentite all'architetto

Per ridurre l'impronta ambientale, l'architetto può quindi adottare tre criteri principali:

- Definire requisiti che aumentino la sostenibilità (ad esempio ridurre i tempi di scaricamento...).

- Favorire la scalabilità e scomponendo il servizio in microservizi
- Scegliere una scomposizione fine per facilitare la risoluzione del bin-packing

Prospettiva sviluppatore

Gli sviluppatori sono coloro che sviluppano il sistema software seguendo i requisiti funzionali e non funzionali che gli sono dettati dall'architetto.

Devono compiere diversi tipi di scelte:

- Tecniche:
  - Linguaggio di programmazione
  - Scelta delle librerie con algoritmi più efficienti
  - Scelta di pattern efficienti, a eventi vs polling
- Implementative:
  - implementare servizi che richiedano un quantitativo di risorse ragionevole, valutando ad esempio
    - Java (compilazione al runtime, garbage collector efficiente, multi-thread)
    - Python (interpretato, reference counting, limiti di concorrenza)
  - Disaccoppiare implementazione da infrastruttura in modo da facilitare la migrazione a Cloud Provider differenti

La scelta di un linguaggio di programmazione, ad esempio, non è banale e richiede di valutare differenti aspetti, tra i quali energia consumata, tempo di esecuzione, memoria utilizzata, senza dimenticare che non tutti i team di sviluppo possono padroneggiare tutti i linguaggi di programmazione.

In questa ottica è fondamentale per lo sviluppatore riuscire a misurare la sostenibilità e il soddisfacimento di alcuni requisiti. Per questo scopo vengono spesso definiti degli indicatori di performance, per valutare vari aspetti del software (ad esempio rapporto tra tempo di uso della CPU e numero di aggiornamenti consegnati).

Inoltre, il processo stesso dello sviluppo di un software ha un proprio impatto ambientale, in quanto si consumano delle risorse. Occorre quindi:

- Sprecare meno energie possibili
- Configurare pipeline che facilitino il supporto del software
- Testare in modo automatico (CI/CD) per evitare difetti del software

Cosa può fare dunque lo sviluppatore dalla sua posizione per ridurre lo spreco di energie?

- Ottimizzazione dell'efficienza del software

- Disaccoppiare applicazione dai dettagli sulla infrastruttura
- Misurare e testare per migliorare il software
- Ridurre lo spreco di risorse durante lo sviluppo

Prospettiva operatore

Gli operatori sono coloro che gestiscono l'infrastruttura e garantiscono la disponibilità dell'applicazione o in generale del servizio sviluppato. Devono quindi:

- Offrire risorse infrastrutturali adeguate a garantire le performance desiderate
- Utilizzare al meglio le risorse per favorire la sostenibilità e soprattutto contenere i costi
- Garantire possibilità di backup e ripristino
- Implementare log e monitoraggio del sistema

In questa ottica, si può quindi:

- Scegliere diverse piattaforme per implementare il servizio, consentendo una risoluzione del bin packing vicino a quella ottima (ad esempio adottando piattaforme di virtualizzazione, container, architetture serverless).
- Automatizzare il processo di scalabilità del servizio (auto-scaling)
- Sfruttare il principio di località dei dati per minimizzare trasmissione di rete e velocizzare i tempi di risposta
- Usare istanze spot per svolgere task non prioritari (ottimizzazione delle risorse)

Prospettiva della legge: il quadro normativo

Le leggi e i regolamenti possono favorire una transizione verso sistemi ad impatto energetico più basso. Le normative possono quindi tassare gli effetti negativi sull'impatto energetico generati da un prodotto (dallo sviluppo alla dismissione) e soprattutto possono obbligare le aziende a adottare modelli di trasparenza nei confronti dei consumatori, che possono così essere consapevoli di ciò che accade. Sfortunatamente, a differenza di alcune istituzioni come l'Unione Europea che stanno sviluppando normative per la transizione ecologica, non esistono trattati né normative al livello globale.

Service-oriented Architecture

Microservices

Un'architettura di microservizi , una variante dello stile strutturale SOA, è un modello architettonico che organizza un'applicazione come una raccolta di servizi a grana fine e ad accoppiamento lasco, che comunicano tramite protocolli leggeri. Uno dei suoi obiettivi è che i team possano sviluppare e distribuire i propri servizi indipendentemente dagli altri. Ciò si ottiene riducendo diverse dipendenze nella base di codice, consentendo agli sviluppatori di evolvere i propri servizi con restrizioni limitate dagli utenti e nascondendo gli utenti da ulteriore complessità. Di conseguenza, le organizzazioni sono in grado di sviluppare software con una crescita e una dimensione rapide, oltre a utilizzare più facilmente i servizi standard. Una architettura di questo tipo è quindi in grado di:

- Ridurre tempi di sviluppo nonché di rebuild o di aggiornamento
- Aumentare la scalabilità di un servizio

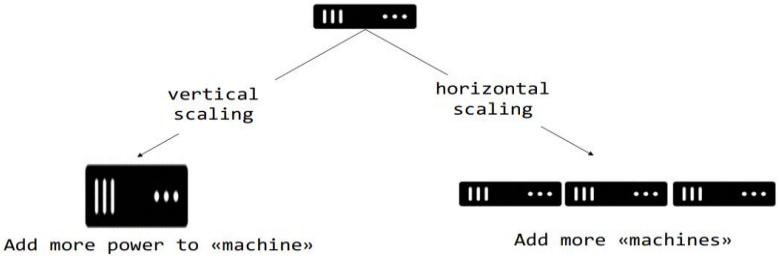
A differenza di applicazioni monolitiche, molto sviluppate in passato (composte di 3 livelli: lato client, lato server, database), per le quali una minima modifica causa lunghi tempi di sviluppo e di revisione della totalità del codice, un'applicazione basata sullo sviluppo di microservizi "a compartimenti stagni", consente di ridurre i tempi e i costi. Vediamo quindi le principali caratteristiche dei microservizi:

- Ogni microservizio è in grado di girare nel suo proprio container.
- I servizi comunicano tra di loro attraverso protocolli leggeri ed efficienti.
- I servizi possono essere implementati ognuno in un diverso linguaggio di programmazione, con diverse tecnologie di stoccaggio, a seconda delle necessità di ognuno.

