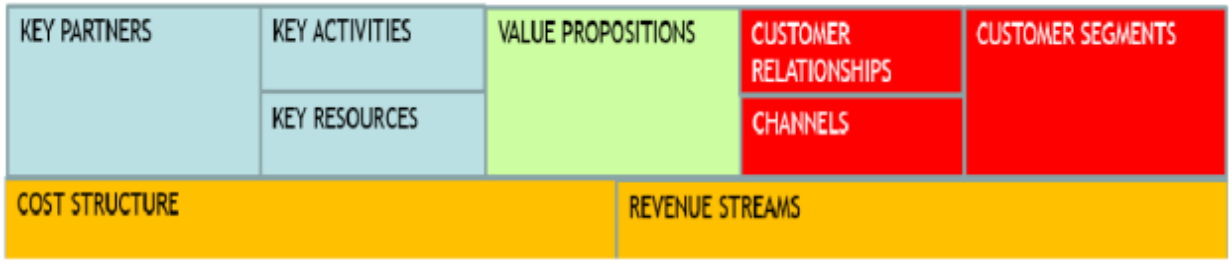
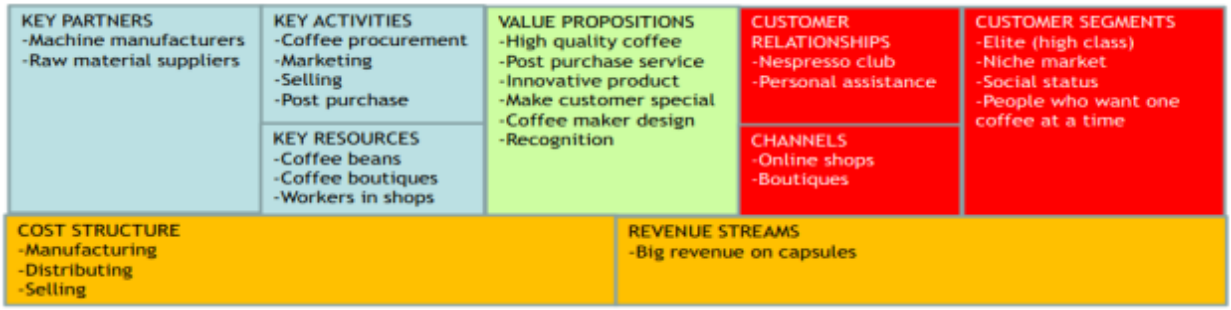
Cheat Sheet (2°comptino)

Business model: descrive come un'azienda crea, consegna e cattura valore. I business model canvas sono una rappresentazione grafica in cui un modello di business viene rappresentato.

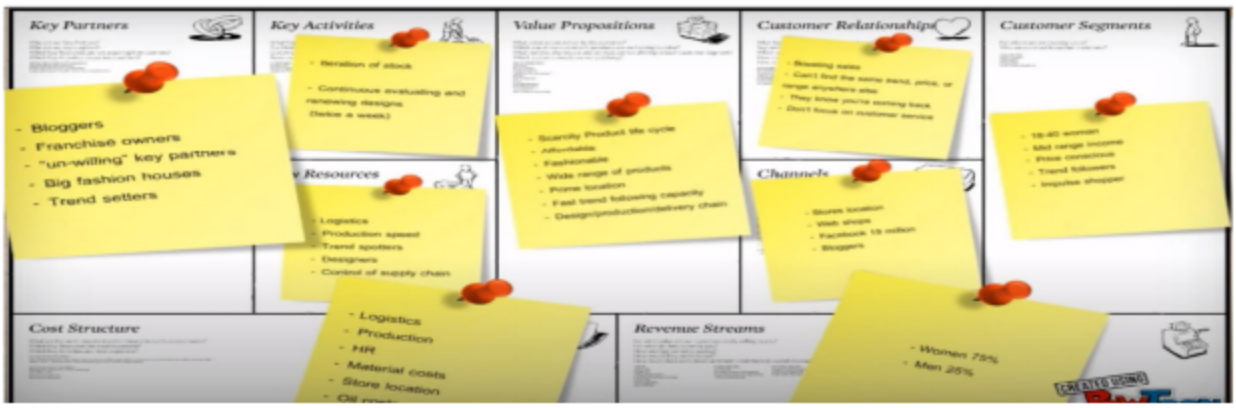


- *Value proposition:* qual è l'aspetto che distingue il servizio che stiamo offrendo? Perchè dovrebbe risolvere qualche problema o interessare? Descrive l'insieme dei prodotti e servizi che creano valore per uno specifico customer segment
- *Customer relationship:* descrive il tipo di rapporto che l'azienda instaura con il cliente (fidelizzazione)
- *Customer segment:* indica il target di mercato e quali tipi di clienti si vuole attirare
- *Channels:* sono i canali con cui il servizio arriva ai clienti
- *Key partners:* è bene cercare dei partner, che però se falliscono rischia di cadere tutto
- *Key activities:* è l'attività principale della nostra azienda
- *Key resources:* sono le risorse chiave che eroghiamo
- *Cost resources:* indica quali sono le uscite
- *Revenue streams:* indica i flussi da cui arrivano le entrare descrivendone anche il tipo

Nespresso: nel 1976 Nestlè dominava il mercato mondiale del caffè istantaneo con un prodotto che si chiamava Nescafè. Nel 1988 il nuovo CEO cambiò business model (in particolare il customer segment e aggiunse le pubblicità) poichè le macchinette dovevano essere indirizzate ad impiegati di alto livello e in generale famiglie bnestanti. Aveva due canali di acquisto: shop online e boutique solitamente nel centro delle città vicino ai negozi di lusso. Dal punto di vista della sostenibilità ambientale Nespresso produce una grandissima quantità di materiale difficilmente riciclabile.



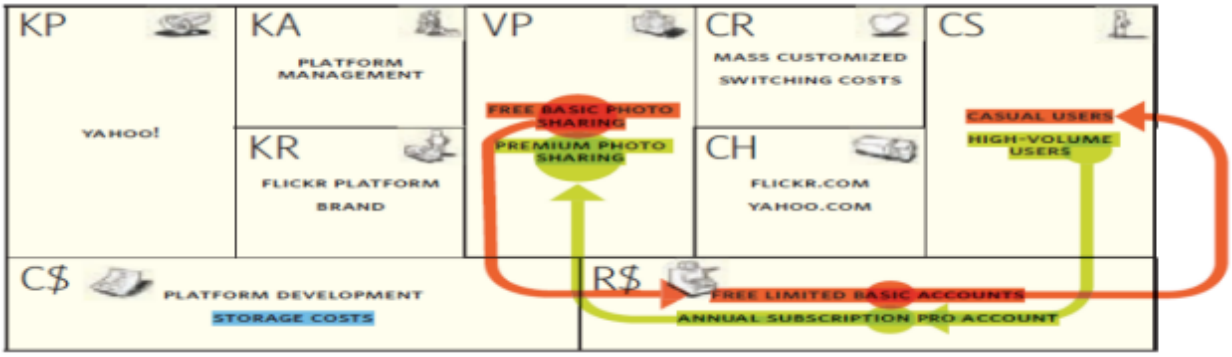
Zara: ha deciso di produrre abiti in base alla preferenza dei clienti effettuando un monitoraggio costante dei consumi degli utenti. Per questo motivo vi è un rinnovo costante della linea dei prodotti. Zara fornisce settimanalmente quello di cui ha bisogno un certo punto vendita, e così facendo nessun punto vendita utilizza un magazzino dove avere le scorte dei prodotti. I prezzi sono molto più convenienti rispetto ai negozi di alta moda, ma i punti vendita vengono collocati vicini ad essi.



