

CLOUD E GREEN COMPUTING

APPUNTI DELLE LEZIONI

Lorenzo Mancinelli

SOMMARIO

1. INTRODUZIONE	6
1.1. QUALITY OF SERVICE (QoS)	6
1.1.1. SERVICE LEVEL AGREEMENT (SLA)	6
1.1.2. GOOGLE COMPUTE ENGINE SPA	6
1.2. RICHIESTA DI UN SERVIZIO	7
1.3. CLOUD	7
1.3.1. MODELLI DI SERVIZIO E DEPLOYMENT	8
1.3.2. OSTACOLI ALL'ADOZIONE DEL CLOUD	8
1.4. NUOVI MODELLI DI BUSINESS	8
1.5. GREEN COMPUTING	9
2. IAAS	10
2.1. SERVER VIRTUALIZATION E HYPERVISOR	10
2.2. IAAS/COMPUTE: AMAZON ELASTIC COMPUTE CLOUD (EC2)	10
2.3. IAAS/STORAGE: AMAZON SIMPLE STORAGE SERVER (S3)	11
2.4. DROPBOX	11
2.5. LABORATORIO AWS	12
2.5.1. COSA VIENE UTILIZZATO	12
2.5.2. STEP 1 – FILE E IAM	13
2.5.3. STEP 2 – S3 BUCKET E EC2	13
2.5.4. STEP 3 – COMPLETARE LO SCRIPT <i>main.py</i>	13
2.5.5. COPIA FILES SU ISTANZA EC2	14
2.5.6. ESEGUIRE SCRIPT SU ISTANZA	14
3. CONTAINERS	15
3.1. MACCHINE VIRTUALI VS CONTAINERS	15
3.2. UN PO' DI STORIA	15
3.3. DOCKER	16
3.3.1. PICCOLA DEMO DIMOSTRATIVA	16
3.3.2. DOCKER COMPOSE	17
3.3.3. DOCKER SWARM	18
3.4. LABORATORIO DOCKER	18
4. PAAS	20
4.1. DEFINIZIONI E VANTAGGI	20
4.2. HEROKU	21
4.2.1. FASI HEROKU	21
4.2.2. TIPI DI DYNOS	22
4.3. MICROSOFT AZURE	22
4.4. OPENSHIFT	23
4.5. CLOUD FOUNDRY	23
4.5.1. CASE STUDY: CSAA INSURANCE	23
4.5.2. CASE STUDY: AUSTRALIAN GOVERNMENT	23
4.6. TROVARE IL PAAS PIU' ADATTO	24
5. VENDOR LOCK-IN	25
5.1. VENDOR LOCK-IN NEL MONDO DEL CLOUD	25
5.2. ESEMPI DI LOCK-IN	25
5.3. PERCHE' VORREI CAMBIARE PROVIDER?	25

5.4. CONSIGLI PER EVITARE/RIDURRE IL LOCK-IN	25
5.5. STANDARD CLOUD	26
5.5.1. OBIETTIVI TOSCA.....	27
6. BUSINESS MODEL.....	28
6.1. BUSINESS MODEL CANVAS	28
6.1.1. ESEMPI DI BUSINESS MODEL CANVAS	28
6.2. FREEMIUM BUSINESS MODEL.....	29
6.2.1. FLICKR	29
6.2.2. REDHAT	30
6.2.3. SKYPE.....	30
6.2.4. DROPBOX	31
6.2.5. NETFLIX	31
6.2.6. SPOTIFY.....	32
6.3. BUSINESS MODEL GENERATION	33
6.3.1. STATISTICHE DI ALCUNE STARTUP	33
6.4. CASO DI STUDIO: AMAZON	34
6.5. CASO DI STUDIO: GOOGLE	35
6.5.1. RIFLESSIONI SUI MOTORI DI RICERCA	35
7. FAAS	37
7.1. AWS LAMBDA.....	37
7.1.1. PREZZI.....	37
7.1.2. ESEMPIO – FUNZIONE LAMBDA IN NODE.JS.....	38
7.1.3. ESEMPIO (PARTE 2) – CONNETTERE LAMBDA AD UN API GATEWAY.....	38
7.2. QUALE FAAS USARE?.....	39
7.3. LABORATORIO FAAS	40
7.3.1. STEP 1 – SETTING UP (1)	40
7.3.2. STEP 2 – SETTING UP (2)	40
7.3.3. STEP 3 – TESTING.....	40
7.3.4. SOLUZIONE.....	41
8. GREEN COMPUTING	42
8.1. INTRODUZIONE	42
8.1.1. PREVISIONI SUL CONSUMO ENERGETICO.....	42
8.2. DATACENTER	42
8.3. IL PROBLEMA DEL PUE	42
8.4. OBSOLESCENZA PROGRAMMATA.....	43
8.5. OBSOLESCENZA PERCEPITA.....	43
8.6. REPORT GREENPEACE: CLICKING CLEAN	43
8.7. TRE AZIONI PRINCIPALI DA PORTARE AVANTI.....	43
8.8. IOT ENVIRONMENTAL IMPACT CALCULATOR.....	44
8.9. CONCLUSIONI	44
8.10. LABORATORIO: INGEGNERIA DEL SOFTWARE SOSTENIBILE.....	44
8.10.1. NOZIONI DI BASE	44
8.10.2. SOSTENIBILITA'	45
8.10.3. SOFTWARE E SOSTENIBILITA'	45
8.10.4. CONSUMO ICT	47
8.10.5. INGEGNERIA DEL SOFTWARE E MODELLO GREENSOFT	48
8.10.6. CASO DI STUDIO: AGGIORNAMENTO DI UN SISTEMA OPERATIVO	49
8.10.7. QUADRO NORMATIVO	53

8.11. LABORATORIO GREEN COMPUTING: JAVA COLLECTION – CONSUMO ENERGETICO ED EMISSIONI.....	54
8.11.1. STRUTTURE DATI.....	54
8.11.2. COMPLESSITA'	54
8.11.3. SCALE UP! – OPERAZIONI PRELIMINARI.....	55
8.11.4. SCALE UP! – ESERCIZIO.....	56
8.11.5. RISULTATI.....	56
8.11.6. CONSIDERAZIONI SUI RISULTATI.....	57
8.11.7. CONCLUSIONI.....	57
9. MICROSERVIZI.....	58
9.1. COSA SONO E PERCHE' USARLI.....	58
9.2. IL PASSATO: MONOLITI	58
9.3. ESSENZA DEI MICROSERVIZI.....	58
9.3.1. PERIODI DI DOWNTIME	59
9.3.2. TEST BRAVELY	59
9.4. DEVOPS	59
9.5. CONCLUSIONE	60
10. ARCHITECTURAL SMELL E SOLUZIONI	61
10.1. DESIGN PRINCIPLES	61
10.2. ARCHITECTURAL SMELLS	61
10.3. UFRESHENER (MICROFRESHENER)	63
11. AZIENDE E MICROSERVIZI	64
11.1. SPOTIFY.....	64
11.1.1. QUANDO E COME È AVVENUTO IL PASSAGGIO AI MICROSERVIZI?	64
11.1.2. VANTAGGI DEI MICROSERVIZI.....	64
11.1.3. SVANTAGGI DEI MICROSERVIZI.....	64
11.1.4. ALCUNI DATI	64
11.1.5. STRUTTURA DEI TEAM.....	65
11.1.6. SPOTIFY ENGINEERING CULTURE	65
11.2. NETFLIX.....	66
11.2.1. PASSAGGIO AI MICROSERVIZI	66
11.2.2. PROBLEMI DI NETFLIX CON I MICROSERVIZI.....	67
11.2.3. OBIETTIVI	67
11.2.4. ALCUNI DATI	68
11.3. COMPARETHEMARKET.COM.....	68
11.4. UBER.....	68
11.5. COLLEGAMENTO TRA MICROSERVIZI E CLOUD	68
12. KUBERNETES	69
12.1. ORCHESTRAZIONE DEI CONTAINER	69
12.2. K8S DESIGN PRINCIPLES	69
12.2.1. DICHIARATIVITA'	69
12.2.2. DISTRIBUZIONE	70
12.2.3. DECOUPLING	70
12.2.4. INFRASTRUTTURA IMMUTABILE	70
12.3. OGGETTI	70
12.3.1. POD	71
12.3.2. DEPLOYMENT	71
12.3.3. SERVICE.....	71

<i>12.3.4. INGRESS.....</i>	72
12.4. CONTROL PLANE	72
<i>12.4.1. NODO MASTER.....</i>	<i>73</i>
<i>12.4.2. NODI WORKER</i>	<i>73</i>
12.5. LABORATORIO KUBERNETES	74
<i>12.5.1. MINIKUBE E KUBECTL.....</i>	<i>74</i>
<i>12.5.2. STEPS</i>	<i>74</i>
13. DATACENTER	76
<i>13.1. SAP DATACENTER</i>	<i>76</i>
<i>13.2. GOOGLE DATACENTER.....</i>	<i>76</i>
<i>13.3. UNIPI DATACENTER.....</i>	<i>76</i>
<i>13.4. DATACENTER INFRASTRUCTURE MANAGEMENT</i>	<i>76</i>
<i>13.4.1. COSA NON FARE ALL'INTERNO DI UN DATACENTER.....</i>	<i>76</i>
<i>13.4.2. COME REAGISCE IL PERSONALE DI FRONTE A UN PROBLEMA</i>	<i>77</i>
<i>13.4.3. BUSINESS CONTINUITY & DISASTER RECOVERY.....</i>	<i>77</i>
<i>13.5. HYPERCONVERGENCE.....</i>	<i>77</i>
14. INTERNET OF THINGS & FOG COMPUTING (DAL CLOUD ALL'IOT)	78
<i>14.1. INTERNET OF THINGS</i>	<i>78</i>
<i>14.2. DEPLOYMENT MODELS</i>	<i>78</i>
<i>14.3. FOG COMPUTING</i>	<i>78</i>
<i>14.3.1. DEPLOYING APPLICATIONS THROUGH THE FOG</i>	<i>78</i>
<i>14.4. MONITORING THE FOG</i>	<i>80</i>
15. SEMINARIO EXTRARED: HIGH TABLE	81
<i>15.1. PRESENTAZIONE EXTRARED.....</i>	<i>81</i>
<i>15.1.1. NUMERI DI EXTRARED.....</i>	<i>81</i>
<i>15.1.2. TECHNOLOGY SERVICE PROVIDER.....</i>	<i>81</i>
<i>15.1.3. LINEE DI BUSINESS</i>	<i>81</i>
<i>15.2. HIGH TABLE: IL CLIENTE DI EXTRARED E I SUOI REQUISITI.....</i>	<i>81</i>
<i>15.2.1. IL CLIENTE: L'ENTE PUBBLICO HIGH TABLE</i>	<i>81</i>
<i>15.2.2. INFRASTRUTTURA INFORMATICA DELL'ENTE.....</i>	<i>82</i>
<i>15.2.3. COSA VUOLE IL CLIENTE</i>	<i>83</i>
<i>15.2.4. COSA NON VUOLE IL CLIENTE</i>	<i>83</i>
<i>15.3. IL PROCESSO DI TRASFORMAZIONE</i>	<i>83</i>
<i>15.3.1. DIGITAL TRASFORMATION – PASSO 1.....</i>	<i>83</i>
<i>15.3.2. DIGITAL TRASFORMATION – PASSO 2.....</i>	<i>84</i>
<i>15.3.3. DIGITAL TRASFORMATION – PASSO 3.....</i>	<i>85</i>
<i>15.3.4. DIGITAL TRASFORMATION – PASSO 4</i>	<i>86</i>
<i>15.4. I TEAM E I LORO RUOLI</i>	<i>87</i>
<i>15.4.1. PLATFORM TEAM (OPS + DEVOPS)</i>	<i>87</i>
<i>15.4.2. MIDDLEWARE TEAM (MIDDLEWARE E MICROSERVIZI)</i>	<i>87</i>
<i>15.4.3. DXP TEAM (PORTALI E WEB UI)</i>	<i>87</i>
<i>15.4.4. AI&DATA TEAM.....</i>	<i>87</i>
<i>15.5. CONCLUSIONI</i>	<i>87</i>

FONTI UTILIZZATE PER LA REDAZIONE DI QUESTI APPUNTI

- Slide del docente reperibili su <https://elearning.di.unipi.it/course/view.php?id=297>
- Appunti degli studenti degli anni 2019/2020 (Appunti CGC) e 2021/2022 (Cloud_and_Green_Computing).
- AWS Identity and Access Management (IAM):
<https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html>

ALCUNE OSSERVAZIONI

- Gli appunti relativi ai laboratori potrebbero contenere solamente le linee guida su cosa si doveva fare in quella esercitazione e alcune osservazioni/conclusioni relative alla soluzione del laboratorio. Potrebbero **non** trattare i passaggi completi svolti a lezione, per i quali si rimanda al materiale didattico fornito dal professore.
- Dal prossimo anno il corso verrà diviso in due corsi separati, Cloud Computing e Green Computing. Sebbene il corso di Cloud Computing dovrebbe rimanere con pressoché gli stessi argomenti trattati in questi appunti, la parte di Green Computing potrebbe non essere sufficiente.