Proprietà essenziali dei microservizi

1. **Service-orientation:** le applicazioni sono sviluppate come insiemi di servizi (architettura modulare) ognuno su un Container diverso, che comunicano fra loro con meccanismi semplici (lightweight), sostanzialmente protocolli rest, quindi richieste-risposta HTTP. Oltre alle chiamate sincrone ci sono dei *dumb message* bus come per esempio delle code asincrone. I servizi sono poliglotti, ovvero, servizi diversi possono essere scritti in linguaggi diversi. Nella prima versione dei microservizi venivano usati degli ESBs cioè dei connettori ricchi di funzionalità che permettevano ai servizi di collaborare tra loro, non erano code asincrone ma avevano molte funzionalità.
2. **Organizzare i servizi intorno a delle business capabilities:** la legge di Conway 1968 dice che *le organizzazioni che progettano dei sistemi finiscono con l'essere vincolati nel produrre delle progettazioni che riflettono la struttura delle proprie organizzazioni*. Per esempio, se nell'azienda vi sono tre divisioni: uno che si occupa di client, uno di server e uno di database, molto probabilmente il sistema informatico finirà per avere tre pezzi. I microservizi invece dicono di passare a un modello totalmente diverso, ovvero, avere dei *cross-functional teams* dove in ogni gruppo c'è una persona esperta di cose diverse.

3. **Decentralizzazione della gestione dei dati:** permettere ad ogni servizio di avere e gestire il proprio database. Questo comporta avere dei problemi di consistenza e integrità. I microservizi utilizzano *l'eventual consistency*, ovvero, invece di mantenere i database sempre consistenti, accettano che per un periodo non lo siano sapendo che lo saranno prima o poi. Se si ha una operazione su due database, se uno dei due non si può aggiornare, non importa, lo si farà dopo. Ci sarà un incaricato che cercherà di portare a termine l'aggiornamento. Si sacrifica 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 rebuild solo di una parte non di tutta l'applicazione.
5. **Scalabilità orizzontale:** si vuole scalare non l'intera applicazione, ma solamente quel servizio che ha problemi di performance.



6. **Fault tolerant:** il punto più a sinistra è l'API Gateway, se fallisse potrebbe causare un fallimento a cascata che può 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. Le chiamate sincrone tra servizi, che sono utili per rendere l'architettura più efficiente, inducono un effetto moltiplicativo del downtime. Per esempio, se un servizio dipende da altri 30 servizi, e questi ultimi servizi hanno un uptime del 99.99% allora 99.9930 (30 servizi) × 24 (ore giorno) × 30 (giorni mese) produce una probabilità di due ore di downtime al mese. Quindi si arriva alla conclusione che si deve progettare tenendo conto dei fallimenti, ovvero, la *fault tollerance deve essere un requisito*. Una soluzione è inserire un *circuit breaker* per evitare il fallimento a cascata: viene messo un proxy che spezza il circuito, se prima C chiamava S, ora C chiama il proxy che manda la richiesta a S. Quindi fa da intermediario. Se dovessero aumentare i tempi di risposta o non riuscisse ad ottenerla, il circuit breaker fornisce comunque una risposta anche se S non gliel'ha data. Per esempio, la barra di ricerca di Spotify: se un utente cerca qualcosa, nel caso di fallimento semplicemente non riceve niente, cioè riceve una cella bianca. Poi l'utente rimette la query e ha successo. Non è un fallimento se l'utente non se ne accorge.

Test bravely

Introdotto da Netflix con uno strumento chiamato **chaos monkey**, che permette di testare in modo coraggioso il funzionamento di una applicazione. Lo si può configurare in modo tale da "uccidere" istanze di macchine virtuali o Container nell'applicazione in fase di produzione per vedere se il sistema riesce a riconfigurarsi e rendere tutto ciò trasparente agli occhi dell'utente.

DevOps

Con DevOps si intende una serie di strumenti e tecniche che si utilizzano per lo sviluppo di applicazioni, servizi e prodotti informatici, intenti a migliorare e facilitare le tecniche di sviluppo di

tali prodotti. Per decenni lo sviluppo del software veniva fatto da un gruppo di development, e una volta ottenuto il prodotto finito, lo passavano agli operator che gestivano il rapporto con i clienti. Questa differenza tra i due gruppi, ambiente di produzione e ambiente di sviluppo, si percepiva anche nell'applicazione stessa. Adesso con DevOps non si fanno progetti ma prodotti: tu lo hai costruito e tu lo mandi in esecuzione. Quindi l'idea non è separare, ma il team full stack che si occupa di un servizio, lo progetta, lo crea e se lo gestisce.

Conclusione

I microservizi non vanno presi in considerazione se l'applicazione funziona in modo semplice e non è troppo complessa come monolite. Se la complessità di base non è elevata, i microservizi appesantiscono soltanto lo sviluppo e perdono le loro potenzialità.

- **Pro**
  - minore lead time
  - scalabilità
- **Contro**
  - eccessiva comunicazione tra servizi
  - complessità
  - facilita nello sbagliarsi
  - non è semplice decentralizzare i dati
  - se un team è poco preparato, avrà grossi problemi ad utilizzare i microservizi

Netflix: 190 countries, 10s of languages, 1000s of device types.

Spotify: 810 active services, 90+ squads, 600+ developers.

Kubernetes

Nel 2014, dopo il successo di Docker, è stato lanciato Kubernetes che è un sistema per "orchestrare" Container. Possiede molte funzionalità simili a Docker swarm ma è più efficiente. Docker e Kubernetes non sono in contrapposizione tra di loro, ma si integrano a vicenda.

Kubernetes è una piattaforma per l'orchestrazione dei containers. K8s gestiscono l'intero ciclo vitale dei singoli containers, che vengono creati e distrutti come risorse a seconda della necessità.

di liberare CPU e memoria quando i task sono terminati e, in generale, di avere una schedulazione efficiente, cosa che Docker non ottiene.

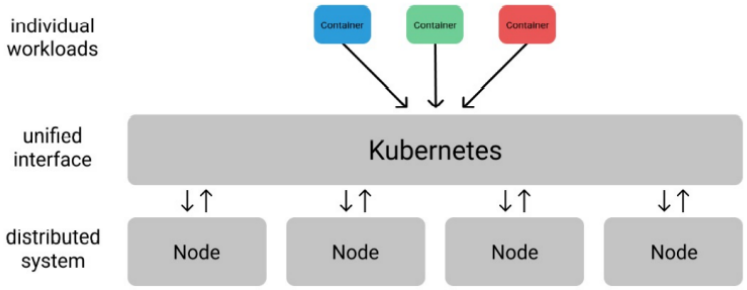
Principali caratteristiche di Kubernetes

- **Definizione semplice di *desire state*:** K8s tiene traccia dello stato attuale in ogni momento e se non coincide con quello indicato da noi interviene per apportare modifiche e far si che il nostro sistema torni da solo nello stato desiderato.
  Lo stato desiderato viene definito attraverso una collezione di oggetti, ognuno dei quali ha una specifica situazione che lo descrive quando è nello stato desiderato e

uno stato che identifica lo stato in cui si trova in un dato momento. Kubernetes testa costantemente ogni oggetto:

- se non risponde ne effettua lo shut-down e ne crea una nuova versione per rimpiazzarlo.
- se lo stato dell'oggetto si è spostato in uno che non coincide con quello desiderato, K8s eseguirà i comandi specifici per ripristinarlo nello stato desiderato.