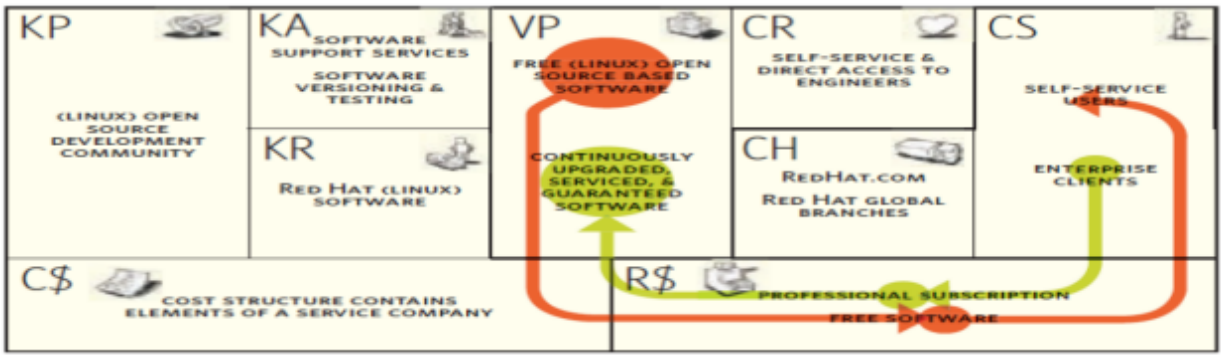
Freemium: consiste in una suddivisione dei servizi, una parte gratis che comprende i servizi di base (free) e una parte a pagamento che comprende i servizi aggiuntivi

(premium). Chi adotta il modello Freemium deve porre molta attenzione al costo che porta al passaggio dalla versione free a quella premium. Il 10% degli utenti fa il passaggio a premium.

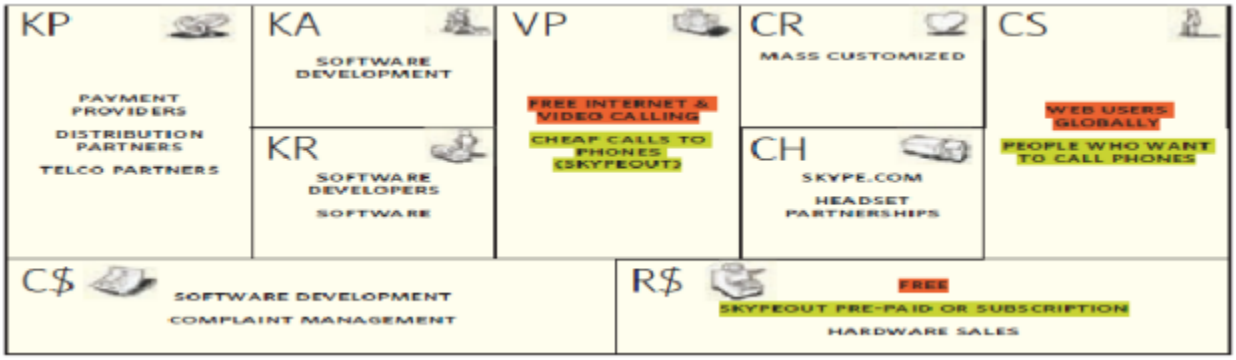
Flickr: è una piattaforma che permette di caricare e condividere immagini. Gli utenti hanno uno spazio di memorizzazione e funzioni limitatati. Gli utenti premium, pagando una quota mensile hanno tutte le funzionalità del servizio. Flickr è nato nel 2002, venne acquistato nel 2005 da Yahoo! per 25 milioni e riacquistato nel 2017 da Verizon per 4 miliardi (Yahoo! aveva rifiutato 44 miliardi da Microsoft nel 2009) per poi diventare parte di SmugMug nel 2018.



RedHat: le aziende che volevano usare un software open source, avevano il timore che esso poteva fallire o che nessuno poteva 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, ... RedHat è stata acquisita per 34 miliardi da IBM nel 2019.



Skype: tutti gli utenti possono 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 nel 2011 da Microsoft per 8.5 miliardi. P2P routing o chiamate gratis attraverso Internet.

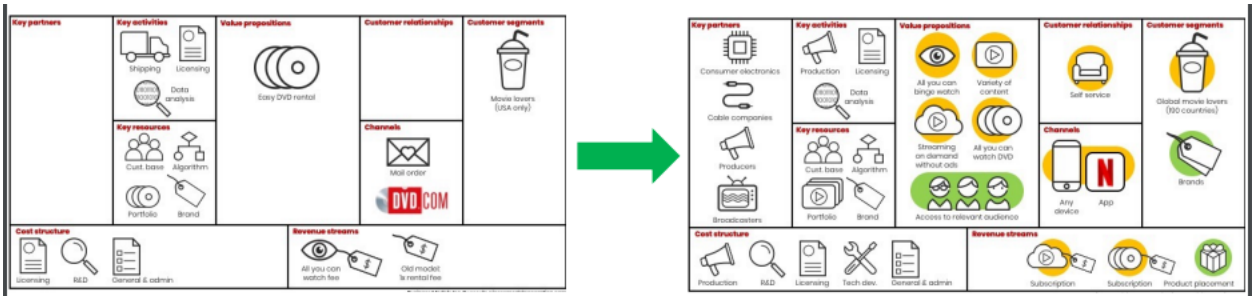


Dropbox: ha 500 milioni di utenti di cui 12 milioni paganti ed il servizio è così suddiviso:

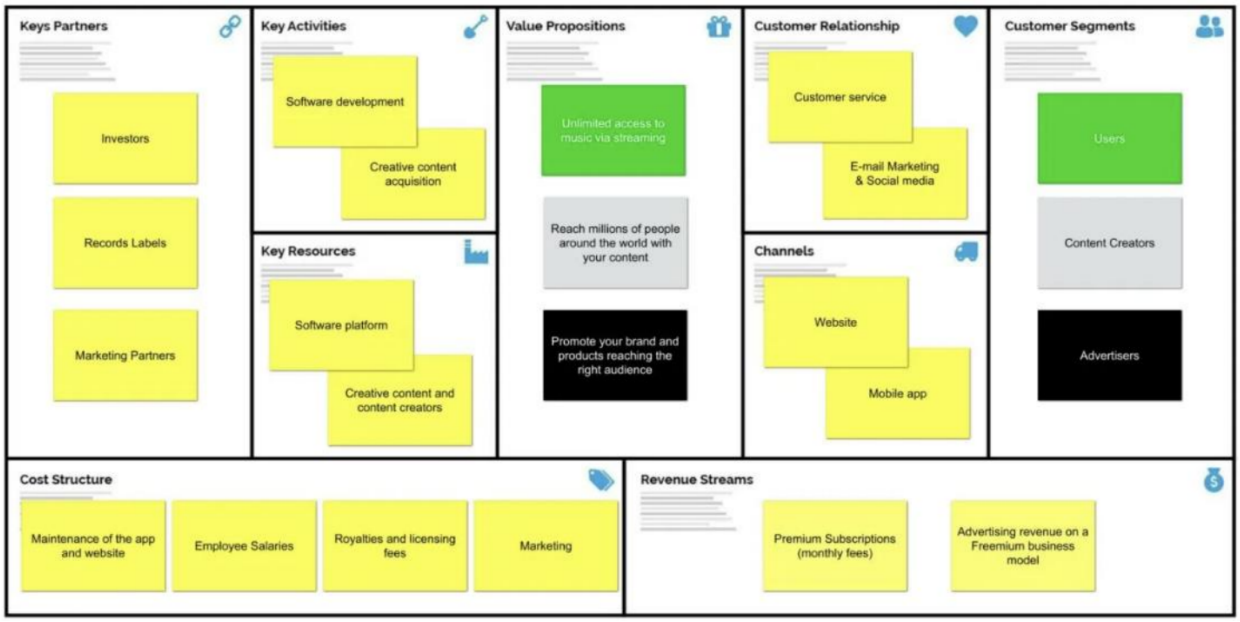
- Privati
  - Free: 2GB
  - Plus: 1TB a 9.99€ al mese
  - Professional: 2TB a 19.99€ al mese
- Aziende
  - Standard: 3TB a 10€ al mese
  - Advanced: no limit a 15€ al mese
  - Enterprise: contact us