1. INTRODUZIONE

Tutti i discorsi che faremo in seguito si basano sul fatto che l'economia di oggi non si basa più sull'acquisto di beni, ma di **servizi**.

1.1. QUALITY OF SERVICE (QoS)

CLOUD: offre in prestito memoria, server, piattaforme, software... senza bisogno di acquistarli.

QoS: parla della disponibilità nel tempo, del tipo di assicurazione nel caso di malfunzionamenti... il costo non è l'unica cosa da guardare.

- Fondamentale per ogni servizio che utilizziamo, non basta che sia economicamente conveniente, ma deve anche essere affidabile.

CONTRATTO: ci sono scritte tutte le informazioni sull'affidabilità.

1.1.1. SERVICE LEVEL AGREEMENT (SLA)

Parte del contratto dove viene indicata l'affidabilità del servizio. Viene scritta da 3 diverse figure:

1. Avvocato specializzato nel settore.
2. Business expert.
3. Sviluppatore.

Spesso l'effettiva affidabilità non viene riportata perché è difficile stabilire a priori quanto un servizio sarà efficiente e, anche sapendo con certezza questo dato, esso dovrà competere con le altre aziende.

ESEMPI SLA:

- **Cloud Amazon S3:** non è riportato che abbiamo un qualche tipo di garanzia o indice di affidabilità → Presente solo una spiegazione su quanto "credito" l'utente può ricevere se si dovessero verificare dei disservizi.
 - Un eventuale rimborso è quasi impossibile da ottenere.
- **Cloud Microsoft:** non garantisce neanche che il cloud sia sicuro → Ci invita a fare regolarmente un backup su un'altra piattaforma.
- **Facebook:** avverte che qualsiasi contenuto condiviso pubblicamente può essere utilizzato da Facebook o essere ceduto a terzi.

1.1.2. GOOGLE COMPUTE ENGINE SPA

Permette di avere dei server virtuali.

Il Covered Service (servizio coperto) fornirà al cliente un **Monthly Uptime Percent**, ovvero la percentuale di tempo in cui il servizio è attivo, maggiore del 99.99% nel caso in cui usi istanze in più zone.

Quello appena descritto è chiamato **SLO (Service Level Objective)**. Se Google non riesce a rispettare questo SLO, e se il cliente rispetta tutti i suoi obblighi previsti nel contratto, il cliente può ricevere dei Financial Credits (rimborsi non in cash ma in termini di credito).

Monthly Uptime Percentage	Percentage of monthly bill for a Single Instance in the Region that did not meet SLO that will be credited to Customer's future monthly bills
95.00% - < 99.50%	10%
90.00% - < 95.00%	25%
< 90.00%	100%

Monthly Uptime Percentage: numero totale di minuti in un mese, meno il numero di minuti del Downtime di tutti i periodi di Downtime in un mese, diviso i minuti totali in un mese.

Downtime Period: periodo di uno o più minuti consecutivi di Downtime.

- ! Minuti parziali o periodi di Downtime intermittenti, per un periodo totale di meno di un minuto, non vengono contati.

Il cliente deve richiedere il Financial Credit per poter usufruire di quanto scritto sopra, deve richiedere supporto tecnico entro 60 giorni dal momento in cui il cliente può richiedere effettivamente un rimborso e deve **fornire a Google i file di log che mostrano errori di connettività e data e ora della comparsa di questi errori**.

1.2. RICHIESTA DI UN SERVIZIO

Varia nel tempo ed è molto difficile prevedere con precisione il numero di richieste. Questa difficoltà a prevedere le richieste può portare ai seguenti problemi:

- **Overprovisioning:** spreco di risorse → Si prevede un flusso maggiore di richieste di quelle effettive.
 - Anche nel caso di previsione corretta c'è comunque una perdita economica.
- **Underprovisioning:** errori e crash nel sistema con conseguente perdita di utenti.

Per questi motivi è nato il concetto di **Cloud**: non serve fare previsione sulle richieste perché i server sono "illimitati" e disponibili su richiesta.

1.3. CLOUD

Modello per permettere un accesso via rete diffuso, conveniente e su richiesta ad un insieme condiviso di risorse di calcolo configurabili che possono essere rapidamente fornite e rilasciate con un minimo costo di gestione o di interazione con il fornitore dei servizi.

IDEE CHIAVE:

- Raggruppare in modo efficiente e on-demand infrastruttura virtuali, offerte come servizi.
- Fornire risorse dinamicamente scalabili, virtualizzate a molti clienti attraverso Internet.
- Separare la fornitura di servizi di calcolo dalla tecnologia sottostante.

ATTRATTIVITÀ ECONOMICA:

- Eliminazione di impegni "in anticipo": non serve un investimento e progettazione iniziale per l'infrastruttura hardware.
- **PAY-PER-USE:** paghi solamente ciò che usi.

- **Vantaggi:** viene preferito anche se si spende di più, perché garantisce elasticità e permette il trasferimento dei rischi su terzi.
- **Svantaggi:** ci si lega tramite l'SLA a terze parti dalle quali potrebbe essere difficile.

MODELLO DI BUSINESS:

- Permette di trasformare alcuni costi fissi in **costi variabili**.
- Shift da spese CapEx (capitale iniziale) a spese OpEx.

1.3.1. MODELLI DI SERVIZIO E DEPLOYMENT

INFRASTRUCTURE AS A SERVICE (IAAS): fornisce server, memoria e rete (virtualizzati).

- **FORNITORE:** gestisce tutta l'infrastruttura.
- **CLIENTE:** responsabile di tutti gli altri aspetti del deployment.

PLATFORM AS A SERVICE (PAAS): fornisce un'intera piattaforma come un servizio.

- **FORNITORE:** gestisce infrastruttura + sistema operativo + enabling software.
- **CLIENTE:** responsabile di installare e gestire l'applicazione.

SOFTWARE AS A SERVICE (SAAS): fornisce software on demand accessibile mediante client thin o API.

- **FORNITORE:** gestisce infrastruttura + sistema operativo + applicazione.
- **CLIENTE:** non è responsabile di niente.

Se offri PAAS sei obbligato a offri IAAS.

ORDINE DI GESTIONE DA PARTE DELL'UTENTE: IAAS > PAAS > SAAS

MODELLO DI DEPLOYMENT (DISPIEGAMENTO):

- **Pubblico:** il vantaggio è la scalabilità, però i dati sono resi pubblici.
- **Privato:** non ha molta scalabilità, però è più controllato e sicuro.
- **Ibrido:** sono una via di mezzo.

1.3.2. OSTACOLI ALL'ADOZIONE DEL CLOUD

CONFIDENZIALITA' DEI DATI: dove vengono memorizzati concretamente? Privacy e integrità sono garantiti? Se sì, come? Come sapremo se si è verificato un errore?

DISPONIBILITA' DEI SERVIZI: cosa succede se un Cloud provider fallisce? Dobbiamo usare più provider. Ricordiamoci che **non vogliamo avere single point of failure**.

- Ci saranno dei fallimenti temporanei e il servizio potrebbe non essere disponibile in alcuni momenti.

VENDOR LOCK-IN: più vengono utilizzate funzioni del Cloud, più si rimane "incatenati" all'interno di quello specifico Cloud. Se proviamo a spostare i nostri dati dovremmo modificare tutte le nostre applicazioni per poterle far funzionare anche su altri Cloud.

- **LOCK-IN TECNOLOGICO:** non basato su un contratto.

1.4. NUOVI MODELLI DI BUSINESS

DROPBOX: memoria gratis a tutti.

SPOTIFY: musica gratuita pagando la SIAE ogni volta che viene riprodotta una canzone.

GOOGLE: motore di ricerca gratuito finanziandosi grazie al customized advertising.

1.5. GREEN COMPUTING

Fattori che contribuiscono all'inquinamento:

1. **Obsolescenza programmata**: si progettano dispositivi per fargli avere una durata breve.
2. **Obsolescenza percepita**: il venditore cerca di spingere il cliente ad acquistare un nuovo prodotto che avrà nuove funzionalità, anche se al cliente queste funzionalità non servono.

Gran numero di rifiuti tecnologici.

2. IAAS

2.1. SERVER VIRTUALIZATION E HYPERVISOR

L'IAAS presenta le infrastrutture come un servizio virtualizzato: offre la possibilità di creare macchine virtuali per gestire il carico di lavoro, rendendo la computazione più scalabile.

VIRTUALIZZAZIONE: un livello di astrazione nell'architettura di una macchina. Il sistema operativo non viene installato direttamente sull'hardware.

SERVER VIRTUALIZATION (Virtualization Layer): livello di virtualizzazione tra il server fisico e il sistema operativo. Sopra questo livello si possono installare le macchine virtuali che contengono sistema operativo e applicazione.

- **Virtual Host:** server fisico su cui girano le altre Virtual Machine.
- **Virtual Machine:** ogni S.O. ospite del Virtual Host.

HYPERVERISOR: crea il livello di virtualizzazione e si occupa della gestione delle macchine virtuali tramite il **Virtual Machine Manager (VMM)**.

TIPI DI HYPERVISOR	
TYPE 1	TYPE 2
<ul style="list-style-type: none">• Caricato direttamente sull'hardware della macchina.• Più performante.	<ul style="list-style-type: none">• Caricato sul sistema operativo dell'hardware.• Non riesce a mantenere più macchine virtuali contemporaneamente, però è più adatto per hardware meno potenti.

2.2. IAAS/COMPUTE: AMAZON ELASTIC COMPUTE CLOUD (EC2)

Mette a disposizione dei server virtuali chiamati **istanze** in modo semplice, veloce ed economico.

L'utente, tramite **Amazon Web Services (AWS) management console** o librerie SDK, sceglie il numero che vuole di istanze e le può configurare, scegliendo la potenza di calcolo.

Cinque opzioni di pagamento:

1. **On demand:** paghi solo per quello che utilizzi.
2. **Spot instances:** istanze non utilizzate che Amazon mette in vendita ogni 5 minuti all'asta.
3. **Reserved instances:** pagamento mensile, si usa quando sappiamo già quali e quante macchine mi serviranno.
4. **Dedicated hosts:** affittare un intero server fisico con istanze EC2 dedicate.
5. **AWS free tier:** livello freemium con limitazione.

VIRTUAL PRIVATE CLOUD: servizio di sicurezza che rende la connessione più sicura all'interno del Cloud.

AMAZON ELASTIC BLOCK STORE: storage persistente, ogni volume viene automaticamente replicato all'interno della sua availability zone per proteggere l'utente dai guasti, offrendo elevata disponibilità e durata.

AUTOSCALING: permette alle risorse allocate al servizio (istanze) di scalare automaticamente nel caso in cui il numero di richieste aumenti esponenzialmente.

2.3. IAAS/STORAGE: AMAZON SIMPLE STORAGE SERVER (S3)

Storage sicuro e facile da usare: memoria dove poter depositare i propri dati senza il bisogno di acquistare fisicamente le unità di archiviazione e senza preoccuparsi della sicurezza. Salvi i dati nel bucket e li riprendi quando ti servono.

INTERFACCIA: drag & drop in un bucket.

Esegue 3 backup su 3 unità di memoria diverse e consente di mantenere tutte le vecchie versioni dei file in modo da poterli recuperare se cancellati erroneamente.

CLASSI DI MEMORIZZAZIONE:

- **S3 standard**: dati a cui si accede con maggiore frequenza. Garantita bassa latenza e throughput elevato.
- **S3 standard infrequent access**: dati a cui si accede con minore frequenza, ma che richiedono un accesso rapido.
- **Amazon glacier**: dati a lungo termine ad accesso raro che non richiedono accesso immediato.