- **Distribuzione**: K8s provvede a unificare l'interfaccia per interagire con un cluster di macchine, senza che ci si debba preoccupare di comunicare separatamente con ogni macchina.

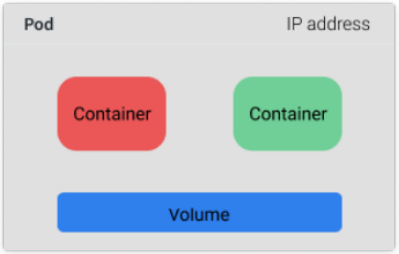


- **Disaccoppiamento:** i container dovrebbero essere sviluppati per provvedere ad eseguire una singola mansione (microservice-based architecture). K8s, infatti, supporta l'idea di servizi disaccoppiati che possono essere scalati e upgradati separatamente.
- **Infrastruttura immutabile:** per ottenere il massimo dai container e dalla loro orchestrazione, si dovrebbe progettare una infrastruttura immutabile. Durante il ciclo di vita di un progetto, dovremmo usare sempre la solita immagine del container, effettuando modifiche solo alla configurazione esterna del container. Se si devono apportare modifiche internamente, si preferisce distruggere il container e avviarlo da capo.

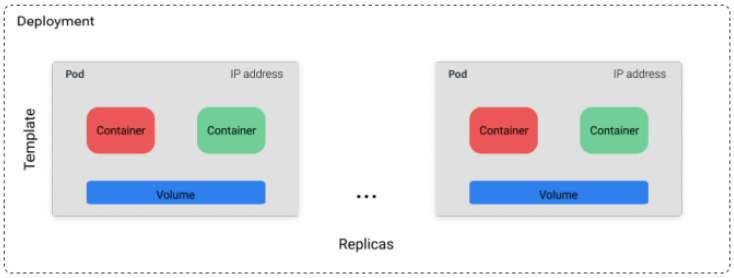
Oggetti di Kubernetes

Gli oggetti di Kubernetes possono essere definiti in manifesti.

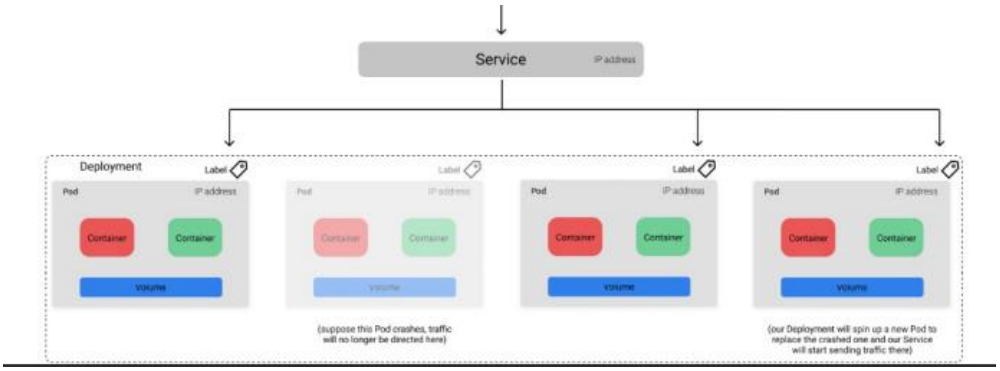
- **Pod**: Un pod è un gruppo di uno o più Containers con storage/rete condivisa, e con una specifica per come gestire i Container. I contenuti del pod vengono allocati e schedulati per essere eseguiti in un contesto condiviso.



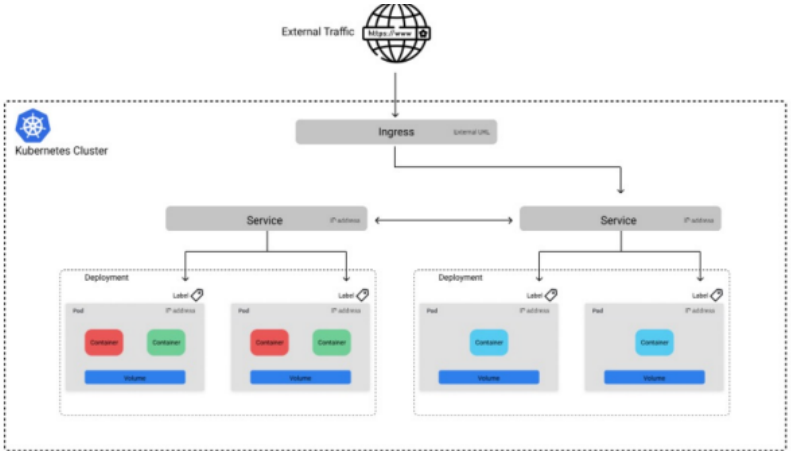
- **Deployment:** include una collezione di Pod definite come un template ed un *replica count*, ovvero il numero n di volte che si vuole runnare il template. Il cluster proverà ad avere n Pod in qualsiasi fase della sua esecuzione.



- **Service:** ad ogni Pod è assegnato un indirizzo IP attraverso il quale possiamo comunicare con la Pod stessa. Il service di K8s provvede a stabilire una connessione stabile con la Pod desiderata, anche se questa subisce modifiche come update, scaling o failures. Il service sa a quale Pod indirizzare il traffico basandosi su delle *Labels* (coppie di chiave-valore) che definiscono i metadata di ogni Pod.



- **Ingress:** Il service ci consente di esporre le applicazioni solo dietro un endpoint stabile disponibile per il traffico interno del cluster. Per esporre la nostra applicazione al traffico esterno al nostro cluster, è necessario definire un oggetto Ingress. Possiamo selezionare quali Servizi rendere disponibili al pubblico.



- Esistono anche molti altri oggetti che qui non vengono specificati... (Job, Volume, Secret, NameSpace...).

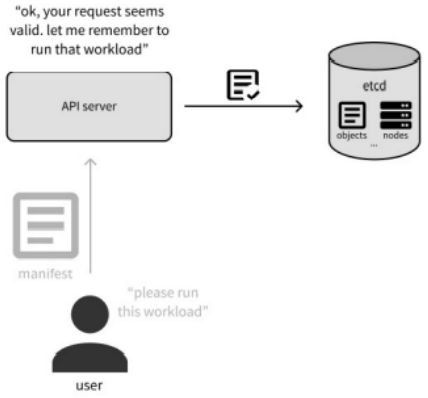
K8s control plane

Esistono due tipologie di macchine all'interno di un cluster:

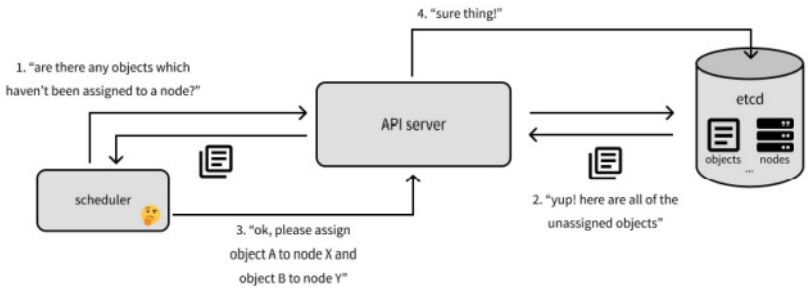
- **Master node:** (spesso è uno solo) è la macchina che contiene la maggior parte delle componenti del control plane.
- **Worker node:** macchina che fa girare i carichi di lavoro dell'applicazione.