	For individuals		For teams		
	Professional €16.58/month Try for free or purchase now	**Professional + eSign** €25.99/month Try for free or purchase now	**Standard** €10/user/month Try for free or purchase now	**Standard + DocSend** €45/user/month Purchase now	**Advanced** €15/user/month Try for free or purchase now
Dropbox core features					
Storage	3 TB (3,000 GB)	3 TB (3,000 GB)	5 TB (5,000 GB)	5 TB (5,000 GB)	As much space as needed
Best-in-class sync technology	✔	✔	✔	✔	✔
Any time, anywhere access	✔	✔	✔	✔	✔
Easy and secure sharing	✔	✔	✔	✔	✔
256-bit AES and SSL/TLS encryption	✔	✔	✔	✔	✔
Legally binding eSignature requests within Dropbox	Up to 3 per month	Unlimited	Up to 3 per month	Up to 3 per month	Up to 3 per month
5 custom eSignature templates		✔			
Industry-leading eSignature security and privacy standards		✔			

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.



Spotify: per pagare meno di royalty ha ideato una strategia che permette di scegliere dove reperire il brano in base alla localizzazione dell'utente poichè i costi variano in base al paese d'ascolto. Ha 217 milioni di utenti attivi e 100 milioni questi sono utenti premium.



Business model generation: i modelli di business più rivoluzionari sono quelli che hanno ignorato lo status quo, ovvero rompere con il passato.

- Cosa succede se offriamo un servizio gratuito per memorizzare foto a tutto il mondo? - Flickr
- Cosa succede se offriamo software gratuito a tutto il mondo? - RedHat
- Cosa succede se offriamo chiamate gratuite a tutto il mondo? - Skype
- Cosa succede se offriamo storage gratuito a tutto il mondo? - Dropbox
- Cosa succede se offriamo musica gratuita a tutto il mondo? - Spotify
- Cosa succede se offriamo un motore di ricerca gratuito? - Google

Nell'innovazione del business ci sono vari possibili epicentri:

- *Resource Driven:* partire dalle risorse che ha già l'azienda, come 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 Cemex che in un momento in cui il cemento era garantito in 48 ore è riuscita a farlo in 4 ore.

- *Custom Driven:* basarsi sulle necessità dei clienti, come 23andMe che offriva test per DNA ai singoli clienti privati.

- *Finance Driven:* basarsi sull'aspetto finanziario, come Xerox che introdusse un modello in cui il costo delle fotocopie veniva offerto in leasing e venivano garantite 2000 copie gratuite.

- *Multiple Epicenter Driven:* epicentri multipli.

Analisi CVR: permette di comprendere gli equilibri esistenti tra costi, volumi d'affari e reddito generato. Consente di effettuare analisi *what if* e simulazioni per comprendere come i vari fattori si influenzano a vicenda, permette di calcolare i rischi operativi e di valutare i modelli alternativi. Si concentra su:

- Prezzi dei prodotti/servizi
- Volumi di attività
- Mix delle vendite
- Costi variabili unitari
- Costi fissi totali

Amazon: fondata da Jeff Bezos con il nome *Cadabra* nel 1994, era un biblioteca online. Il business plan era di non aspettarsi profitto per i seguenti 4-5 anni, il primo profitto fu di 5 milioni e arrivò nel 2001. Amazon cerca continuamente di essere innovativa facendo partire dei progetti come:

- Amazon Go: negozi fisici senza casse
- Amazon Prime Air: una volta ordinato un prodotto viene recapitato entro 30 minuti da un drone

Per quest’ultimo progetto Amazon ha dei problemi soprattutto per la vendita in Europa perchè 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 è 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. Amazon ha 300 milioni di utenti (2017) e 2 milioni di venditori di Amazon (2018), circa 560 000 dipendenti (2018) e 3724 miliardi di entrate all’anno detenendo il 47,8% del mercato cloud. Entrate di amazon in miliardi nel 2022:

- 220 da negozi online
- 117.72 da retail o venditori di terze parti
- 80.10 da AWS
- 35.22 da iscrizioni
- 18.96 da negozi fisici

Google: inizia la sua storia con il famoso motore di ricerca gratuito. Le aziende pagano Google per mostrare la propria pubblicità attraverso il motore di ricerca. È riuscito a dominare il mercato della pubblicità grazie al *custom advertising* dove Google riesce a garantire di mostrare la pubblicità ai soli clienti potenzialmente interessati.

REpresentational Sate Transfer (REST): originariamente introdotto come stile architettonico, sviluppato come un modello astratto dell’architettura web per guidare la riprogettazione e definizione di HTTP e URI, è un protocollo che usa HTTP per la comunicazione. Ogni azione risulta in una transizione allo stato successivo dell'applicazione di trasferire una rappresentazione dello stato all'utente. Si basa sui seguenti principi:

- Identificazione delle risorse tramite URI
- Interfaccia uniforme: i client invocano metodi HTTP per creare/leggere/aggiornare/eliminare risorse. Si usano POST e PUT per creare e aggiornare lo stato della risorsa, si usa DELETE per eliminare una risorsa e GET per ottenere lo stato corrente di una risorsa.

- Messaggi auto descrittivi: le richieste contengono informazioni sufficienti per elaborare il messaggio. Le risorse sono disaccoppiate dalla loro rappresentazione in modo che sia possibile accedere al contenuto in una varietà di formati (HTML, XML, JSON, …)
- Interazioni tramite hyperlinks con stato: ogni interazione con una risorsa è senza stato. Il server non contiene lo stato del client, ogni stato della sessione è mantenuto nel client. Le interazioni con stato si basano sul concetto di trasferimento di stato esplicito.

OpenAPI: (progetto collaborativo della Fondazione Linux) mira a creare un formato di descrizione standardizzato e indipendente dal fornitore delle API REST. Inizialmente conosciuto come Swagger, è un semplice linguaggio di descrizione (basato su JSON) per specificare gli endpoint dell'API HTTP, come vengono utilizzati e la struttura dei dati che entra ed esce.