È possibile creare regole per far cambiare classe ai file automaticamente. Fornisce gestione di accesso ai dati sicura, crittografia sui dati e scalabilità automatica. Si paga per GB e in base alla classe di memorizzazione solo per quello che si usa, ma è prevista un AWS free usage tier.

2.4. DROPBOX

Servizio di file hosting che offre memorizzazione Cloud, sincronizzazione dei file e strumenti di collaborazione **gratuitamente** con limitazioni: chi vuole può acquistare l'account premium per avere più capacità di memorizzazione.

- Nel 2018 contava più di 500 milioni di utenti.

PASSAGGIO DA AMAZON S3 AI PROPRI SERVER

Hanno avuto molti problemi:

1. Dropbox progettò hardware apposito, chiamato *Diskotech*, per facilitare lo storage dei dati, e parallelamente sviluppò un nuovo software *Magic Pocket*, che non “performava” sul nuovo hardware: dovettero fare il porting dell’intero nuovo codice.
2. **Gli utenti non si sarebbero dovuti accorgere di niente**, avere interruzioni di servizio o perdite di dati.
3. Durante la costruzione dei nuovi data center vi sono stati degli incidenti stradali che coinvolgevano i camion che trasportavano i server hardware.
4. Dovevano cercare di trasferire tutto prima del rinnovo del contratto con Amazon.

PRO	CONTRO
<ul style="list-style-type: none">• Autonomia.• Costi meno elevati per fornire il servizio.• Possibilità di offrire il servizio in modo più flessibile ed efficiente.• Supporto tecnico più veloce.	<ul style="list-style-type: none">• Costi elevati sia per la realizzazione dei data center sia per la migrazione.• Rischio elevato per competenze da acquisire.• Rischio elevato per perdita di utenti durante e dopo la migrazione.

- ! Le entrate di Dropbox nel 2021 e 2022 sono destinate ad aumentare.

2.5. LABORATORIO AWS

2.5.1. COSA VIENE UTILIZZATO

STORAGE: Amazon Simple Storage Service (S3).

SICUREZZA, IDENTITA' E CONFORMITA': AWS Identity and Access Management (IAM).

- **Cos'è IAM?** Servizio Web che ti aiuta a controllare in modo sicuro l'accesso alle risorse AWS. Si può controllare chi è autenticato e autorizzato a utilizzare le risorse.
- **Feature di IAM:**
 - **Accesso condiviso al tuo account AWS:** puoi concedere ad altre persone l'autorizzazione per amministrare e utilizzare le risorse nel tuo account AWS senza dover condividere la password o la chiave di accesso.
 - **Autorizzazioni granulari:** puoi concedere autorizzazioni diverse a persone diverse per servizi diversi.
 - **Accesso sicuro alle risorse AWS per le applicazioni eseguite su Amazon EC2.**
 - **Autenticazione a più fattori (MFA):** tu o i tuoi utenti dovrete fornire non solo una password o una chiave di accesso per lavorare con il vostro account, ma anche un codice da un dispositivo appositamente configurato.
 - **Identity federation:** puoi consentire agli utenti che hanno già password altrove di ottenere l'accesso temporaneo al tuo account AWS.
 - **Informazioni sull'identità per l'assicurazione.**
 - **PCI DSS Compliance:** IAM supporta l'archiviazione, l'elaborazione e la trasmissione dei dati delle carte di credito ed è stato convalidato come conforme allo standard di sicurezza dei dati (DSS) PCI (Payment Card Industry).
 - **Integrato con molti servizi AWS.**
 - **Eventually Consistent:** IAM raggiunge un'elevata disponibilità replicando i dati su più server all'interno dei data center di Amazon in tutto il mondo. Se una richiesta di modifica di alcuni dati ha esito positivo, la modifica viene salvata e archiviata in modo sicuro. Tuttavia, la modifica deve essere replicata su IAM, operazione che può richiedere del tempo. Tali modifiche includono la creazione o l'aggiornamento di utenti, gruppi, ruoli o criteri.
 - **Free to use.**
- **Come accedere a IAM:**
 - **AWS Management Console:** la console è un'interfaccia basata su browser per gestire le risorse IAM e AWS.
 - **AWS Command Line Tools:** si possono eseguire attività IAM e AWS semplicemente dal terminale del proprio sistema operativo. AWS fornisce due tipi di strumenti a riga di comando: AWS Command Line Interface (AWS CLI) e AWS Tools per Windows PowerShell.
 - **AWS SDKs:** AWS fornisce kit di sviluppo software costituiti da librerie e codice di esempio per vari linguaggi di programmazione, oltre a piattaforme di programmazione. Forniscono un modo conveniente per creare un accesso programmatico a IAM e AWS.
 - **IAM HTTPS API:** consente di inviare richieste https direttamente dal servizio. Quando si utilizza l'API HTTPS, si deve includere il codice per firmare digitalmente le richieste utilizzando le proprie credenziali.

CALCOLO: Amazon Elastic Compute Cloud (EC2).

2.5.2. STEP 1 – FILE E IAM

1. Scaricare i file run.sh e main.py.
2. Creare un file input.txt e inserire 200 interi (positivi/negativi), uno per riga.
3. Andare sulla **Console IAM**.
4. Aggiungere utente:
 - a. Selezionare Accesso Programmatico.
 - b. Fornire la policy di accesso completo a S3 (*AmazonS3FullAccess*).

2.5.3. STEP 2 – S3 BUCKET E EC2

1. Dalla **Console S3**, creare un nuovo Bucket.
2. Caricare sul Bucket il file input.txt.
3. Dalla **Console EC2**, avviare un'istanza:
 - a. Selezionare “Red Hat Enterprise Linux 8”.
 - b. **N.B.** Durante la configurazione creare un **nuovo gruppo di sicurezza** (con impostazioni di default).
 - c. Al termine della configurazione **salvare le chiavi** (serviranno successivamente per l'accesso tramite ssh).

2.5.4. STEP 3 – COMPLETARE LO SCRIPT main.py

```
import boto3

if __name__ == '__main__':

    ACCESS_KEY = "<YOUR_ACCESS_KEY>" # si trova sul file cgcl.accessKeys.csv
    SECRET_KEY = "<YOUR_SECRET_KEY>" # si trova sul file cgcl.accessKeys.csv

    BUCKET_NAME = "cgcl32022"
    INPUT_FILENAME = "input.txt"
    OUTPUT_FILENAME = "output.txt"
    # STEP 1 - per utilizzare le SDK "Boto3", consulta la doc:
    # https://boto3.amazonaws.com/v1/documentation/api/latest/guide/s3-examples.html
    # creare il client S3
    s3_client = boto3.client("s3", aws_access_key_id=ACCESS_KEY,
                            aws_secret_access_key=SECRET_KEY)

    # STEP 2 - download "input.txt" file dal bucket creato in precedenza
    s3_client.download_file(BUCKET_NAME, INPUT_FILENAME, INPUT_FILENAME)

    # STEP 3 - ordinare gli interi all'interno del file "input.txt" e scriverli sul file
    # "output.txt"
    with open(INPUT_FILENAME, "r") as f:
        num = [int(n) for n in f.readlines()]

    with open(OUTPUT_FILENAME, "w") as f:
        for i in sorted(num):
            f.write(f"{i}\n")

    # STEP 4 - caricare il file "output.txt" sul bucket creato in precedenza
    s3_client.upload_file(OUTPUT_FILENAME, BUCKET_NAME, OUTPUT_FILENAME)

print("Done!")
```

2.5.5. COPIA FILES SU ISTANZA EC2

1. Aprire un terminale nella directory contenente il file delle chiavi dell'istanza EC2 (chiavi.pem) e digitare i seguenti comandi:
 - a. Linux: chmod 400 chiavi.pem.
 - b. Windows (da PowerShell):

```
icacls.exe chiavi.pem /reset
icacls.exe chiavi.pem /grant:r "$(env:username):(r)"
icacls.exe chiavi.pem /inheritance:r
```
2. Copiare tramite ssh su EC2 i file run.sh e main.py:

```
scp -i chiavi.pem run.sh main.py ec2-user@<IPv4_EC2_INST>:/home/ec2-user
scp -i chiavi.pem run.sh main.py ec2-user@<Public_IPv4_DNS>:~/
```

2.5.6. ESEGUIRE SCRIPT SU ISTANZA

1. Connetersi all'istanza via SSH.

```
ssh -i chiavi.pem ec2-user@<IPv4_EC2_INSTANCE>
```
2. Eseguire lo script run.sh.

```
sudo bash run.sh
```

Una volta terminati tutti gli step, controllare che il risultato presente sul file output.txt sia corretto.

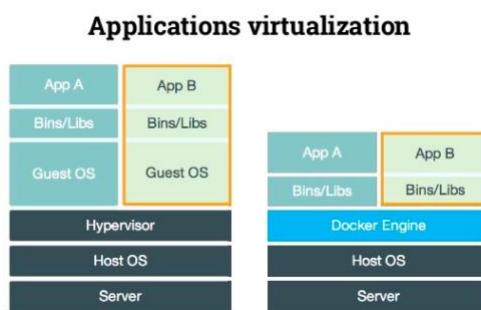
3. CONTAINERS

3.1. MACCHINE VIRTUALI VS CONTAINERS

Il nucleo dei sistemi operativi permette di avere più istanze isolate, che vengono chiamate Container. I Container sono quindi ambienti software in grado di eseguire e isolare dall'esterno l'esecuzione di processi e applicazioni.

DOCKER: piattaforma che permette di eseguire applicazioni in un ambiente di sviluppo isolato. Le applicazioni vengono quindi trasportate tramite questi Container e, una volta trasportata, eseguita.

Al posto dell'*hypervisor* viene montato il cosiddetto **docker engine**, che permette di creare i Container che contengono le applicazioni e le sue dipendenze.



Sia nelle macchine virtuali che nei Container vi è un livello di virtualizzazione; tuttavia, nei Container manca il sistema operativo guest.

VANTAGGI (Rispetto ad una macchina virtuale)

- Più **leggieri**: un Container chiede ad un server meno risorse.
- I tempi di avvio sono più **brevi**.
- Più **semplici** da costruire.

SVANTAGGI (Rispetto ad una macchina virtuale): condividono più risorse all'interno di un server fisico: c'è meno isolamento, quindi una superficie più ampia per gli attacchi di sicurezza. Un Container è quindi **meno sicuro**.

3.2. UN PO' DI STORIA

I Container esistono da una decina di anni...

- 1998 – l'utility jail di FreeBSD ha esteso il sandboxing (esecuzione di applicazioni in spazi limitati) ai processi tramite chroot.
- 2005 – Google ha iniziato a sviluppare CGroups per i kernel Linux e ha iniziato a spostare le sue infrastrutture nei Container.
- 2008 – i Container Linux (LXC) forniscono una soluzione basata su Container completa.

...ma con Docker la diffusione è aumentata perché ha aggiunto:

1. Portabilità delle *immagini*.
2. Interfaccia utente *friendly*.

Docker offre:

- **Docker engine:** creazione ed esecuzione di Container.
- **Docker hub:** pubblicare e scaricare immagini di Container.

3.3. DOCKER

Sfrutta la virtualizzazione basata sui Container per mandare in esecuzione più istanze utente isolate sullo stesso sistema operativo.

I Container sono **volatili**: non tengono traccia dei dati una volta terminata l'esecuzione.

VOLUMI: meccanismo per memorizzare i dati in modo persistente.

Come funzionano i Container?

- L'applicazione viene impacchettata in un'**immagine**, un file che rappresenta l'intera applicazione.
- Questa diventa un template che poi verrà usato per creare e mandare in esecuzione un Container.
- I template vengono memorizzati in un **registry** (Docker Registry) che è strutturato in repositories. Ogni repository contiene un insieme di immagini per versioni diverse del 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 può effettuare delle modifiche che possono a loro volta essere *committate* in una nuova immagine.

COMANDI PRINCIPALI	
pull	Scarica un'immagine da un Docker Registry.
run	Manda in esecuzione il Container definito dall'immagine.
commit	Creare ed effettuare delle modifiche che possono essere committate in una nuova immagine.
build	Specificare la costruzione dell'immagine in un file chiamato dockerfile . <ul style="list-style-type: none"> • Nel dockerfile scriviamo una serie di istruzioni che ci permettono di creare un'immagine automaticamente.

3.3.1. PICCOLA DEMO DIMOSTRATIVA

Quello che vogliamo fare è creare un'applicazione che stampi "Hello Word" sfruttando Docker.