Master Node (approfondimento)

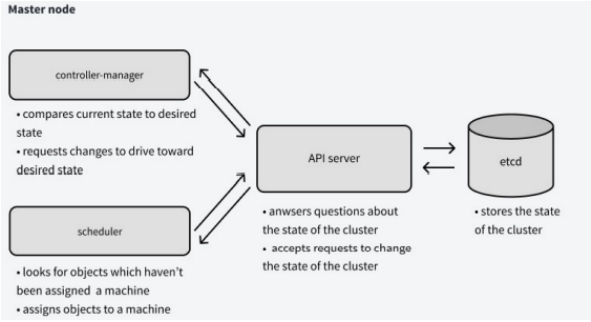
L'utente fornisce nuove specifiche sugli oggetti all'API server del master node, l'API server valida la richiesta e agisce come una interfaccia unificata per richiedere lo stato corrente del cluster. Lo stato del cluster viene stoccato in un *distribuited key-value store etcd*.



Lo scheduler, a questo punto, determina dove gli oggetti devono essere eseguiti, chiedendo all'API server quali siano gli oggetti che non sono ancora stati assegnati ad una macchina, determinando a quale macchina dovrebbero essere assegnati e rispondendo all'API server per dare conferma dell'assegnamento.

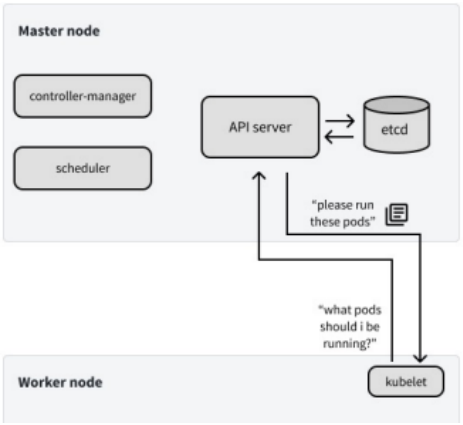


Adesso il controller-manager monitora lo stato del cluster tramite l'API server. Se lo stato attuale differisce da quello desiderato, il controller-manager applicherà i cambiamenti necessari per guidare lo stato del cluster verso quello desiderato.



Worker Node (approfondimento)

- **Kubelet:** Un kubelet è un *node-agent* che, comunicando con il server API apprende quale task è stato assegnato al nodo. È responsabile quindi dell'esecuzione del workload da parte del nodo stesso. Inoltre, quando un nuovo nodo entra a far parte del cluster, il kubelet segnala al server API la presenza del nuovo nodo, in modo che lo scheduler possa assegnargli dei task.



- **Kube-proxy**: permette ai container di comunicare con altri container residenti su nodi diversi del cluster.

Quando non usare K8s

- se il workload è eseguibile su una sola macchina
- se il carico di computazione è leggero
- se non si ha intenzione di dividere un monolite in microservizi
- se non si ha intenzione di effettuare molti cambiamenti al servizio che si distribuisce

Differenze tra Docker e Kubernetes

Docker Swarm	Kubernetes
▪ Simpler to install ▪ Softer learning curve	▪ Features auto-scaling ▪ Higher fault tolerance ▪ Huge community ▪ 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

Datacenters

Google

Energia distribuita in modo ”overhead” (dall’alto). Utilizza un sistema di raffreddamento con delle torri e dei tubi in cui scorre l’acqua. È l’azienda che tiene la temperatura pi`u alta rispetto alle altre aziende (27 °C). Server racks con architettura personalizzata. Quando cambiano i drive, dopo averli sostituiti, li distruggono. Google cerca di ridurre l’emissione di CO2.

Unipi

A Pisa ci sono 3 nodi DCI (*Data Center Interconnect*) in città con una capacità di connessione tra loro di 100Gb/s. Il datacenter più grande si trova a San Piero a Grado. La connessione è di tipo **no single point of failure**, ovvero, è presente ridondanza per ridurre al minimo i punti di fallimento. Per il traffico est-ovest, quello che si riferisce al traffico di dati all’interno della rete di datacenter, è implementata una **spine-leaf topology** che contribuisce all’aspetto di ridondanza, e quindi di affidabilità del sistema. Il traffico nord-sud è quello che fa riferimento al traffico con l’esterno. Il datacenter di San Piero usa un sistema di raffreddamento adiabatico che permette di avere un indice PUE < 1.3. Il PUE è un indice per l’efficienza energetica, sempre > 1 ma il più vicino possibile a 1, e si calcola come *total facility power/IT equipment power*. È uno degli indicatori più utilizzati per classificare quanto è green un datacenter.

Datacenter infrastructure management

Le azioni principali per gestire un datacenter sono il **planning** (meeting, organizzazione...), **l’installazione e gestione dei server** e delle connessioni, mantenere il cablaggio pulito, e infine tutti gli **aspetti riguardati la sicurezza**. Per monitorare i datacenter si ha anche una visione grafica (DCIM) per controllare *energia, sicurezza, ambiente e raffreddamento attraversi grafici e reports* con notifiche immediate in caso di errori o fallimenti. Un altro aspetto importante è quello della documentazione: se la documentazione non viene svolta in modo regolare, può portare vari problemi perché non si conoscono le informazioni né storiche né aggiornate su quale sia lo stato dell’infrastruttura.

Cosa non fare all’interno di un datacenter:

- gestione poco ordinata del cablaggio
- introdurre cibi e bevande
- trascurare la documentazione
- lasciare porte aperte tra una stanza e l’altra
- click accidentali di interruttori

Il personale di un datacenter deve essere qualificato e deve rispettare delle regole:

- rapido ed efficiente nella reazione ai problemi
- gestire il panico
- seguire la checklist

In particolare, la **checklist** è importante sia per l'operatore che per il gestore, per mantenere dei comportamenti allineati e garantire una alta qualità nel servizio.

Continuità del servizio e recovery

Per i data center è di vitale importanza mantenere il servizio stabile e funzionale e prevenire i danni dovuti a disastri ambientali o gravi incidenti. È fondamentale implementare tecniche efficienti di prevenzione e spegnimento degli incendi, prevenzioni di alluvioni e, essendo più frequente come problema, è importante prevenire cali di tensione o blackout totali attraverso installazioni di gruppi di continuità in grado di supportare il datacenter in caso di problemi alla linea elettrica, per un tempo sufficiente alla risoluzione e al riallaccio.