OAS Tools: è un ecosistema di strumenti server-side per la creazione di servizi REST compatibili con le specifiche OpenAPI. Sono inoltre presenti tool per la generazione automatica delle specifiche a partire da esempi concreti di risorse.

Il comando `oas-tools init` (specificando <<server>>) permette di generare lo scheletro di un microservizio in maniera automatica.

Il comando `npm start` permette di lanciare il server una volta fatto l’accesso alla cartella da terminale.

Microservizi: sono una tecnologia molto importante nello sviluppo del software, soprattutto utilizzando Cloud (Amazon, Netflix, Spotify, Google, Ebay, Facebook, Twitter, LinkedIn). I due motivi principali per usare i microservizi sono:

- Riducono il lead time per nuove feature e aggiornamenti: la fase di rebuilt e redeployment è molto costosa e aggiungere una feature significa rifare rebuilt e redeployment, grazie ai microservizi viene fatta più velocemente.
- Riuscire a scalare in maniera efficace: aumentando la capacità di risposta dell’applicazione. Un modo è creare delle repliche dell’applicazione ma se la criticità dell’aumento del numero delle richieste non è per tutta l’app non è necessario replicare interamente tutto, i microservizi aiutano a scalare solo quella parte. 422

milioni di persone usano Spotify una volta al mese e 182 milioni sono iscritti. Nel 2022 Netflix aveva 222 milioni di iscritti in tutto il mondo.

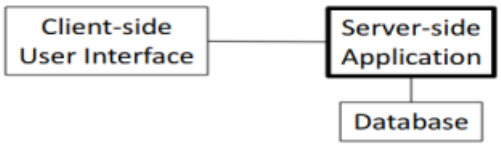
Non bisogna considerare i microservizi finchè non abbiamo un sistema troppo complesso per essere gestito da un monolite.

Lead time: è il tempo che passa da quando un progettista incomincia ad immaginare una funzionalità e la disegna sulla lavagna, al momento in cui, il primo utente da il primo click su quella funzionalità. In altre parole è il tempo che ci vuole a rendere utilizzabile una funzionalità o un nuovo aggiornamento.

Scalabilità orizzontale: capacità di scalare replicando un singolo microservizio.

Scalabilità verticale: capacità di scalare replicando tutta l'applicazione.

Monoliti: la tipica architettura di un'applicazione interprise è composta da tre livelli: interfaccia lato client, applicazione lato server e database.



L'applicazione lato server è tipicamente quello che viene chiamato monolite, ovvero un unico grande eseguibile in grado di rispondere alle richieste HTTP, fare query e aggiornare il database e inviare HTML views al client. Questa architettura ha il problema che un piccolo cambiamento nell'applicazione richiede il rebuilding e redeploying dell'intero monolite, cosa che può essere molto costosa, l'altro problema è l'impossibilità di scalare solo una parte dell'app.

Essenza dei microservizi:

- *Service orientation:* le applicazioni sono sviluppate come insiemi di servizi ognuno su un container diverso, che comunicano fra loro con meccanismi semplici (protocolli REST). Oltre alle chiamate sincrone ci sono dei *dumb message bus* come per esempio delle code asincrone. I servizi sono poliglotti (scrivibli in diversi linguaggi). 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.

- *Organizzare i servizi intorno alle 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. 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.
- *Decentralizzazione della gestione dei dati:* permettere ad ogni servizio di avere e gestire il proprio database. Questo comporta però ad 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 per poi ritornare in funzione. Si sacrifica la consistenza per l'efficienza. CAP Theorem: in presenza di una partizione di rete non si possono avere sia disponibilità che consistenza.
- *Independently deployable service:* ogni servizio è indipendente e può essere lanciato senza lanciare gli altri servizi. Si fa il rebuilt solo di una parte di tutta l'app.
- *Scalabilità orizzontale:* si vuole scalare non l'intera app ma solo su quel servizio che problemi di performance.
- *Resistenza ai fallimenti:* il punto più a sinistra è l'API Gateway, se fallisce 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, inducano 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 0.9999^{30} $(30 \text{ servizi}) \times 24 \text{ (ore al giorno)} \times 30 \text{ (giorni al mese)}$ 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 faul tollerance deve essere un requisito. Una soluzione e 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 cercava qualcosa, nel caso di fallimento semplicemente non riceveva niente, cioè riceveva

una cella bianca. Poi l'utente rimette la query e ha successo. Non è un fallimento se l'utente non se ne accorge.

To offer a functionality F, a service synchronously invokes two other mutually independent services, each featuring 90% uptime.
We can predict that F will be probably NOT available for around ___ in a month.

- 6 minutes
- 6 hours
- 6 days

A: $p_{na} = 1 - (0.9 \times 0.9) = 0.19$
6 days!

Chaos Monkey: introdotto da Netflix, permette di testare in modo coraggioso il funzionamento di una applicazione. Lo si puo 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: 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. Con DevOps non si fanno progetti ma prodotti: tu lo hai costruito e tu lo mandi in esecuzione. Quindi l'idea non è di separare ma il team full stack che si occupa di un servizio, lo progetta, lo crea e se lo gestisce.

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. 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 eseguite in contesto condiviso. Un *kubelet* è un agente nodo che è in esecuzione su ogni nodo che prende un insieme di

specifiche pod (principalmente attraverso il server API) e assicura che i container descritti in tali specifiche siano in esecuzioni e ”sani”. La differenza tra i due riguarda la semplicità e la potenza. Swarm è più semplice da installare ed è abbastanza semplice da imparare. Tuttavia Kubernetes offre il meccanismo di auto-scaling, è più robusto in termini di tolleranza ai guasti ed è supportata da CNCF. Quindi se l’applicazione che si deve gestire e relativamente semplice, Docker Swarm probabilmente è la scelta giusta. Se invece l’applicazione raggiunge un livello di complessità considerevole e la dimensione dei cluster non è banale, allora è consigliato Kubernetes.

K8s gestisce l'intero ciclo di vita dei singoli container, avviando e arrestando le risorse secondo necessità, se un container si spegne inaspettatamente, K8s reagisce lanciando un altro container al suo posto. K8s fornisce un meccanismo per consentire alle applicazioni di comunicare tra loro anche se i singoli container sottostanti vengono creati e distrutti. Dato un insieme di container workloads da eseguire e un set di macchine su un cluster, l'agente di orchestrazione del contenitore esamina ogni container e determina la macchina ottimale per programmare quel carico di lavoro.

Minikube dashboard: permette di avere una visione grafica di un cluster kubernetes.

Il comando `minikube start` crea un cluster locale formato da un solo nodo.

Il comando `kubectl apply -f NOME_FILE` permette di lanciare un determinato Pod.

Il comando `kubectl get pod` permette di avere la lista dei Pod in esecuzione.

Il comando `kubectl logs NOME_POD` restituisce informazioni sui log del Pod.

Il comando `kubectl describe pod NOME_POD` è un comando molto potente che ci consente di avere numerose informazioni sul Pod (come l’IP interno).

Il comando `minikube ssh` permette di entrare in una shell con cui interagire con i container.

Il comando `kubectl get all` permette di visualizzare tutte le risorse create da Kubernetes.

Il comando `kubectl delete pod NOME_POD` elimina un determinato Pod.

Il comando `kubectl delete pod --all` elimina tutti i Pod.