Dopo aver installato Docker, scriviamo quindi la nostra applicazione php in un file che chiameremo index.php:

```
<?php
echo "Hello, World"
```

Salviamo questo file in una cartella **src**. Andiamo quindi a creare il nostro dockerfile nello stesso folder di **src** (e non al suo interno). Il bello di Docker è che non dobbiamo creare un'immagine da zero, ma possiamo scaricare un'immagine già pronta e poi costruire il nostro dockerfile sopra di essa.

- ! Alcune immagini sono ufficiali, altre no, quindi attenzione a quale immagine si scarica.

```
FROM php:7.0-apache
COPY src/ /var/www/html
EXPOSE 80
```

Quando abbiamo più Container in esecuzione sulla stessa macchina, ci sono tante porte, e ogni Container può richiedere di usare porte diverse). Con `EXPOSE 80` diciamo che il Container in questione ascolterà sulla porta 80.

Quando andiamo ad eseguire il comando `run`, dobbiamo “inoltrare” la porta 80 dell’host nella porta 80 del Container. In questo modo, quando arriva una richiesta da parte dell’host, Docker la inoltrerà al Container, e quando arriverà al Container la riga `EXPOSE 80` consentirà al Container di accettare la richiesta.

! In generale i numeri di porta possono essere diversi.

```
$ docker run -p 80:80 hello-world
```

Se ora vogliamo fare una modifica al file `index.php`, il server `localhost` non mostrerà i cambiamenti, perché non avrà modo di vedere le modifiche. Queste modifiche verranno infatti messe in delle nuove copie (che avranno i dati aggiornati) e salvate nel Container. Per dare la possibilità a `localhost` di vedere questi cambiamenti, dobbiamo montare un volume condiviso tra Container e server `localhost`.

```
$ docker run -p 80:80 hello-world -v  
/Users/username/Desktop/docker/src/:/var/www/html/ hello-world
```

Per poter distribuire l’applicazione e farla eseguire da qualche altra parte, dobbiamo comunque ricostruire l’immagine.

3.3.2. DOCKER COMPOSE

Permette di specificare come si vuole che sia formata l’applicazione multiservizio da mandare in esecuzione. Ogni Container conterrà un microservizio.

Con Docker Compose possiamo quindi scrivere i nostri microservizi separatamente per poi ricomporli insieme tramite il file Docker Compose, nel quale scriviamo quanti servizi deve avere la nostra applicazione. Docker Compose crea automaticamente una rete virtuale tra questi servizi. Ogni microservizio può quindi richiamare gli altri basandosi sui nomi definiti dentro il file Docker Compose.

```
--File docker-compose.yml--  
  
version: '3'  
services:  
    product-service:  
        build: ./product  
        volumes:  
            - ./product:/usr/src/app  
        ports:  
            - 5001:80 /* Associamo la porta 80 del Container dedicato al  
                    product service con la porta 5001 del server */  
  
    website:  
        image: php:apache  
        volumes:  
            - ./website:/var/www/html  
        ports:  
            - 5000:80 /* Associamo la porta 80 del Container dedicato al  
                    website con la porta 5000 del server */  
        depends_on:  
            - product-service
```

3.3.3. DOCKER SWARM

Swarm mode: modalità che permette di gestire un insieme (*cluster*) di host Docker chiamati swarm.

- Quando dobbiamo mandare in esecuzione un Container, la swarm mode lo manda a uno dei vari host: possiamo quindi fare una distribuzione del carico su un insieme di docker host.

I vari task vengono ripartiti tra host diversi. I nodi del cluster possono avere due ruoli:

1. **Manager:** delegano e distribuiscono i task.
2. **Workers:** eseguono i task.

L'utente può definire lo stato per la configurazione a tempo di esecuzione dell'applicazione. Il manager swarm effettua un monitoraggio costante delle attività e quando vi è una differenza tra lo stato attuale e lo stato desiderato dall'utente, interviene.

3.4. LABORATORIO DOCKER

Utilizzare Docker per eseguire una app su un container.

1. Rispondere nel file *report* ad alcune domande sui comandi principali di Docker.
2. Scrivere il Dockerfile per eseguire l'applicazione *simpleapp*.
 - a. Utilizzare come immagine di partenza `python:3.8-slim-buster`.
3. Provare l'applicazione.

--File report--

- **Mostrare l'output del comando `docker run hello-world`.**
- **Se si riesegue il comando precedente, la sua esecuzione è più veloce? Se sì, perché?**
Sì, perché docker riesce a trovare l'immagine in locale. Infatti, durante la seconda esecuzione non verrà stampata a video la scritta "Unable to find image 'hello-world:lates' locally".
- **Spiega cosa fa il seguente comando docker, descrivendo TUTTE le sue componenti:**
`docker run -i -t debian /bin/bash`.
docker – lancia docker.
run – prima crea un nuovo livello scrivibile del container sopra l'immagine specificata e poi manda in esecuzione il container con un'istanza dell'immagine "debian".
-t – permette di allocare una sessione virtuale del terminale all'interno del Container.
-i – specifica la modalità interattiva.
`/bin/bash` – manda in esecuzione una bash all'interno del Container su cui gira l'immagine debian.
- **Dopo aver eseguito il comando precedente, cosa succede eseguendo il comando `exit`? Perché?**
Il comando exit permette di eseguire il logout dalla shell collegata al Container creato in precedenza. Nel caso in questione, con il comando `exit` viene terminato anche il Container, perché `/bin/bash` è l'unico eseguibile in esecuzione sul Container.
- **Mostrare l'output del comando che permette di ottenere la lista delle immagini al momento presenti sulla propria macchina.**
Il comando è `docker images`.
- **Mostrare l'output del comando che permette di ottenere la lista dei container in esecuzione.**
Il comando è `docker ps -a` – mostra tutti i container eseguiti e in esecuzione.

- **Scrivere la lista di TUTTI i comandi utilizzati per poter eseguire l'applicazione “simpleapp” con Docker, e testarne il funzionamento in locale.**

Creazione dockerfile:

```
FROM python:3.8-slim-buster
ADD ./mycode
WORKDIR /mycode
RUN pip install -r requirements.txt
EXPOSE 5000
CMD ["python3", "-m", "flask", "run", "--host=0.0.0.0"]
```

Costruiamo un'immagine Docker a partire dal Dockerfile e da un contesto, ovvero l'insieme di files specificati nel PATH o URL passato come parametro.

```
docker build -t simpleapp
```

Il processo di build può riferirsi ad ogni file presente nel contesto. Il flag `-t` è diverso da quello che utilizziamo nella run, qui ha il significato di `--tag`, ovvero taggare, nominare l'immagine che sto andando a creare.

```
docker run -p 5000:5000 simpleapp
```

Il flag `-p` (`--publish`) serve a “pubblicare” una porta del container all’host, ovvero a renderla visibile ed usabile nei confronti dell’host. In questo caso al porta dell’host e del container sono la stessa, la numero 5000. A sinistra del carattere “`:`” è specificata la porta dell’host, a destra quella del container.

4. PAAS

4.1. DEFINIZIONI E VANTAGGI

I PAAS ci offrono un intero ambiente di sviluppo e gestione dell'applicazione.

DEFINIZIONE 1 – Capacità che viene fornita agli utenti per mandare in esecuzione applicazioni sia create dall'utente, sia di terze parti che utilizzano linguaggi di programmazione, librerie, servizi, strumenti supportati dal provider, senza preoccuparsi del gestire l'infrastruttura del Cloud sottostante.

- Si può mandare in esecuzione applicazioni su una piattaforma virtualizzata.

DEFINIZIONE 2 – Modello di Cloud Computing in cui un provider di terze parti fornisce strumenti hardware e software che vengono utilizzati per il deployment e per lo sviluppo.

DEFINIZIONE 3 – CATEGORIA DI SERVIZI Cloud Computing che permette di fornire, istanziare, mandare in esecuzione e gestire un *bundle* che comprende sia una piattaforma di calcolo sia uno o più applicazioni.

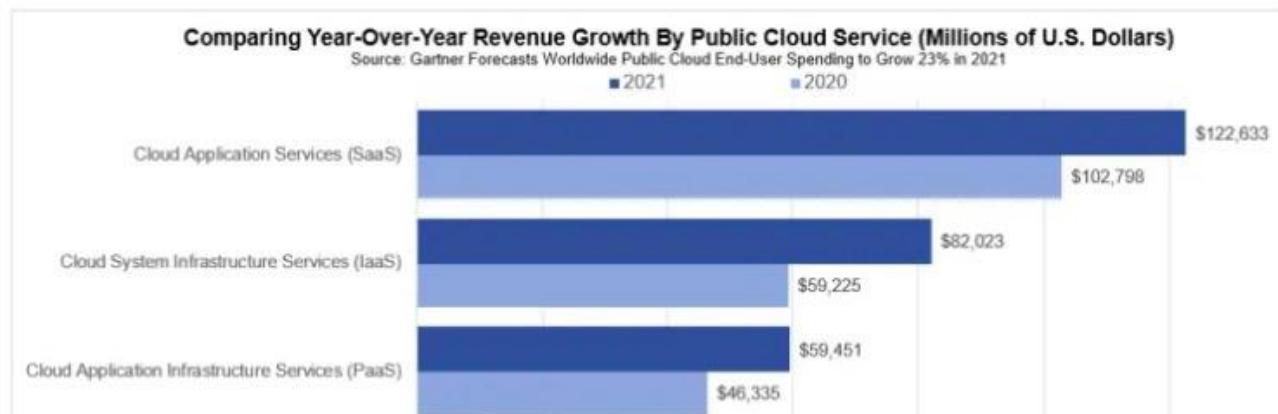
L'idea di base è di avere una piattaforma con cui gestire le applicazioni. Gli operatori dell'applicazione non devono più preoccuparsi della parte di calcolo, storage e networking ma possono concentrarsi solamente sulla parte dell'applicazione.

VANTAGGI

1. Creare prototipi in pochi minuti.
2. Creare nuove versioni o deploy del codice più rapidamente.
3. Assemblare in modo semplice le varie parti dell'applicazione.
4. Scala automaticamente.
5. Non bisogna preoccuparsi della sicurezza.

SVANTAGGI

1. Se c'è un blackout del mio fornitore PaaS, non posso più utilizzare i servizi già nell'immediato.
2. Vendor lock-in.
3. Cambiamenti interni al PaaS.



4.2. HEROKU

Fornisce una serie di servizi integrati e un intero sistema che permette:

1. Di fare deployment delle applicazioni.
2. Di completarle, mandarle in esecuzione e gestirle.

CONTAINER – Ambiente software isolato in grado di eseguire e isolare dall'esterno l'esecuzione di processi e applicazioni.

- È leggero e facilmente modificabile.
- Risolvono il problema della **portabilità**: una volta giunto a destinazione, viene prelevata l'immagine del Container e ricostruita l'applicazione.
- È un modo per virtualizzare la rappresentazione di un'intera applicazione con tutte le sue dipendenze.

DYNOS – Container utilizzati da Heroku: eseguono codice dato dall'utente.

- Permettono una scalabilità molto flessibile a seconda delle richieste e delle risorse.
- Si può gestire il tipo, numero e dimensione di Dynos per applicazione.

L'utente non si deve più preoccupare della gestione dell'infrastruttura e della scalabilità dell'applicazione.

FUNZIONAMENTO

1. L'applicazione riceve la richiesta che viene inviata a uno dei **Web Dynos**.
2. La richiesta viene analizzata e messa in coda asincrona (ottima per la scalabilità orizzontale).
3. Il **Worker Dynos** prende la richiesta e la soddisfa: se necessario, può memorizzare in maniera persistente il risultato in un database.

La scalabilità permette di aumentare il numero di Web Dynos per gestire un elevato numero di richieste ricevute contemporaneamente.

4.2.1. FASI HEROKU

1 – BUILDTIME

Servono 3 cose per costruire l'applicazione:

1. Codice sorgente.
2. Lista di dipendenze.
3. Un *procfile*: file di testo che indica qual è il comando per far partire il codice.

Date in input queste tre cose, parte il **sistema automatico di built**: dopo aver ricevuto il codice, scarica il **Build Packet**, produce uno **slug** (insieme di codice sorgente, dipendenze, output...) e lo mette in esecuzione in un Dynos.

Il componente finale per eseguire l'applicazione è il sistema operativo, che è aggiunto da Heroku e si chiama **stack**.