- Copia dei dati separate fisicamente tra loro (data center ubicati in luoghi diversi)
- Testing di piani di evacuazione e antiincendio (altrimenti è inutile anche solo progettarli)
- Formare e preparare il personale, anche personale esterno alla organizzazione ma pronto ad intervenire in caso di disastro
- Piani di recupero nel caso la prima soluzione fallisca o sia inapplicabile al problema

Hyperconvergence

- La **struttura tradizionale** di un datacenter è di avere server, networking e storage separati tra loro e poi montati assieme, questo può causare l'incompatibilità tra le componenti.
- La **struttura converged** invece fornisce subito un insieme di componenti pretestati e validati con un sistema di gestione avanzato 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 un singolo box; riduce di molto i costi hardware e aumenta la scalabilità, 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 (non tutti i DB possono essere virtualizzati) e i relativi software non sono economici.

Dal cloud all'Internet of Things

L'internet delle cose è diventato negli ultimi anni estremamente pervasivo, entrando a far parte delle nostre vite sottoforma di una grande quantità di aspetti (dispositivi wearable, smart city, sistemi di guida autonoma, altoparlanti intelligenti, smart devices, ...); si stima che, se nel 2021 esistevano circa 10 miliardi di dispositivi attivi, nel 2030, saranno almeno 25.4 miliardi.

Tutti questi dispositivi producono una quantità di dati enorme, e grazie al Cloud, è possibile gestirla. Il problema è che in alcune applicazioni, questo tipo di architettura non è efficace, o meglio, c'è bisogno di processare e filtrare i dati prima di salvarli sul Cloud.

Deployment models

La distribuzione dei servizi IoT viene tradizionalmente implementata secondo due modelli:

- **IoT + Edge:** prevede la processazione dei dati in locale, o meglio al confine tra il dispositivo stesso e il cloud. Il vantaggio è quello di elaborare i dati nello stesso luogo dove vengono prodotti e la condivisione nel cloud di una parte ridotta di dati in quanto già elaborati. Lo svantaggio sta nella limitata capacità di calcolo e di storage.
- **IoT + Cloud:** è ampiamente utilizzato e prevede l'invio dei dati nel cloud per l'elaborazione. Il vantaggio di questo modello è la spropositata potenza di calcolo e storage di cui dispone il cloud e il non dovere implementare funzionalità di elaborazione sul dispositivo in locale. Lo svantaggio risiede nella necessità del collegamento alla rete, nella latenza e nei colli di bottiglia dovuti alla larghezza di banda di cui si dispone.

Fog computing

Gli obbiettivi del fog computing sono quelli di estendere il cloud attraverso i dispositivi IoT per supportare meglio le applicazioni IoT sensibili alla latenza e affamate di larghezza di banda.

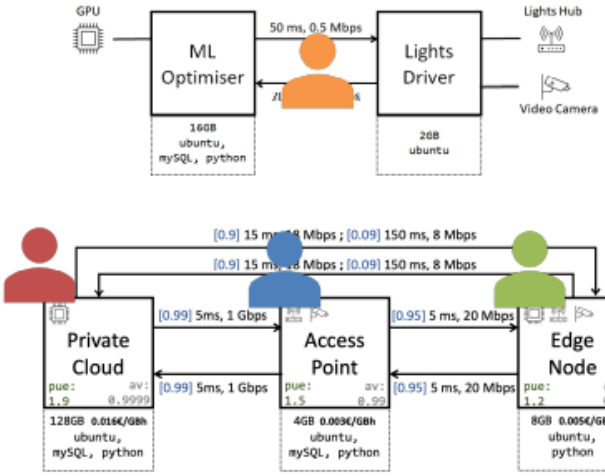
L'idea è avere un'infrastruttura di nodi eterogenei distribuiti in mezzo tra IoT e Cloud per migliorare la capacità di calcolo e di storage. Questi nodi eterogenei possono essere: piccoli datacenter, grandi server, laptop, cellulari, ... Il fog computing aiuta le applicazioni che sono sensibili alla latenza e le applicazioni che hanno bisogno di una ampiezza di banda significativa per fornire una certa qualità del servizio.

Ci sono molte sfide aperte tra cui:

- **Adaptive application deployment**: come mandare in esecuzione un'applicazione multiservizio in una infrastruttura grande distribuita e eterogenea non è semplice, soprattutto se si vuole tenere conto dei vincoli come qualità del servizio, sicurezza, ...
- **Application management**: la gestione delle applicazioni durante il ciclo di vita del sistema.
- **Lightweight monitoring**: il monitoraggio distribuito su infrastrutture offerte da provider diversi.
- **Privacy/security/trust**: se l'infrastruttura è offerta da provider diversi ci possono essere problemi di sicurezza e privacy.

Sviluppo di applicazioni attraverso il Fog

Quello che succede è che si deve tenere conto di molti aspetti, tra i quali, la gestione dell'infrastruttura del fog:



Inoltre, saranno necessari vari strumenti per migliorare più aspetti del funzionamento dell'applicazione, tra i quali **costi, QoS, sicurezza e consumo energetico del Fog**.

Le applicazioni di "nuova generazione" sono tipicamente applicazioni multiservizio, quindi composti da più pezzi, che devono essere mandati in esecuzione sui nodi di una infrastruttura eterogenea. Le applicazioni in generale hanno dei requisiti hardware, software, e vincoli di qualità del servizio. L'infrastruttura ha tre caratteristiche importanti: eterogeneità, grandezza e dinamicità (ci sono dei nodi che possono entrare e uscire liberamente dalla rete dato che ci possono essere dispositivi mobili).

Un primo problema è il **placement**, ovvero come si fa a posizionare i servizi di un'applicazione in maniera opportuna su una infrastruttura tenendo conto dei parametri che abbiamo descritto sopra. Un esempio di approccio è il **declarative placement**: linguaggio dichiarativo, basato sulla logica del primo ordine. La figura mostra come si può definire in modo semplice il problema del placement. Il linguaggio si chiama **prolog** e la notazione :- significa 'se'. Può essere letto come 'il predicato servicePlacement è vero se è vero service, node, ...