Il comando `kubectl get deployment` permette di vedere il contenuto del Deployment.

Il comando `kubectl delete ingress,service,deployment,rs,pod --all` permette di eliminare tutto.

Il comando `kubectl exec -it bash -- /bin/bash` (dopo aver lanciato il Pod) permette di aprire una shell bash all'interno del container.

Il comando `apt update && apt install dnsutils curl` (all'interno della shell bash) permette di installare una serie di tool utili.

Il comando `curl IP_POD:9876/health` permette di accedere al Deployment di quell'indirizzo.

Il comando `nslookup NOME_SERVIZIO` restituisce l'IP del Service.

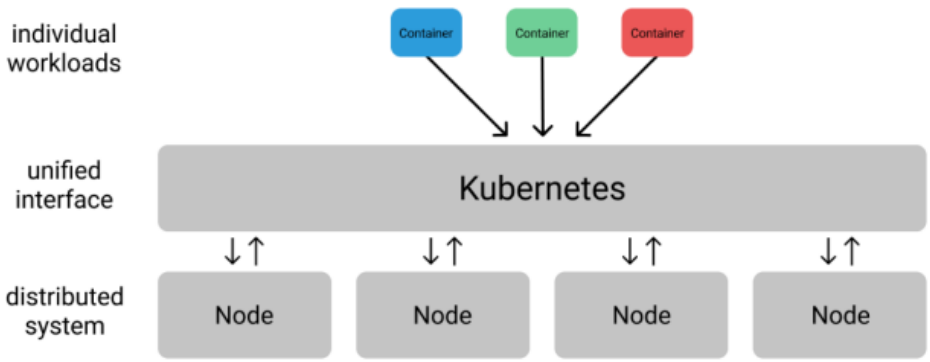
I comandi `minikube addons enable ingress` e `minikube addons enable ingress-dns` installano alcuni addons per minikube.

Il comando `minikube ip` restituisce l'IP dell'Ingress.

Il comando `minikube service` restituisce l'URL di un servizio.

Design principles (K8s):

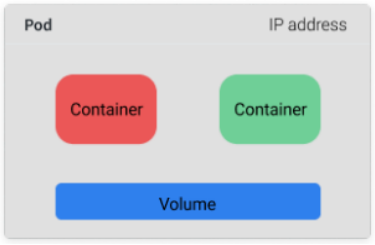
- *Dichiaratività:* definiamo semplicemente lo stato desiderato del nostro sistema. K8s rileverà quando lo stato effettivo del sistema non soddisfa le nostre aspettative e interverrà per risolvere il problema facendo autoriparare il nostro sistema. Lo stato desiderato è definito da una raccolta di oggetti, ogni oggetto ha una specifica in cui si fornisce lo stato desiderato e uno stato che riflette lo stato corrente dell'oggetto. K8s esegue costantemente il polling di ogni oggetto per assicurarsi che il suo stato sia uguale alla specifica, se un oggetto non risponde produrrà una nuova versione per sostituirlo, se lo stato si è discostato dalla specifica emetterà i comandi necessari per riportare quell'oggetto allo stato desiderato.
- *Distribuzione:* K8s fornisce un'interfaccia unificata per l'interazione con un cluster di macchine.



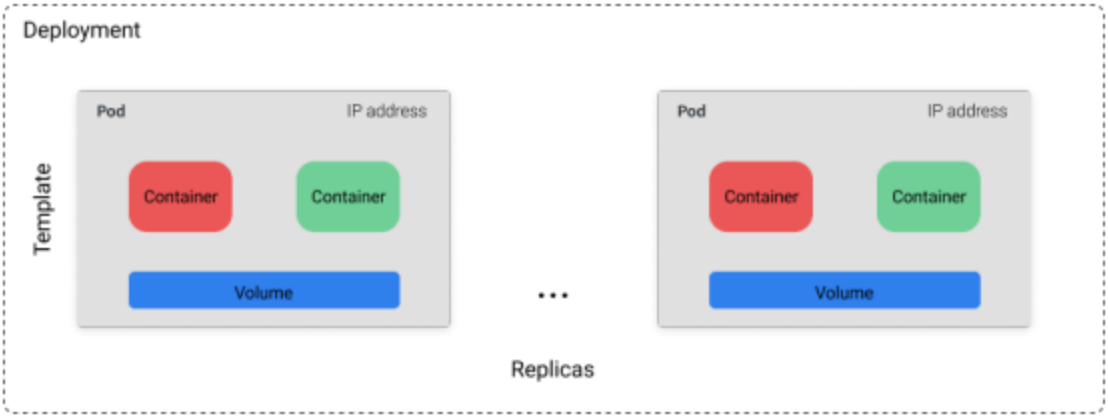
- *Disaccoppiamento:* i container dovrebbero essere sviluppati tenendo presente un'unica preoccupazione: architettura basata su microservizi. K8s supporta naturalmente l'idea di servizi disaccoppiati che possono essere scalati e aggiornati in modo indipendente.
- *Infrastruttura immutabile:* durante il ciclo di vita di un progetto (sviluppo - test - produzione) dovremmo utilizzare la stessa immagine del container. I container sono progettati per essere effimeri, pronti per essere sostituiti da un' altra istanza di container in qualsiasi momento. Il mantenimento di un'infrastruttura immutabile semplifica il rollback delle applicazioni a uno stato precedente, possiamo semplicemente aggiornare il nostro file di configurazione per utilizzare un'immagine precedente.

K8s objects: possono essere definiti in un file *manifest* (YAML o JSON files).

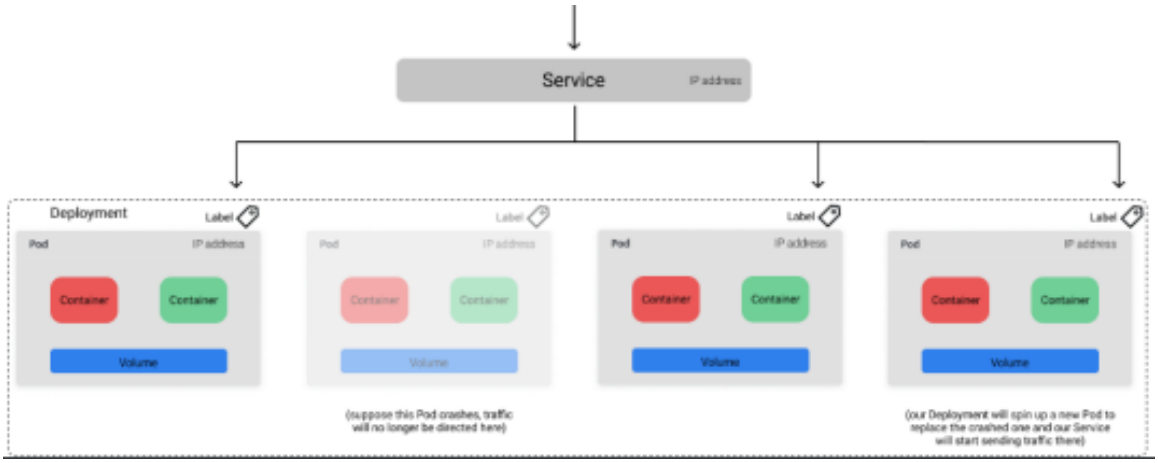
Pod: un pod consiste in uno o più container, un livello di rete condiviso e volumi di file system condivisi.