2 – RUNTIME

1. Quando si fa il deployment o si scala manualmente l'applicazione, Heroku crea uno o più Dynos dove ognuno avrà lo stesso stack e slug che rappresenta l'applicazione.
2. Heroku esegue il comando che l'utente ha specificato e fa partire l'applicazione.

Heroku permette di configurare le risorse che a tempo di esecuzione si vogliono utilizzare. Esistono quattro principali tipi di Dynos:

1. *Free, hobby* – ci sono le funzionalità di base.
2. *Standard* – supportano la scalabilità orizzontale.
3. *Performance* – c'è l'autoscaling in cui si stabilisce a priori quali sono i parametri secondo cui si vuole che Heroku scali.

3 – ADDS-ONS

Con questi, gli sviluppatori possono estendere l'applicazione.

- **Vantaggi:** si possono integrare nuove funzionalità in un tempo molto breve e facilitano lo sviluppo (non dobbiamo costruire da zero ciò che vogliamo).
- **Svantaggi:** ci legano all'utilizzo della piattaforma e instaurano una forma di *vendor lock-in* rendendo la nostra applicazione più difficilmente portabile.

Per aggiungere un add-on ad un'applicazione Heroku, basta usare il comando

```
heroku addons:create ADD-ON --app example-app
```

4.2.2. TIPI DI DYNOS

Dyno Type	Memory (RAM)	CPU Share	Compute	Dedicated	Sleeps
free	512 MB	1x	1x-4x	no	<u>yes</u>
hobby	512 MB	1x	1x-4x	no	no
standard-1x	512 MB	1x	1x-4x	no	no
standard-2x	1024 MB	2x	4x-8x	no	no
performance-m	2.5 GB	100%	12x	yes	no
performance-l	14 GB	100%	50x	yes	no

- I tipi standard supportano la **scalabilità**.
- I tipi performance supportano la **scalabilità e l'autoscaling**.

4.3. MICROSOFT AZURE

Servizi offerti:

- Scalabilità verticale.
- Supporto di diversi linguaggi e ampia varietà di strumenti per gli sviluppatori.
- SQL database.
- Machine learning per analisi predittive.
- Servizi media.
- Meccanismi di sicurezza.
- Data storage.

È un PAAS ma offre anche servizi IAAS.

4.4. OPENSHIFT

RedHat OpenShift:

- Offre funzionalità semplici per il build e il deployment automatico.
- Può monitorare lo stato delle applicazioni e farle ripartire in caso di fallimenti.
- È basato su *kubernetes*.
- È una piattaforma DevOps che permette la costruzione e l'erogazione dei servizi: un'organizzazione può costruire il proprio workflow DevOps in modo da permettere ai teams di collaborare mantenendo autonomia.

4.5. CLOUD FOUNDRY

È un Open Source. È un insieme di servizi distribuiti quali:

- **Cloud Foundry Bosh:** permette di utilizzare tutta la struttura sottostante, ovvero i servizi IAAS offerti da provider diversi.
- **Healt Manager:** monitora la qualità del servizio per capire come sta crescendo e funzionando la nostra applicazione, riportando eventuali discrepanze al controllore.
- **Dynamic Router:** instrada tutto il traffico che arriva dalla rete esterna.
- **Cloud Controller:** mantiene e gestisce tutti i sistemi di controllo della piattaforma, anche l'interfaccia con i clienti.
- **User Authorization and Authentication:** gestisce l'autenticazione e l'autorizzazione.
- **Servire Brokers:** permette di accedere ad altri servizi esterni o nativi.

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

4.5.1. CASE STUDY: CSAA INSURANCE

PROBLEMI:

- Situazione poco stabile.
- Non avevano automazione.
- “Debito tecnico”: molte tecnologie obsolete.
- Troppi “silos”: blocchi di grandi applicazioni che invece sarebbero dovute stare separate.
- Poca capacità di fare analisi sui dati.

SOLUZIONI: riportiamo i vari passaggi nel tempo.

- **Pivotal Cloud Foundry:** funzionò e lo usarono nella produzione vera e propria.
- **Cloud pubblico:** passaggi in pochi giorni che avrebbero invece richiesto diversi mesi, ottenendo un'automazione robusta.
- **Cloud privato.**

RISULTATI:

- Aumento del 200% di produttività.
- Aumento di deployment frequency del 1400%.

4.5.2. CASE STUDY: AUSTRALIAN GOVERNMENT

PROBLEMA: gestire più di 1500 siti web di agenzie locali.

OBIETTIVI:

- Diminuire la burocrazia.

- Riuscire a compattare e coordinare questi servizi sparsi.
- Ridurre il tempo di deployment delle applicazioni.
- Ridurre il downtime a zero.

SOLUZIONI:

- Creato un'agenzia digitale per semplificare l'interazione con gli utenti per i servizi governativi.
- Progettato il passaggio su **Cloud Foundry** grazie ad un team cross-functional.
- Iniziare a **modificare i servizi più utilizzati** anziché iniziare da quelli meno utilizzati.

RISULTATI:

- Per le prenotazioni di appuntamenti da un'attesa di 92 minuti a 2 minuti.
- Semplificata la richiesta per i permessi di soggiorno.
- Digital Marketplace per appalti e gare.
- Possibilità di avere una dashboard.

4.6. TROVARE IL PaaS PIÙ ADATTO

Ci sono degli strumenti come **PaaS Finder** che servono a trovare il PaaS più adatto a te, semplicemente specificando le tue esigenze.

5. VENDOR LOCK-IN

Rende un cliente dipendente da un prodotto o un servizio: il cliente è impossibilitato ad usare un provider diverso senza sostanziali costi relativi al cambio.

5.1. VENDOR LOCK-IN NEL MONDO DEL CLOUD

DATA GRAVITY: più è grande la quantità dei dati, più è difficile spostarli.

- Lambda è una delle peggiori forme di lock-in.

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

LOCKOUT (Altro grande problema da evitare)

A differenza dei tradizionali software e hardware *in-house*, **se un provider Cloud smette di erogare il suo servizio, il cliente perde i suoi dati**.

LIVELLI DI LOCK-IN: *FaaS > PaaS > IaaS*

5.2. ESEMPI DI LOCK-IN

PLATFORM LOCK-IN

- Bloccati dalle librerie proprietarie e dalle configurazioni di sistema.
- Provider diversi potrebbero implementare le stesse funzionalità in modi diversi o non implementare affatto le stesse funzionalità.

ALTRI ESEMPI

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

5.3. PERCHE' VORREI CAMBIARE PROVIDER?

1. Il provider fallisce.
2. Aumento dei costi del provider.
3. Diminuisce la qualità del servizio.
4. Possono cambiare i termini di servizio.
5. Troppi **outages (interruzioni)**.
6. Non supporta alcune features che vorrei.

5.4. CONSIGLI PER EVITARE/RIDURRE IL LOCK-IN

IN GENERALE.

Pianifica la tua uscita: prima di iniziare un nuovo contratto con un Cloud, assicurati di avere una strategia di uscita → Niente rimane stabile nel mondo dell'IT.

Gestisci i tuoi dati: assicurati che il cloud sia data-centrico. La portabilità dei tuoi dati è più importante della portabilità dell'applicazione.

Garantisce agilità ma pensa bene all'utilizzo di servizi proprietari dei cloud-vendor: è importante che sia garantita agilità, ma seleziona bene le funzionalità dei vari cloud, non fermarti a quelli che propongono funzionalità già pronte e facili da usare.

NELLO SPECIFICO.

IaaS:

- Usa Docker o soluzioni simili.
- Evita integrazioni dirette con il database.

IaaS/PaaS:

- Implementa integrazioni API/REST.
- Usa API comuni.

SaaS:

- Assicurati che ci sia un metodo standard per l'export dei dati.
- Testa il metodo di export dei dati.
- Favorisci soluzioni con REST API stabili e ben conosciute.

Per ogni Cloud:

- Favorisci tecnologie Open Source.
- Evita la dipendenza dalle tecnologie di un unico Cloud vendor.
- Usa i messaggi quando possibile.
- Usa due Cloud.

5.5. STANDARD CLOUD

Sono standard pubblici e dicono come i Cloud devono essere.

CDMI – Cloud Data Management Interface: definisce come deve essere un'interfaccia REST per gestire i dati.

OCCI – Open Cloud Computing Interface: protocolli REST e API per gestire IaaS.

CIMI – Cloud Infrastructure Management Interface: standardizza il modo in cui fornitori e consumatori di un servizio possono interagire con l'IaaS.

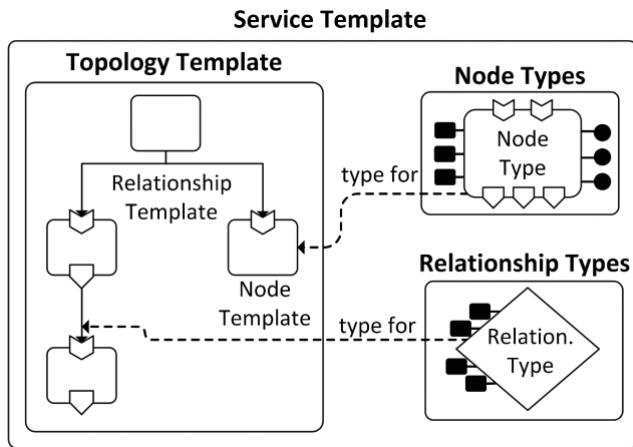
OVF – Open Virtualization Format: formato aperto, sicuro, efficiente ed estendibile per il packaging e la distribuzione del software da eseguire nei sistemi virtuali.

CAMP – Cloud Application Management for Platforms: standard di come deve essere fatta una API di un gestore PaaS in modo che sia garantita la portabilità.

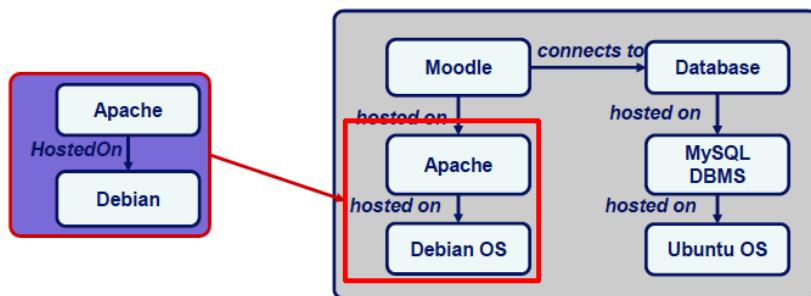
TOSCA – Topology & Orchestration Specification for Cloud Applications: linguaggio (basato su YAML) per descrivere le applicazioni cloud come grafi di relazioni e componenti tipizzate per abilitare la creazione di applicazioni cloud portabili e l'automazione del loro deployment e management.

5.5.1. OBIETTIVI TOSCA

PORTABILITÀ: TOSCA fornisce un linguaggio basato su YAML per descrivere un'applicazione Cloud come un grafo di componenti e relazioni tipizzate.



RIUSABILITÀ' DEI COMPONENTI: TOSCA permette la definizione, l'assemblamento e l'impacchettamento dei blocchi di costruzione autonomi.



AUTOMATIZZARE IL DEPLOYMENT: ogni nodo TOSCA ha un set di operazioni standardizzate.

```
create
configure
start
stop
delete
```

Ognuna di queste operazioni può essere implementata semplicemente allegandola ad esempio in uno script eseguibile, che implementa il suo deployment e il suo management.

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

Deployment dichiarativo delle applicazioni TOSCA: fare il deploy di un nodo con tutti i requisiti soddisfatti, eseguendo lo script associato alle operazioni `create`, `configure` e `start`.

6. BUSINESS MODEL

6.1. BUSINESS MODEL CANVAS

BUSINESS MODEL: descrive come un'azienda crea, consegna e cattura valore.

BUSINESS MODEL CANVAS: rappresentazione grafica in cui un modello di business viene rappresentato.



- **Value proposition:** l'aspetto che distingue il servizio che stiamo offrendo.
- **Customer relationship:** rapporto con il cliente.
- **Customer segment:** target di mercato e tipi di clienti.
- **Channels:** canali con cui il servizio arriva ai clienti.
- **Key partners:** partner chiave.
- **Key activities:** attività principali della nostra azienda.
- **Key resources:** risorse chiave.
- **Cost structures:** le uscite della nostra azienda.
- **Revenue streams:** i flussi da cui arrivano le entrate e il loro tipo.