Declarative backtrack search FogBrainX

https://github.com/di-unipi-socc/fogbrainx/

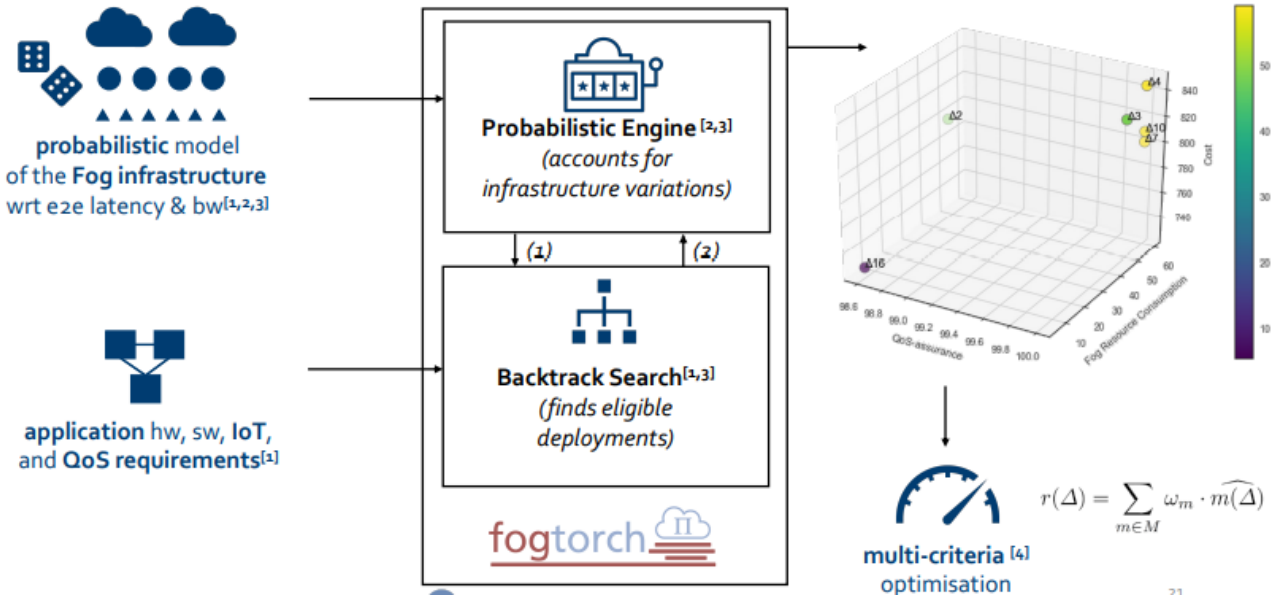
```
placement([S|Ss],P,(AllocHW,AllocBW),Placement) :-
    nodeOk(S,N,P,AllocHW), % checks sw, IoT and cumulative hw requirements
    linksOk(S,N,P,AllocBW), % checks latency and cumulative bw requirements
    placement(Ss,[on(S,N)|P],(AllocHW,AllocBW),Placement).
placement([],P,_,P).

nodeOk(S,N,P,AllocHW) :-
    service(S,SWReqs,HWReqs,IoTReqs),
    node(N,SWCaps,HWCaps,IoTCaps),
    swReqsOk(SWReqs,SWCaps),
    thingReqsOk(IoTReqs,IoTCaps),
    hwOk(S,N,HWCaps,HWReqs,P,AllocHW). % checks cumulative hw requirements on N
```

Un altro aspetto importante è la **dinamicità dell'infrastruttura**: non si può supporre che la capacità di un nodo sia costante nel tempo oppure che la connessione tra due nodi in termini di latenza sia sempre la stessa. Quindi le caratteristiche offerte dall'infrastruttura variano nel tempo. Per tener conto di questa variazione si possono usare dei modelli probabilistici utilizzando delle distribuzioni di probabilità. Nella figura si vede che per il 90% del tempo cabinetServer offrirà 4 unità di hardware mentre per il 10% ne offrirà soltanto 1.

```
0.9::node(cabinetServer, [ubuntu], 4, ..., ....).
0.1::node(cabinetServer, [ubuntu], 1, ..., ....).
...
0.97:: link (accessPoint, cabinetServer, 13, 4).
0.03:: link (accessPoint, cabinetServer, 75, 0.5).
```

Si deve anche assicuraci di garantire una certa **QoS**, sempre in relazione alla dinamicità della struttura. Il calcolo probabilistico viene esteso con più criteri per ottenere una stima più esatta della necessità dell'applicazione in termini di requisiti di QoS e consumo delle risorse del Fog.

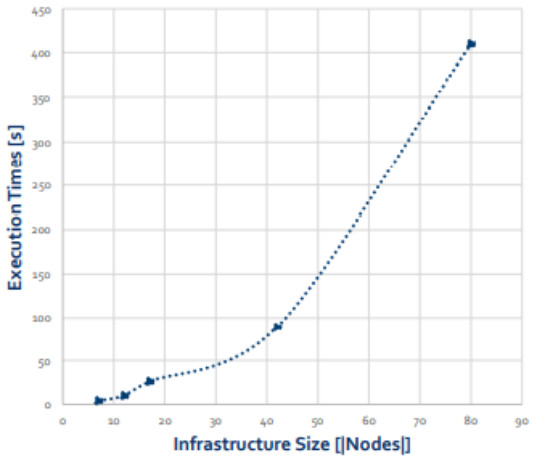


In generale si deve capire in quali nodi posizionare o spostare alcuni servizi dell'applicazione, per garantire un migliore funzionamento.

Id	Placement	QoS-assurance
P_1	Lights Driver → Edge Node ML Optimiser → Private Cloud	0.86
P_2	Lights Driver → Access Point ML Optimiser → Private Cloud	0.97

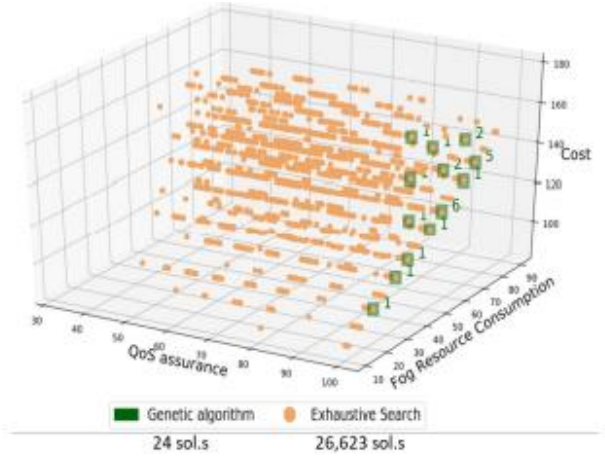
Ad esempio, in questa figura, possiamo supporre che, nonostante le variazioni dinamiche della rete (disponibilità dei nodi, larghezza di banda, latenza), il placement P2 sarà in grado di venire incontro il 97% delle volte ai requisiti della nostra applicazione.

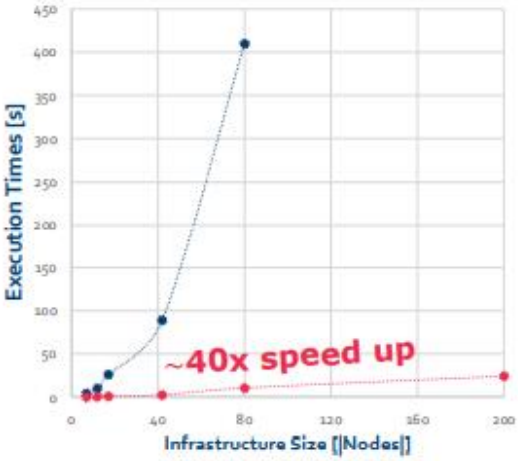
Il grande problema di questo modello è la **scalabilità**. Si può studiare che la complessità di un'applicazione è del tipo $O(|\#\text{ nodi}|^{|\#\text{ servizi}|})$, facendo crescere velocemente la richiesta di capacità di calcolo.