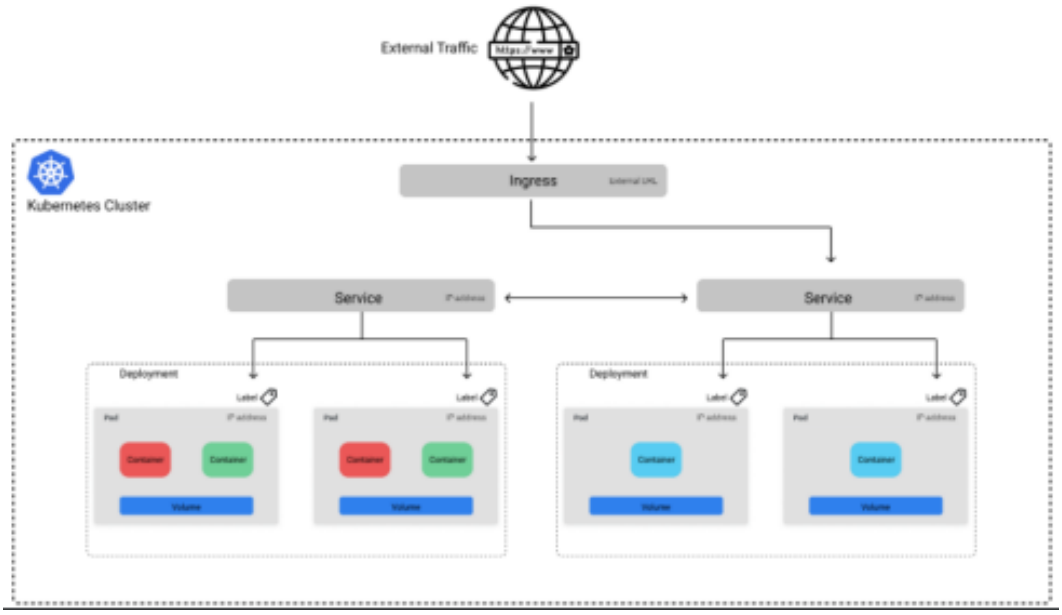
Deployment: include una collezione di Pods definiti da un template e un conteggio delle repliche (numero n di quante copie del template vogliamo eseguire). Il cluster cercherà sempre di avere n Pods disponibili.



Service: ogni Pod è assegnato ad un indirizzo IP unico che possiamo usare per comunicare con esso. K8s service fornisce un endpoint stabile per indirizzare il traffico ai pod desiderati anche se gli esatti pod sottostanti cambiano a causa di aggiornamenti/ridimensionamento/errori. I servizi sanno a quali pod devono inviare il traffico in base alle etichette (coppie di valori-chiave) che definiamo nei metadati del pod.

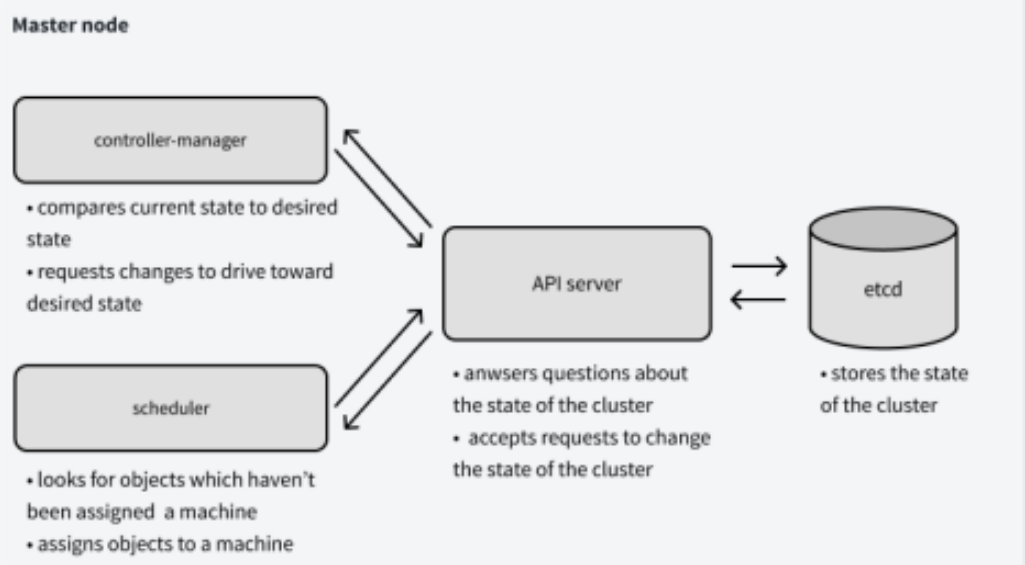


Ingress: l'oggetto Service ci consente di esporre le applicazioni solo dietro un endpoint stabile disponibile per il traffico interno del cluster. Per esporre la nostra applicazione al traffico esterno al nostro cluster, è necessario definire un oggetto Ingress. Possiamo selezionare quali Servizi rendere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

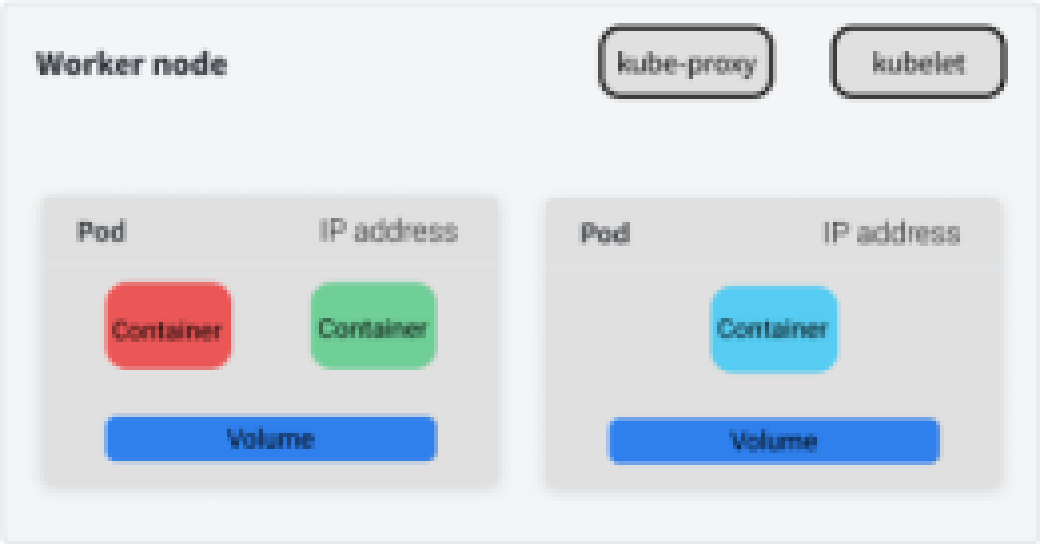


K8s control plane: esistono due tipi di macchine in un cluster:

- Master node (di solito unico), che contiene la maggior parte dei componenti del control plane: l'utente fornisce una nuova/aggiornata specifica dell'oggetto al server API. Il server API convalida le richieste di aggiornamento e funge da interfaccia unificata per le domande sullo stato corrente del cluster. Lo stato del cluster e le specifiche dell'applicazione sono archiviati in un archivio di valori-chiave distribuito ETCD. Lo scheduler determina dove devono essere eseguiti gli oggetti e chiede al server API quali oggetti non sono stati assegnati a una macchina, determina a quali macchine devono essere assegnati quegli oggetti e risponde al server API per riflettere questa assegnazione. Il controller-manager monitora lo stato del cluster attraverso l'API server. Se lo stato attuale differisce dallo stato desiderato farà dei cambiamenti attraverso l'API server per portare il cluster allo stato desiderato.