6.1.1. ESEMPI DI BUSINESS MODEL CANVAS

NESPRESSO

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 cambiò business model, in particolare il *customer segment*: le macchinette dovevano attrarre impiegati di alto livello e in generale famiglie benestanti.

I canali di acquisto sono due:

1. Shop online.
2. Boutique, solitamente nel centro della città.

Club Nespresso: offre dei vantaggi ai clienti, permette di tracciare le preferenze e l'attività dei clienti.

Critiche:

1. Sostenibilità ambientale: grandissima quantità di materiale difficilmente riciclabile.
2. Commercializzazione del latte in polvere: nei paesi in via di sviluppo l'acqua con cui veniva sciolto il latte era contaminata e molti bambini sono morti.

KEY PARTNERS -Machine manufacturers -Raw material suppliers	KEY ACTIVITIES -Coffee procurement -Marketing -Selling -Post purchase	VALUE PROPOSITIONS -High quality coffee -Post purchase service -Innovative product -Make customer special -Coffee maker design -Recognition	CUSTOMER RELATIONSHIPS -Nespresso club -Personal assistance	CUSTOMER SEGMENTS -Elite (high class) -Niche market -Social status -People who want one coffee at a time
KEY RESOURCES -Coffee beans -Coffee boutiques -Workers in shops	CHANNELS -Online shops -Boutiques			
COST STRUCTURE -Manufacturing -Distributing -Selling		REVENUE STREAMS -Big revenue on capsules		

ZARA

Produce abiti in base alla preferenza dei clienti, con un monitoraggio costante dei consumi degli utenti: vi è un rinnovo costante della linea dei prodotti.

Fornisce settimanalmente quello di cui ha bisogno un certo punto vendita: nessun punto vendita utilizza un magazzino dove avere le scorte dei prodotti (no warehousing).

I prezzi sono più convenienti rispetto ai negozi di alta moda, ma i punti vendita sono collocati vicini ad essi.

TICKET RESTAURANT

Le aziende che non possono ospitare un servizio mensa, sottoscrivono dei contratti con Ticket Restaurant, che a sua volta si mette d'accordo con dei ristoranti/bar.

Le aziende forniscono ai dipendenti dei buoni pasto che possono utilizzare per andare nei ristoranti partner di Ticket Restaurant.

Ticket Restaurant si prende una quota sia dalle aziende che dai ristoranti. I buoni hanno una scadenza: se scadono, Ticket Restaurant risparmia qualche euro.

6.2. FREEMIUM BUSINESS MODEL

Suddivisione dei servizi:

1. Parte **Free**: comprende i servizi base.
2. Parte **Premium**: offre servizi aggiuntivi e/o migliorati.

Si deve porre molta attenzione al costo che comportano gli utenti free e alla percentuale di conversione degli utenti che decidono di passare a premium.

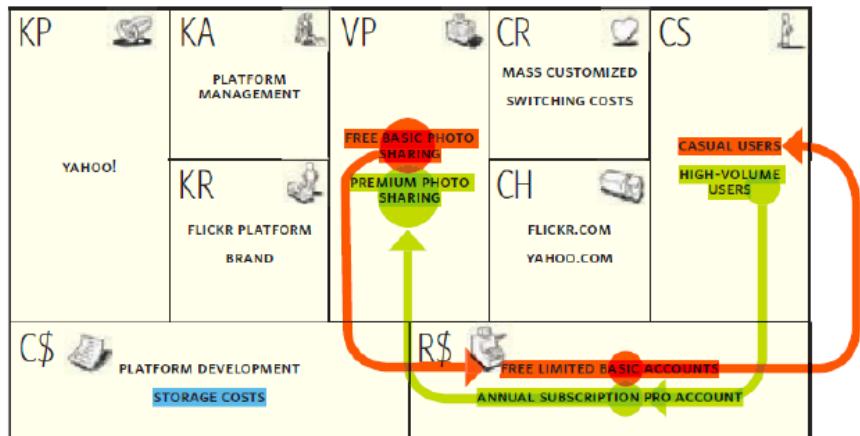
6.2.1. FLICKR

Consente di caricare e condividere immagini.

- **Utenti free**: spazio di memorizzazione e funzioni limitate.
- **Utenti premium**: tutte le funzionalità del servizio.

Uno degli obiettivi delle startup è quello di riuscire a farsi acquisire da qualche azienda più grande:

- Nato nel 2002, Flickr viene acquistato nel 2005 da Yahoo! per 25 milioni di dollari, acquistato a sua volta nel 2017 da Verizon per 4 miliardi.
- Nel 2018 Flickr diventa parte di SmugMug.

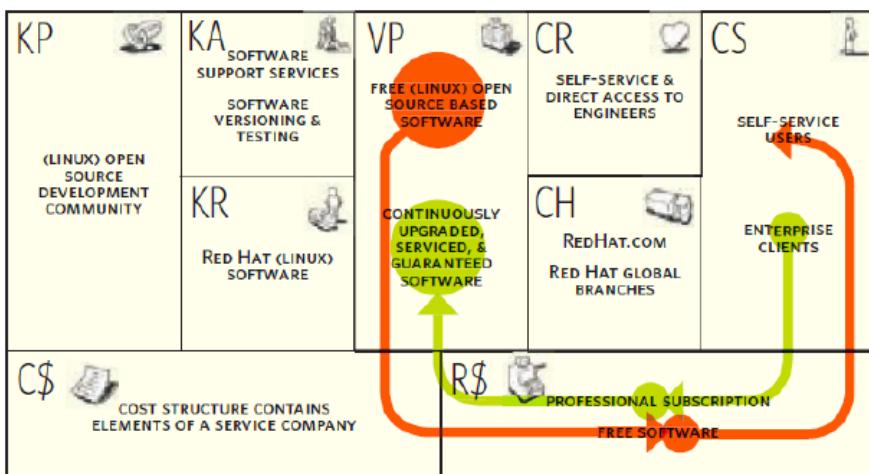


6.2.2. REDHAT

Le aziende che volevano usare un software open source avevano il timore che esso potesse fallire o che nessuno poteva offrire assistenza in caso di problemi.

- **Utenti free:** software gratuiti open source.
- **Utenti premium:** aggiornamento dei software, sicurezza, manutenzione.

Acquisita per 34 miliardi da IBM nel 2019.

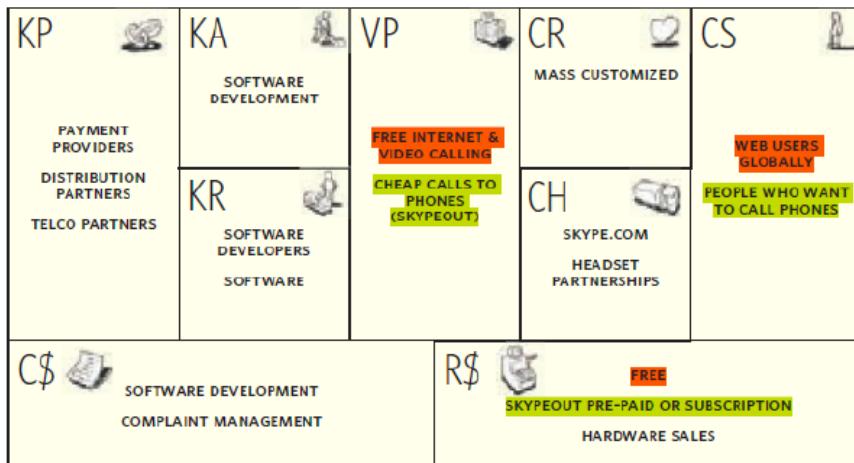


6.2.3. SKYPE

Chiamate e videochiamate gratuite per tutti.

- Pagando, si ha la possibilità di chiamare anche i cellulari a prezzi competitivi.

Comprato nel 2005 da Ebay e nel 2011 da Microsoft.



6.2.4. DROPBOX

	For individuals		For teams		
	Professional €16.58/month	Professional + eSign €25.99/month	Standard €10/user/month	Standard + DocSend €45/user/month	Advanced €15/user/month
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		✓			

Ha 500 milioni di utenti di cui 12 milioni paganti.

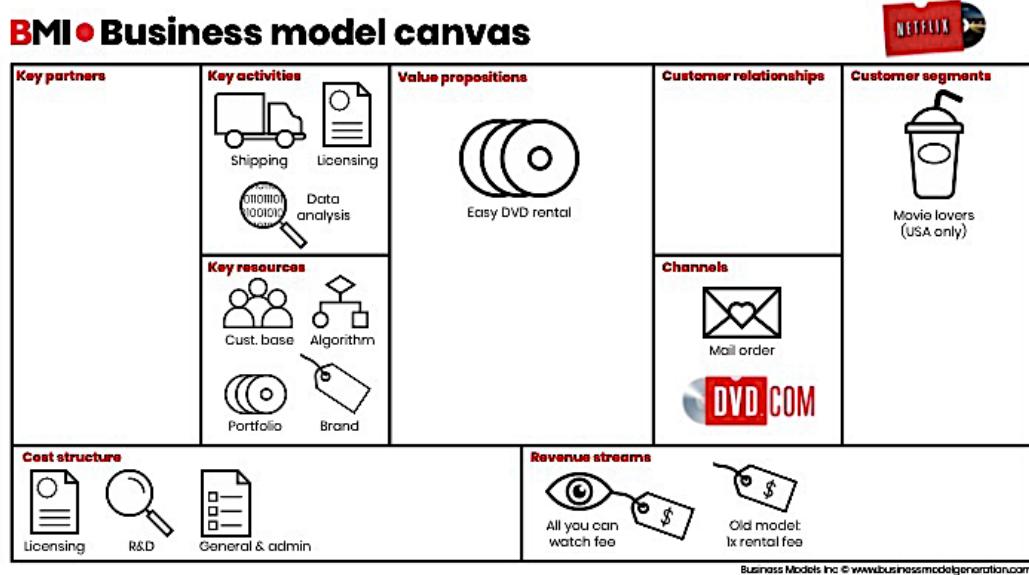
6.2.5. NETFLIX

! Usa attualmente il 15% del traffico Internet in download nel mondo.

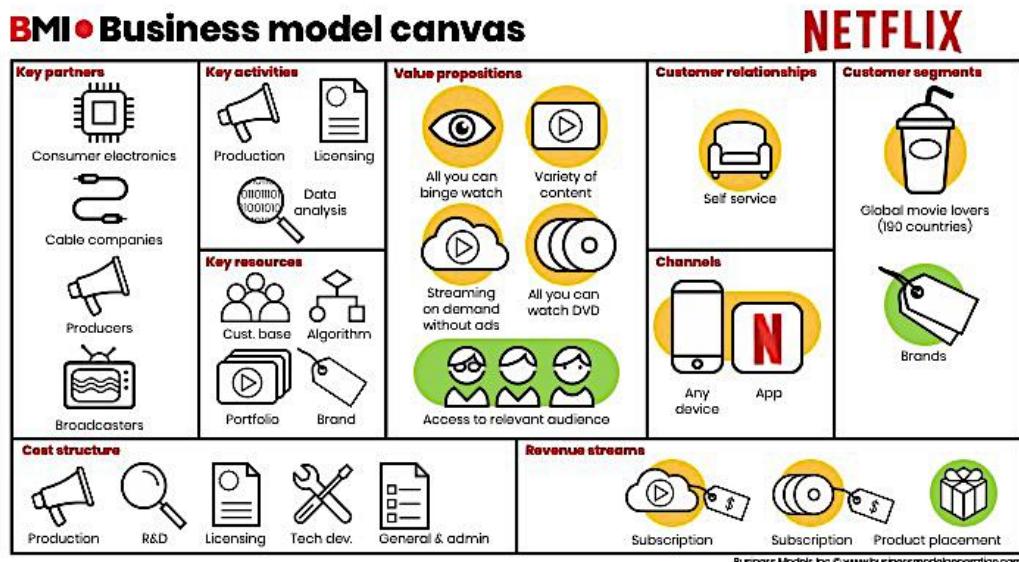
Paga i diritti dei film e guadagna per il “noleggio” verso gli utenti.

Strategia opposta rispetto a Dropbox: nonostante avesse già costruito i data center, ha chiuso la propria rete, affidandosi completamente a Amazon AWS

Inizialmente il business model era...



...ora invece è



6.2.6. SPOTIFY

Per pagare meno di royalty ha ideato una strategia che permette di scegliere dove reperire il brano in base alla localizzazione dell'utente.

! 217 milioni di utenti attivi di cui 100 milioni premium.