*4 application services

La soluzione è una nuova combinazione di FogTorchπ **Monte Carlo (MC)** con un **Algoritmo Genetico (GA).**



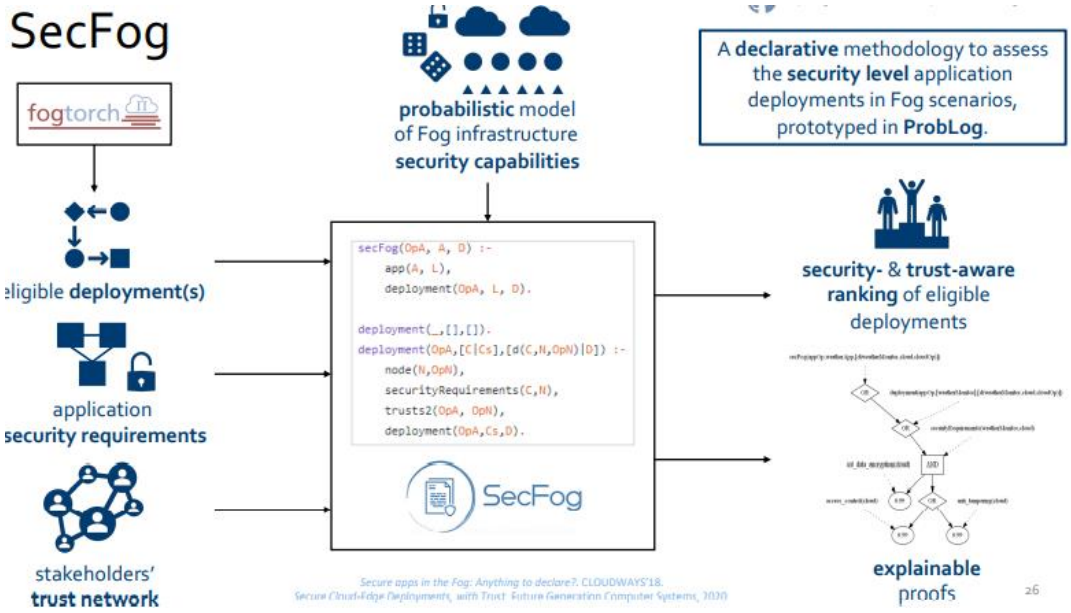


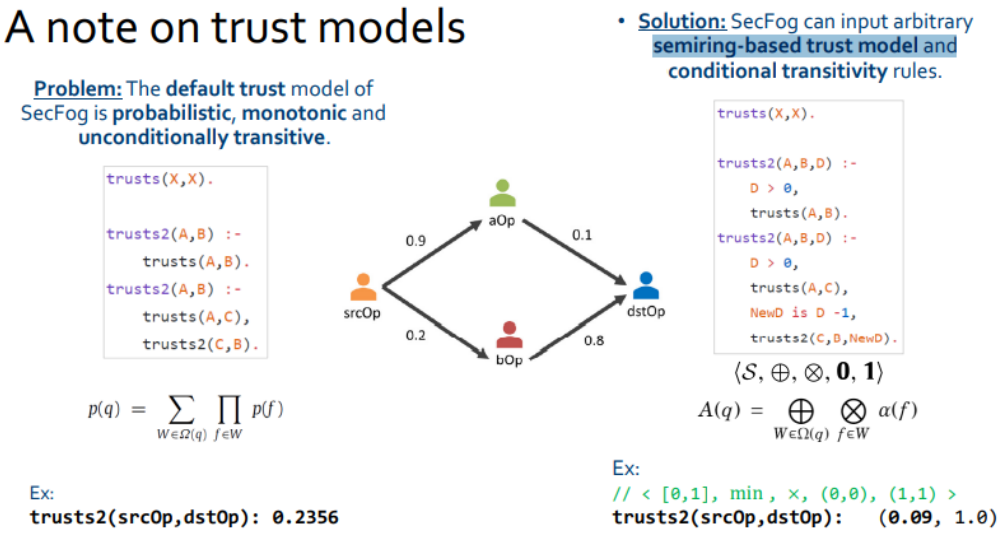
L'ultimo aspetto da menzionare è **il trust**: l'infrastruttura continua tra IoT e Cloud non è di un solo proprietario, ma ci sono diversi provider, questo significa che il deployment di un'applicazione in generale può richiedere che parti dell'applicazione vadano in nodi operati da provider diversi. Qui è importante riuscire a considerare le relazioni di confidenza (trust relations), dove si associano una percentuale di quanto si fida di un certo provider.

SecFog modellizza lo studio del trust di una cerca configurazione, tenendo conto della fattibilità di uno sviluppo, dei requisiti di sicurezza della applicazione e della rete di fiducia degli stakeholder, restituendo un ranking delle soluzioni sviluppabili in base al livello di fiducia garantito.

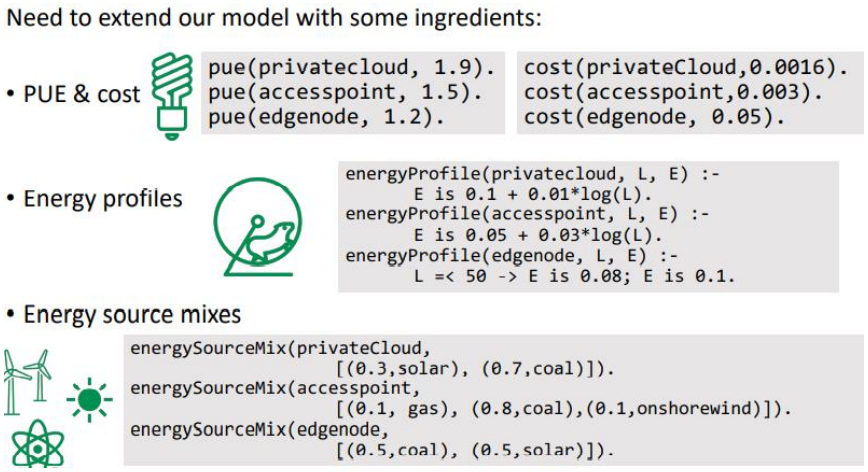
Id	Placement	QoS-assurance	Security and trust
P_1	Lights Driver → Edge Node ML Optimiser → Private Cloud	0.86	0.67
P_2	Lights Driver → Access Point ML Optimiser → Private Cloud	0.97	0.71