- Worker node, che esegue i workloads dell'applicazione: kubelet agisce com un nodo agente che comunica con l'API server per vedere quali workloads sono stati assegnati al nodo. È il responsabile di far girare i Pod per eseguire i workloads assegnati. Quando un nodo entra nel cluster, kubelet annuncia l'esistenza del nodo all'API server così lo scheduler può assegnarli un Pod.



Kube-proxy: consente ai container di comunicare tra loro attraverso i vari nodi del cluster.

IoT: è quell'area in cui si studia come dispositivi elettronici, se dotati di connettività di rete, creano una serie di applicazioni innovative. Secondo una stima c'erano circa 10 miliardi di dispositivi IoT attivi nel 2021 e si prevede un aumento di più di 25.4 miliardi nel 2030.

Deployment models:

- IoT + Edge, comporta l'elaborazione locale dei dati, o meglio, al confine tra il Cloud e i dispositivi locali. Il vantaggio è quello di elaborare i dati direttamente nello stesso luogo dove vengono prodotti e una bassa latenza. Lo svantaggio è la capacità limitata per lo storage e l'elaborazione e la condivisione dei dati fra applicazioni.
- IoT + Cloud, è ampiamente utilizzato, i dati vengono mandati sul Cloud per l'elaborazione dove vi è una potenza di calcolo enorme (nessuna limitazione). Lo svantaggio è che le parti dell'applicazioni devono essere collegate a Internet per poter raggiungere la rete e vengono trasferiti molti dati. Altri due svantaggi sono alta latenza e colli di bottiglia per l'ampiezza di banda.

Cloud-Edge Continuum: estende il Cloud verso l'IoT con un’infrastruttura distribuita, eterogenea per ottenere il "meglio di entrambi i mondi":

- Potenza di calcolo
- Connettività
- Latenza

Le applicazione *next-gen* sono containerizzate, basate su microservizi e deployate su un’infrastruttura Cloud-Edge Continuum. I requisiti per un’app sono: hardware, software, QoS e sicurezza e abbiamo bisogno di alcuni tool per trovare il giusto compromesso. Per piazzare un’applicazione sul Cloud-Edge Continuum esistono diversi approcci:

- MILP: difficile da leggere, informazioni non numeriche difficili da codificare, lento da eseguire
- ML: infrastruttura molto dinamica, mancanza di spiegabilità
- Dichiarativo: semplice da leggere, più veloce di MILP, informazioni non numeriche semplici da codificare, facile da spiegare

Per gestire adeguatamente i deployment dell’applicazione nel Cloud-Edge Continuum (dopo il primo deployment) si fa:

- Monitoring (applicazioni e infrastrutture)
- Continuous reasoning: si attua l’analisi differenziale concentrandosi sulle ultime modifiche e riutilizzando i risultati precedentemente calcolati per aumentare/ridurre, migrare, riavviare i servizi applicativi
- Enacting

Approccio dichiarativo: si suddivide in due fasi:

1. Dichiara che cos'è un posizionamento idoneo

> 📢 *Il servizio S può essere piazzato nel nodo N se:*
>
> - *I requisiti hardware di S sono soddisfatti da N*
> - *I requisiti di connessione IoT di S sono soddisfatti da N*
> - *I requisiti software di S sono soddisfatti da…*

2. Lascia che il motore di inferenza lo cerchi

> ⚙️ $:-placement([s1,\ \ldots,\ sm],\ P)$

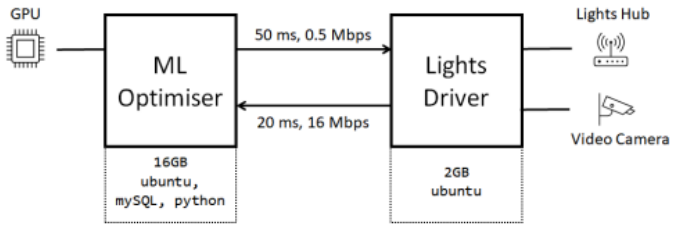
\+ Probabilità di modellare la dinamicità dell'infrastruttura

\+ Semianelli per modellare (non monotoni, condizionalmente transitive) relazioni di fiducia tra i diversi portatori di interesse

---

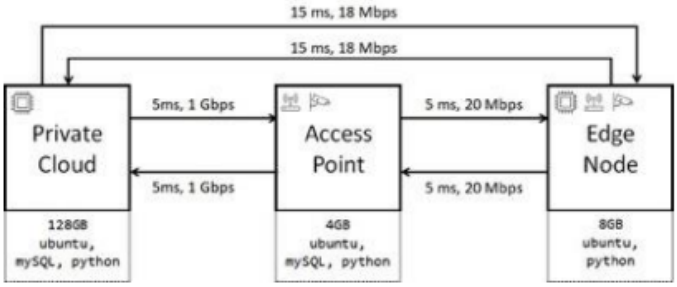
Prolog:

- Variabile: simbolo che inizia con lettera maiuscola `Var`
- Fatto: informazione sempre vera `fact(...)`
- Regola: quando chiami Head esegui Body, se Body è vero allora Head è vero `Head(...) :- Body`
- Operazione Matematica: `Res = X op Y`




```
application(lightsApp, [mlOptimiser, lightsDriver]).
    service(mlOptimiser, [mySQL, python, ubuntu], 16, [gpu]).
    service(lightsDriver, [ubuntu], 2, [videocamera, lightshub]).
    s2s(mlOptimiser, lightsDriver, 50, 0.5).
    s2s(lightsDriver, mlOptimiser, 20, 16).
```

Applicazione modellata




```
node(privateCloud,[ubuntu, mySQL, python], 128, [gpu]).
node(accesspoint,[ubuntu, mySQL, python], 4, [lightshub, videocamera]).
node(edgenode,[ubuntu, python], 8, [gpu, lightshub, videocamera]).

link(privateCloud, accesspoint, 5, 1000).
link(accesspoint, privateCloud, 5, 1000).
link(accesspoint, edgenode, 5, 20).
link(edgenode, accesspoint, 5, 20).
link(privateCloud, edgenode, 15, 18).
link(edgenode, privateCloud, 15, 18).
```

Continuum modellato

```
placements(A,Placements) :-
   findall((Cost,P), (placement(A,P), hourlyCost(P,Cost)), Ps),
   sort(Ps,Placements).

hourlyCost([on(S,N)|P],NewCost) :-
    hourlyCost(P,OldCost),
    service(S,_,HW,_), cost(N,C),
    NewCost is OldCost + C * HW.
hourlyCost([],0).
```

Costo di un placement

```
node(privateCloud,[ubuntu, mySQL, python], 128, [gpu]).
pue(privateCloud,1.9).
energyProfile(privateCloud,L,E) :- E is 0.1 + 0.01*log(L).
totHW(privateCloud,150).
energySourceMix(privateCloud,[(0.3,solar), (0.7,coal)]).
```

Consumo energetico

Per quanto riguarda la CO2 prodotta, applichiamo la seguente formula

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

Dove E_s è l'energia consumata, p_i la percentuale in cui è presente la fonte energetica i nel mix energetico e μ_i le emissioni della fonte i.