Stakeholders:

- **Free users:** utenti free.
- **Royalty holder:** vengono pagati ogni volta che viene riprodotto un brano.
- **Subscriber:** utenti premium.
- **Advertiser:** pagano Spotify per venire sponsorizzati.

Supponendo di dover pagare 0,004 USD per play (per i copyright): quanto dovremmo ricavare complessivamente dalla pubblicità per non andare in rosso nei primi e nei secondi sei mesi di attività?

	GS1	GD1	GD2	GS2
num. utenti free	450.000	9.000	497.000	9.500
num. utenti premium	50.000	1.000	3.000	500
num. track di 1 utente free in 1 giorno	20	15	20	20
num. track di 1 utente premium in 1 giorno	20	15	20	20
costo infrastruttura	€ 732.000		€ 0	€ 21.000
costo personale	€ 180.000	€ 180.000	€ 0	€ 30.000
fee utenti premium (mensile)	€ 7,99	€ 8,00	€ 7,99	€ 6,00
ENTRATE	€ 2.397.000	€ 48.000	€ 143.820	€ 18.000
USCITE	€ 8.112.000	€ 288.000	€ 7.200.000	€ 195.000
differenza	-€ 5.715.000	-€ 240.000	-€ 7.056.180	-€ 177.000
		(-118.000 con crescita)		

6.3. BUSINESS MODEL GENERATION

La cosa più importante è mettere al centro il cliente e il rapporto che si crea con esso.

I modelli di business più rivoluzionari sono quelli che hanno ignorato lo *status quo*, ovvero hanno rotto con il passato: questo lo troviamo in molte domande **what if**.

Nell'innovazione del business ci sono vari possibili epicentri:

1. **Resource Driven:** partire dalle risorse che ha già l'azienda.
 - a. **Amazon Fullfilment:** offre a terze parti la possibilità di vendere e far arrivare i propri beni ai clienti.
2. **Offer Driven:** basarsi sull'offerta.
 - a. **Cemex:** in un momento in cui il cemento era garantito in 48 ore, è riuscita a farlo in 4 ore.
3. **Customer Driven:** basarsi sui clienti.
 - a. **23andMe:** test per DNA ai singoli clienti privati.
4. **Finance Driven:** basarsi sull'aspetto finanziario.
 - a. **Xerox:** introdusse un modello in cui il costo delle fotocopie veniva offerto in leasing e venivano inoltre garantite 2000 copie gratuite.
5. **Multiple Epicenter Driven:** epicentri multipli.

6.3.1. STATISTICHE DI ALCUNE STARTUP

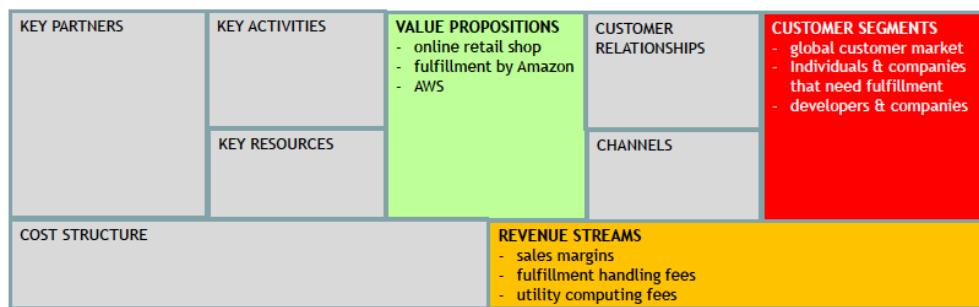
Airbnb: valore di 35 miliardi di dollari. 2 milioni di persone a notte dormono in una struttura prenotata su Airbnb.

Epic Games: Fortnite ha 250 milioni di giocatori, guadagno di centinaia di milioni di dollari al mese grazie alle skin.

Facebook: 2,45 miliardi di persone ogni mese.

Instagram: acquistato da Facebook per 1 miliardo di dollari.

6.4. CASO DI STUDIO: AMAZON



Nato con il nome di Cadabra nel 1994, era una biblioteca online. Il business plan era di non aspettarsi profitto per i successivi 4-5 anni: il primo profitto lo hanno avuto nel 2001.

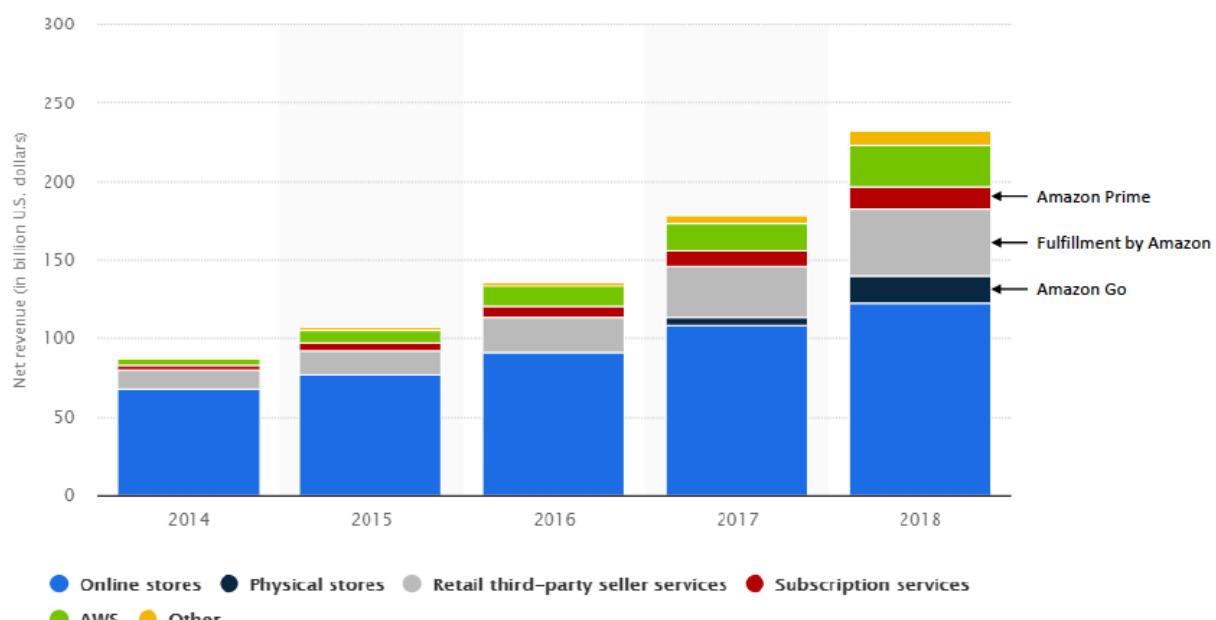
Cerca continuamente di essere innovativa con:

1. **Amazon Go:** negozi fisici senza casse.
2. **Amazon prime air:** una volta ordinato il prodotto, ti viene recapitato 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": da anni Amazon 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. È una stazione di droni, per cui può essere allocato sopra o al margine di una città, e gli ordini che vengono effettuati dagli utenti delle città possono essere serviti direttamente dal dirigibile attraverso i droni.

Alcune statistiche:

1. Dicembre 2017 – 300 milioni di utenti.
2. Gennaio 2018 – 2 milioni di vendori.
3. Gennaio 2018 – 560 mila dipendenti.
4. 3.724 miliardi di dollari di entrate ogni anno.
5. 47,8% del mercato Cloud.



6.5. CASO DI STUDIO: GOOGLE

Motore di ricerca **gratuito**.

Le aziende pagano Google per mostrare la propria pubblicità attraverso il motore di ricerca.

Google ha dominato il mercato della pubblicità graie al **customized advertising**: viene mostrata la pubblicità ai soli clienti potenzialmente interessati ad un certo prodotto.

6.5.1. RIFLESSIONI SUI MOTORI DI RICERCA

RIFLESSIONE 1

Tutti possiamo generare informazioni facilmente accessibili a tutti.

- ! Più di **4 miliardi di utenti** su Internet e quasi **2 miliardi di siti web**.

Quasi tutte le ricerche di informazioni passano attraverso un unico punto di accesso, ovvero Google.

- ! **40.000 ricerche al secondo** e circa **3.5 miliardi di ricerche giornaliere**.

Solitamente il **75% degli utenti non va oltre la prima pagina** e il **60% dei click è sui primi 3 risultati**: se non troviamo niente nei primi 3 risultati cambiamo query di ricerca.

- ! La prima pagina dei risultati di Google è una parte molto piccola di tutte le informazioni esistenti.
- ! I risultati di Google cambiano di giorno in giorno, cambiando leggermente la query, da utente a utente.



RIFLESSIONE 2

Google **non controlla le fonti né se un'informazione è vera o no.**

RIFLESSIONE 3

Google **può tenere traccia delle nostre attività** poiché possiede molti servizi.

Cosa succederebbe se Google manipolasse i risultati delle ricerche? **Search Engine Manipulation**

Effect: si è visto che è possibile spostare del 20% le preferenze di voto degli elettori indecisi mostrando degli opportuni risultati di ricerca in modo trasparente per gli utenti.

! Le manipolazioni sono quindi nascoste in modo che gli utenti non ne siano consapevoli.

RIFLESSIONE 4

Quali sono gli effetti di Google per la nostra **memoria**? Se sappiamo che possiamo avere accesso a certe informazioni in qualsiasi momento, non le ricordiamo, ma ci limitiamo a ricordare dove possiamo cercarle o trovarle.

Memoria transattiva: sappiamo che ci sono cose che non ricordiamo ma sappiamo che altre persone invece le sanno → **Internet diventa una memoria transattiva.**

7. FAAS

7.1. AWS LAMBDA

Permette agli utenti di mandare in esecuzione del codice (*funzione*) senza gestire in alcun modo i server. L'utente carica il codice sulla piattaforma e a questo punto Lambda si prende l'incarico di mandare in esecuzione il codice e di scalarlo opportunamente.

- L'utente non si preoccupa nemmeno dell'esistenza delle macchine virtuali o dei meccanismi di virtualizzazione.

Possiamo associare automaticamente al codice dei **trigger**, degli eventi che scatenano l'esecuzione della funzione. Questi trigger possono essere sia altri servizi di AWS (es. ogni volta che modifico un bucket di S3 voglio lanciare una funzione che faccia dei controlli di consistenza), sia chiamati direttamente da qualsiasi applicazione web o mobile. Ci sono 3 modi per mandare in esecuzione la funzione:

1. Caricare una funzione che l'utente ha già definito.
2. Usare l'IDE di AWS per scrivere la funzione.
3. 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 necessaria per eseguire il codice, fornendo statistiche e log in tempo reale.

7.1.1. PREZZI

Paghi solamente il tempo di calcolo necessario per eseguire la funzione caricata dall'utente; quindi, il costo di servizio è calcolato solo in base alla durata dell'esecuzione e alla memoria utilizzata.

In particolare, 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: \$0.0000166667 per ogni GB/second.

MODELLO SENZA FREEMIUM

$$\frac{\text{Richieste}}{1,000,000} \cdot \text{Costo per ogni milione di richieste} + (\text{Richieste} \cdot \text{Secondi}) \cdot \frac{\text{Memoria (MB)}}{1024} \cdot \text{Costo GB/secondo}$$

MODELLO CON FREEMIUM

La formula è la stessa, ma il primo addendo diventa

$$\frac{\text{Richieste} - \text{Richieste gratuite}}{1,000,000} \cdot \text{Costo per ogni milione di richieste}$$

E il secondo addendo

$$\left((\text{Richieste} \cdot \text{Secondi}) \cdot \frac{\text{Memoria (MB)}}{1024} - \text{Compute time} \right) \cdot \text{Costo GB/secondo}$$

7.1.2. ESEMPIO – FUNZIONE LAMBDA IN NODE.JS

Vogliamo scrivere una funzione che generi un numero casuale compreso tra due numeri scelti. Sfrutteremo l'IDE di AWS.

Scelto il template da usare per la nostra funzione, la prima cosa che facciamo è modificarlo aggiungendo un handler per la variabile exports.

```
exports.handler = (event, context, callback) => {  
    /* Codice handler */  
}  
}
```

Vediamo che ha 3 parametri in ingresso. I primi due, per quello che stiamo facendo ora, non sono molto importanti, ma il terzo è da tenere in considerazione, in quanto stabilisce quello che si vuole far restituire alla funzione associata all'handler, sia in caso di errore (null) sia in caso di successo (generatedNumber).

```
exports.handler = (event, context, callback) => {  
    let min = 0;  
    let max = 10;  
  
    let generatedNumber = Math.floor(Math.random() *max) + min;  
  
    callback(null, generatedNumber);  
}
```

Scritta la funzione, possiamo definire il nostro handler.