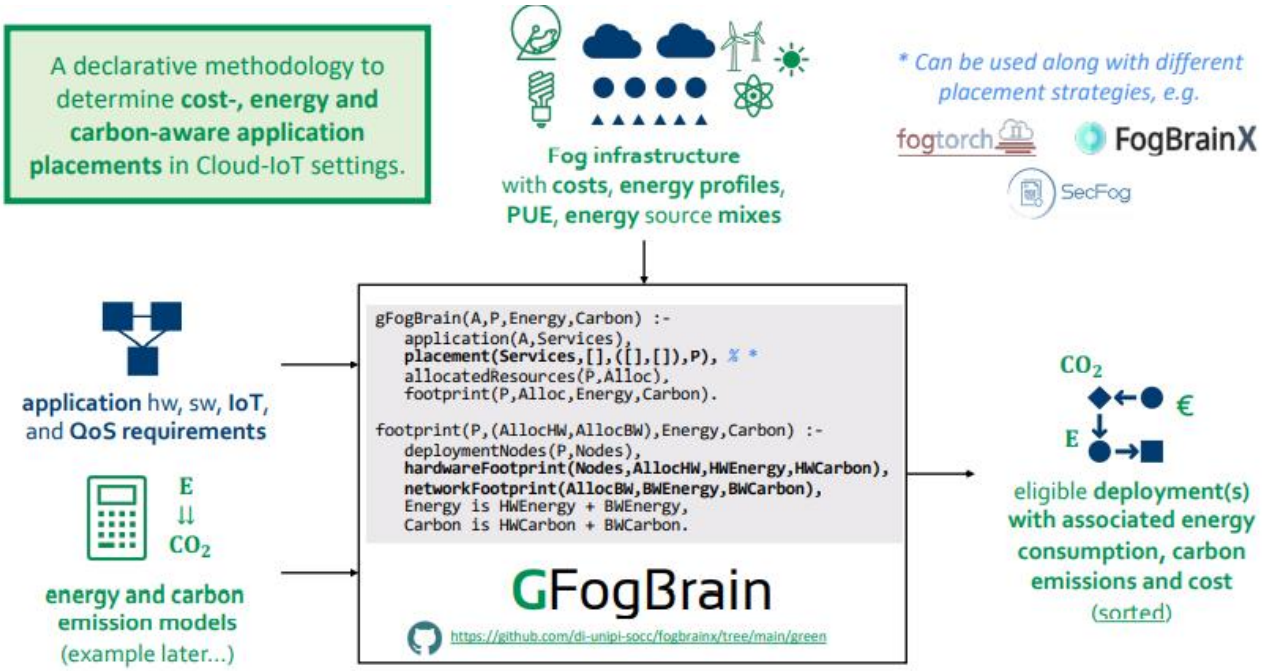
Ad esempio, considerando questi dati, il placement P2 ci garantirebbe, oltre ad un QoS maggiore, anche un maggiore indice di affidabilità.





Resta infine da valutare quale tra i placement implementabili mi garantisca i costi operazionali minimi e minimizzi le emissioni di CO_2.





Energy and carbon emissions (an example)

- We use the formulae of [1] to estimate energy consumption of service s at node n

$$E_s = \Delta E_{n,s} \times \mathrm{PUE}_n \,[\mathrm{kWh}]$$

- The carbon emissions due to such consumption

$$I_s = E_s \times \sum_i p_i \mu_i \;\; [\mathrm{kgCO_2}]$$

- We estimate also network energy consumption and emissions related to network transmission

$$E_N = 450 \times 0.00008 \times M \;[\mathrm{kWh}] \text{ and } I_N = 0.475 \times E_N \;[\mathrm{kgCO_2}]$$

Id	Placement	QoS-assurance	Security and trust	Energy cons.	CO_2 emissions	Operational cost
P_1	Lights Driver → Edge Node ML Optimiser → Private Cloud	0.86	0.67	0.60 kWh	0.29 $kgCO_2$	0.0356 €/h
P_2	Lights Driver → Access Point ML Optimiser → Private Cloud	0.97	0.71	0.63 kWh	0.32 $kgCO_2$	0.0316 €/h

Nello stesso esempio utilizzato prima si nota come il placement P2 garantisca migliori QoS, sicurezza e costi operazionali, a scapito però delle emissioni di CO_2.

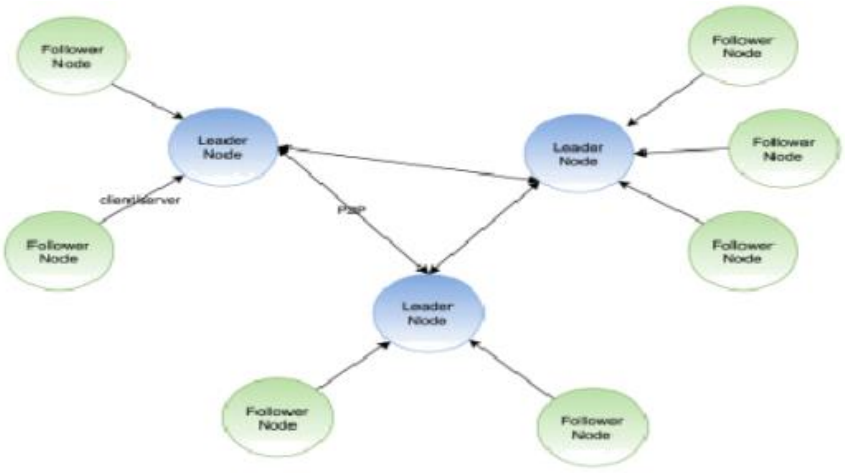
Monitoraggio del Fog

Quando si implementa un meccanismo di monitoraggio, si genera una degradazione della rete. L'idea è di avere un meccanismo di monitoring distribuito, leggero, resistente ai malfunzionamenti e che esegua un monitoraggio:

- delle risorse hardware disponibili

- livello di qualità del servizio *end-to-end* tra i nodi
- dei dispositivi IoT collegati.

Un modo per farlo è utilizzare il **modello super peer**, ovvero distribuire una serie di agenti per il monitoraggio organizzati con una struttura a gruppi, dove più *nodi follower* sono associati al *nodo leader*, mantenendo il numero di leader limitato.



In questo modello le connessioni tra i leader saranno di tipo *P2P* mentre tra i leder e i follower saranno del tipo *client/server*.

Inoltre, questo modello avrà una scalabilità limitata, si cerca infatti di mantenere il numero dei leader nell'ordine di *sqrt(N)*, facendone crescere così il numero abbastanza lentamente. Infine, avremo due parametri da monitorare:

- **latenza e banda intra-group sono misurati in maniera diretta**
- **latenza e banda inter-group sono solo stimati**
- *Lat(x,y) = Lat(x,L(x)) + Lat(L(x),L(y)) + Lat(L(y),y)*
- *BW(x,y) = min(max BW(x,n(x), max BW(y,n(y), BW(L(x),L(y))*
-