CO2 prodotta

Google Datacenter: l'energia è 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 più alta rispetto alle altre aziende. Quando cambiano i drive, perchè hanno smesso di funzionare, dopo averli sostituiti, li distruggono. Google cerca di ridurre l'emissione di CO_2.

UniPi Datacenter: a Pisa ci sono 3 nodi DCI (DataCenter Interconnect) in città a 400 Gbit/sec. Il datacenter più grande si trova a San Piero a Grado. La connessione è *no single point of failure*, ovvero, vi è presente la ridondanza per ridurre al minimo i punti di fallimento. Ci sono più di 9000km di fibra, 64 racks e circa 550 server che si connettono con la rete GARR a 100 Gbit/sec. Per il traffico est-ovest, quello che si riferisce al traffico di dati all'interno della rete di datacenter, vi è un *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 di circa 1.2. Il PUE è un indice per l'efficienza energetica, deve essere superiore a 1 ma il più vicino

possibile a 1, e si calcola $PUE = \frac{total\ facility\ power}{IT\ equipment\ power}$ (indica quanto è green un datacenter e non misura il grado di utilizzo di energia rinnovabile).

Cooling: circa il 40% del consumo di energia si perde nel raffreddamento.

Data Center Infrastructure Managment (DCIM): le azioni principali per gestire un datacenter sono il planning (meeting, organizzazione...), 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 a vari problemi perchè non si conoscono le informazioni né storiche né aggiornate su qual è lo stato dell'infrastruttura.

Cosa non fare in un datacenter:

- Gestione disordinata del cablaggio
- Introdurre cibo o bevande
- Tralasciare la documentazione
- Lasciare la porta aperta tra un accesso e l'altro, neanche per il trasporto di oggetti
- Non prevenire click accidentali di interruttori

Come reagisce il personale ad un problema:

- Rendersi pronti immediatamente
- Gestire il panico
- Seguire la checklist

Checklist: è importante perchè garantisce la qualità delle operazioni e riduce la responsabilità dei dipendenti, serve sia all'operatore che al gestore altrimenti ognuno si comporterebbe in maniera diversa.

Business continuity & Disaster recovery:

1. Mantieni una copia completa dei tuoi dati mission-critical al di fuori della tua regione di produzione
2. Il piano di evacuazione/incendio va prima testato (altrimenti va considerato fallimentare)
3. Garantire che le modifiche alla produzione si riflettano correttamente nel piano di evacuazione/incendio
4. Avere un piano coerente e accessibile anche in caso di grave disastro
5. Avere personale preparato
6. Ricorda la legge di Murphy: "Qualunque cosa possa andare storta, andrà storta". Avere piani di recupero nel caso un piano fallisca

---

Alcuni disastri:

- Un fulmine ha colpito il datacenter di Microsoft e Amazon
- L'uragano Sandy ha causato interruzioni a un numero di imprese operanti intorno a Newark
- Un camion ha sfondato un datacenter in Texas
- Degli scoiattoli hanno mangiato i fili
- Una nave ha calato l’ancora sui fili comunicanti con il datacenter sotto il livello del mare

---

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 pretestate e validate con un sistema di gestione integrato con cui poter eseguire le operazioni principali. La struttura *hyper-converged* invece sfrutta la virtualizzazione: combina la virtualizzazione del server con il networking e lo storage in una singola scatola, quindi riduce di molto i costi hardware ma ha alcune limitazioni (fino al 2016): non si può aumentare lo storage senza aumentare anche la potenza di calcolo, non sono supportate tutte le applicazioni (le applicazioni DB non virtuali non possono utilizzare lo storage virtualizzato), e i relativi software non sono economici.

Application Modernization and Migration (AMM): la modernizzazione delle applicazioni è il processo di prendere le applicazioni esistenti o legacy e aggiornarne l'infrastruttura della piattaforma, l'architettura interna e le funzionalità per sfruttare al massimo l'architettura cloud. Questo processo può essere complesso ma le nuove tendenze come l'iperautomazione, le modernizzazioni del micro frontend, ecc., possono renderlo più semplice e affidabile.

Esistono diverse strategie:

- *Lift and Shift - Rehost:* ricreiamo in cloud l'infrastruttura on premise da VM a VM e il networking si riconfigura in cloud mediante appositi servizi. Vantaggi: pay-as-you-go, infrastruttura estensibile, risparmio. Si utilizza per app che non si possono toccare, quando si ha molta fretta.
- *Lift and Shift - Replatform:* migriamo parte dell'infrastruttura su servizi specifici del cloud provider come Database (Amazon RDS) e Storage (S3). Vantaggi: tutti quelli di Rehost ma più in sicurezza, resilienza e risparmio. Si utilizza per app che non si possono toccare ma su cui ci lasciano il tempo di applicare qualche variazione al contorno.
- *Refactor:* modifichiamo l'architettura dell'app introducendo caratteristiche cloud-native, come Monolite (MSA cloud-native 12factor + containers + serverless), Database (Amazon RDS, Aurora) e Storage (S3). Vantaggi: tutti quelli di Replatform ma in più: scalabilità, performance e funzionalità avanzate. Si utilizza per gestione di picchi di carico (scaling verticale giunto al limite).
- *Repurchase:* alcuni servizi vengono dismessi e ri-acquistati sotto forma di SaaS. Vantaggi: servizio completamente gestito da terzi. Si utilizza per grosse app dalle funzionalità standard quali ERP, CRM ecc… Esempi sono: Netsuite, SAP, Salesforce e Dynamics.
- *Retain / Retire*

Possono essere applicate tutte insieme.

Cloud ibrido: è un modello di cloud computing che utilizza una combinazione di almeno un private cloud e almeno un public cloud. È quindi una piattaforma on premise (data center) + cloud pubblico che gestisce sia VM che container in cloud e localmente. È composta da:

- Data center

- Cloud pubblico (inizialmente opzionale)
- Orchestratore di container/VM ibrido (fa la differenza)

Big Red X: è una distribuzione Kubernetes (Enterprise) che supporta applicazioni cloud-native e applicazioni distribuite cloud-native containerizzate (anche serverless usando lambda), comprende un sistema di osservabilità. Garantisce:

- Supporto Enterprise grazie ad un vendor internazionale
- Gestione di cluster on-premise e on-cloud
- Supporto dell'intero SDLC

---

12 Factor App:

- Pacchettizzazione dell'app con le sue dipendenze (portabilità)
- Configurazione separata dalla logica applicativa e iniettata dall'esterno (flessibilità dello stesso pacchetto in ambienti differenti)
- Startup veloce e shutdown pulito (scheduler-friendly)
- Esternalizzazione dello stato (facilita lo scaling)
- Log come stream di dati
- Microservizi (scaling selettivo)

Progetto di refactoring:

- Assessment dell'AS IS
- Progettazione del TO BE
- Infrastruttura: installazione di Big Red X
- Piattaforma: setup del workflow DevOps e setup dello stack di osservabilità
- Migrazione: analisi delle applicazioni, trasformazione delle applicazioni, aggiunta dello strato middleware