- Nome: index_handler, dove index indica il nome del file e handler è il nome della variabile associata a exports.
- Role per eseguire la funzione.
- Permissions.
- Timeout entro il quale il programma viene terminato se non c'è alcuna risposta.

Una volta associata una funzione ad un handler che si occuperà di gestirla e mandarla in esecuzione, dobbiamo però associarla anche a un trigger che scatenerà l'esecuzione, facendo partire l'handler. Andiamo a vedere quindi come associare la funzione ad un API Gateway.

7.1.3. ESEMPIO (PARTE 2) – CONNETTERE LAMBDA AD UN API GATEWAY

Abbiamo due modi per fare questa cosa, ma noi vedremo solo il primo.

TRAMITE LA CONSOLE DI AWS LAMBDA

Aprendo la parte di console dedicata alla gestione della nostra funzione, andiamo sul tab trigger e aggiungiamo un nuovo trigger, scegliendo come tipo di trigger API Gateway.

Gli diamo un nome, gli indichiamo il nome della risorsa cui deve fare riferimento (il nome della funzione), il metodo di comunicazione con il server (GET method), il deployment stage e il tipo di sicurezza (in questo caso OPEN perché vogliamo renderlo “evocabile” a tutti).

Abbiamo quindi creato il nostro trigger, che sarà identificato da un ID e da un URL che gli utenti potranno cliccare per far partire l'esecuzione del programma.

Tra API Gateway e funzione, chi è il trigger? L'**API Gateway**.

IMPORTANTE

7.2. QUALE FAAS USARE?

BUSINESS WIEV		
	COMMERCIALI	OPEN SOURCE
	AWS Lambda Google Cloud Functions MS Azure Functions	Apache Openwshisk Fission Fn Knative Kubeless Nuclio OpenFaaS
Licensing	Usano licenze proprietarie, a parte Azure.	Usano Apache 2.0: licenza permissiva.
Installation	Quasi tutti i server sono usati <i>as-a-service</i> (come servizio), tranne Azure.	Quasi tutti i server sono usati <i>on-premise</i> (attraverso installazione).
Source code	Non sono rilasciate come open source, tranne alcune parti di Azure.	Sono ospitate su GitHub e nella maggior parte dei casi sono implementate in Go. <ul style="list-style-type: none"> • Si riduce il problema di vendor lock-in.
Release	Sono sempre aggiornate e in produzione.	Sono leggermente indietro rispetto alle piattaforme commerciali.
Interface	Supportano <i>command line interface</i> (CLI). Le API e le GUI non sono sempre fornite.	
Community	Stack Overflow.	GitHub.
Documentation	Informazioni su come sviluppare applicazioni e su come usare la piattaforma.	Informazioni su come sviluppare applicazioni e su come usare la piattaforma. Non molti dettagli sullo sviluppo della piattaforma.

TECHNICAL WIEV		
	COMMERCIALI	OPEN SOURCE
	AWS Lambda Google Cloud Functions MS Azure Functions	Apache Openwshisk Fission Fn Knative Kubeless Nuclio OpenFaaS
Development	I linguaggi di programmazione più supportati sono Java, Node.js, Python, Docker (personalizzare il runtime). Danno la possibilità di utilizzare editor.	I linguaggi di programmazione più supportati sono Java, Node.js, Python, Docker (personalizzare il runtime).
Versioning	Permettono di avere meccanismi dedicati.	Permettono spesso solo di fare <i>implicit version</i> .
Event sources	Tutte le piattaforme supportano l'invocazione sincrona basata su http, mentre le invocazioni asincrone sono supportate da poche piattaforme.	

	Sono supportati meccanismi di schedulers per processare flussi di dati e messaggi.	
Function orchestration	La maggior parte delle piattaforme offre un “orchestratore” di funzioni dedicato.	
Testing & Debugging	Sono integrati all'interno dell'ambiente.	Si possono fare solo delle test call o debuggin basato sui log.
Observability	Si usano strumenti dello stesso provider.	Integrano servizi di terze parti.
Application delivery	Approccio dichiarativo del deployment: permette all'utente di dichiarare qual è lo stato che si vuole raggiungere senza dichiarare i passi.	
Code reuse	Lambda e Azure offrono tante funzioni già pronte.	
Access management	Supportano autenticazioni alle risorse.	Si appoggiano a servizi esterni.

Poter scrivere così facilmente le funzioni e renderle collegabili è un grosso vantaggio in termini di velocità di sviluppo e riduce la quantità di testing, ma una volta sviluppata tutta l'applicazione, se si vuole cambiare provider si deve aprire il codice sorgente e modificarlo.

FaaSener Prototype: permette di scegliere il FaaS più adatto.

7.3. LABORATORIO FAAS

In questo laboratorio abbiamo usato una **Lambda Function** per “criptare” e “decriptare” un file di testo.

! La funzione è già fornita (`lambda_function.py`).

Tutti gli step vanno eseguiti tramite Console.

7.3.1. STEP 1 – SETTING UP (1)

1. Creare una funzione Lambda.
2. Creare un nuovo **Ruolo IAM** durante la creazione della funzione.
3. Aggiungere il codice scaricato.
4. Aggiungere i permessi **full-access** su S3 e Lambda.

7.3.2. STEP 2 – SETTING UP (2)

1. Aggiungere un Livello (Layer) usando l'archivio “`python.zip`”.
 - a. **N.B.** Il file `.zip` non va rinominato, né modificato.
2. Creare Bucket contenente 3 cartelle:
 - a. `crypted/`
 - b. `decrypted/`
 - c. `plain/`

7.3.3. STEP 3 – TESTING

1. Dalla Console Lambda, eseguire un test che simuli la ricezione di un “**Evento di PUT**”.
 - a. Un file (specificato nell'evento) deve essere già caricato nella cartella `plain` quando si esegue il test.
2. Per automatizzare l'Evento:
 - a. Aggiungere un **Trigger** alla funzione Lambda.
 - b. **N.B.** L'evento che scatena il trigger deve essere l'**inserimento** di un file con **suffisso .txt** all'interno della **cartella plain** del bucket.

7.3.4. SOLUZIONE

La soluzione e lo svolgimento completo non sono presenti, tuttavia riportiamo due osservazioni:

1. Una Lambda function è fatta da 3 “pezzi”: il trigger, il codice con i Layer e le **destinazioni**. Una destinazione, a differenza di un trigger, viene invocata quando decidiamo noi (se la funzione va a buon fine, se non va a buon fine ecc...) e può essere un insieme ristretto di servizi di Amazon.
2. È consigliabile utilizzare due bucket diversi per input e output: utilizzando lo stesso bucket, si rischia di creare un loop, in quanto se si ha come trigger l'inserimento di un file in un bucket, e si inserisce proprio la funzione che serve a inserire un file in un bucket, si entra in un loop infinito → Amazon ti avverte di questa cosa e ti chiede se si è sicuri di procedere.

8. GREEN COMPUTING

8.1. INTRODUZIONE

L'ICT causa inquinamento, contribuisce ai gas serra e produce **e-waste** (spazzatura elettronica). Si producono annualmente **50 milioni di tonnellate di e-waste**.

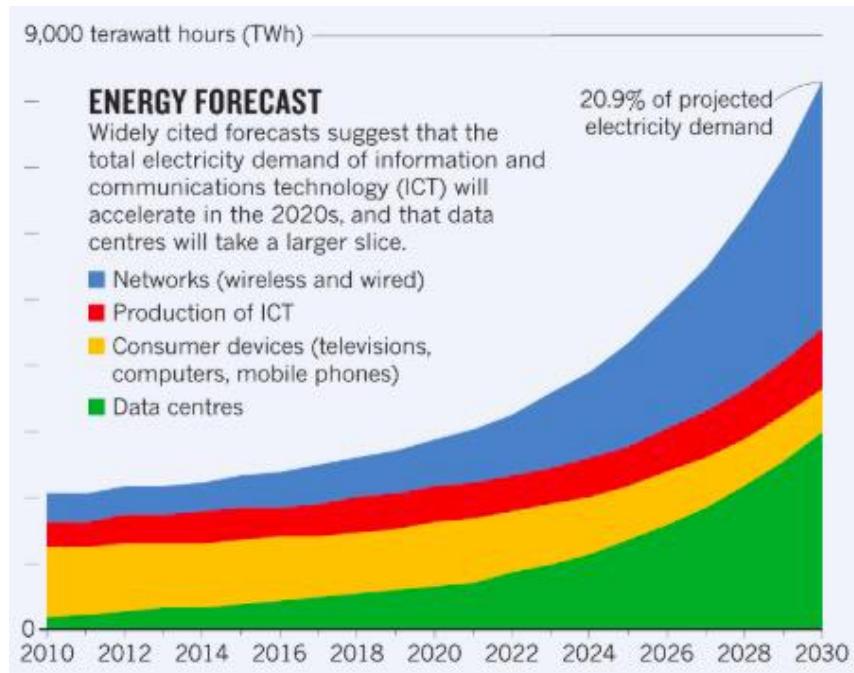
GREEN COMPUTING: pratiche per avere un ICT sostenibile dal punto di vista ambientale.

- Diminuzione del consumo di elettricità.
- Progettazione di dispositivi efficienti dal punto di vista energetico.
- Riduzione dell'e-waste.

Il consumo elettrico in Europa è del 9% per ICT, mentre il 4% per le emissioni CO₂.

! Per produrre 1 PC sono necessari 1.800 kilogrammi di materie prime.

8.1.1. PREVISIONI SUL CONSUMO ENERGETICO



8.2. DATACENTER

Il Cloud computing è realizzato tramite apparecchiature che si trovano all'interno di grandi centri di dati, chiamati appunto datacenter. In queste strutture ci sono centinaia di server che:

1. Richiedono molta elettricità per funzionare.
2. Tendono a scaldare l'ambiente, quindi hanno bisogno di sistemi di raffreddamento.

Uno dei problemi grossi, che consuma di più, è quindi il raffreddamento di tutte le apparecchiature: **il 40% del consumo energetico è dovuto ai sistemi di raffreddamento.**

8.3. IL PROBLEMA DEL PUE

PUE: indice che cerca di fare un rapporto tra l'energia necessaria a far funzionare un datacenter e l'energia necessaria per la parte di IT.

- Misura l'efficienza energetica ma non misura il grado di utilizzo delle energie rinnovabili.

8.4. OBSOLESCENZA PROGRAMMATA

Un prodotto è designato per avere un ciclo di vita breve.

- Non deve essere troppo breve, altrimenti il cliente pensa che non sia buono: bisogna fare in modo che il prodotto duri abbastanza affinché il cliente sia contento e che pensi sia di buona qualità.

È una delle cose che produce più e-waste.

ESEMPIO (1): nel 1925, le principali multinazionali statunitensi ed europee produttrici di lampadine a incandescenza fondarono il cartello di Phoebus, che ha standardizzato l'aspettativa di vita delle lampadine a 1.000 ore, rispetto alle 2.500 ore iniziali.

ESEMPIO (2): nell'ottobre del 2018, Apple e Samsung furono multate dall'ACGM per aggiornamenti software che causarono problemi e riduzione delle funzionalità di alcuni cellulari.

8.5. OBSOLESCENZA PERCEPITA

Un cliente è convinto che ha bisogno di un nuovo prodotto anche se il prodotto in suo possesso funziona perfettamente.

La nuova versione del prodotto deve essere **visibilmente differente**: si deve vedere che il prodotto appena uscito sul mercato è nuovo rispetto a quello che il cliente ha.

8.6. REPORT GREENPEACE: CLICKING CLEAN

Parla di come si comportano le varie grandi aziende informatiche come Apple, Facebook e Google rispetto alla sostenibilità ambientale: nel 2009 ha lanciato loro una sfida, spingendole ad utilizzare 100% energia rinnovabile.

Le 3 aziende iniziarono a partecipare nel 2013. Qualche anno dopo, nel 2017, altre 20 aziende hanno deciso di fare lo stesso. Perché queste aziende avrebbero dovuto partecipare?

1. Aumento dei **costi** per i contratti a lungo termine per la fornitura di energia rinnovabile.
2. Crescente preoccupazione per il cambiamento climatico tra dipendenti e **clienti**.

A partire dal 2010, l'acquisto diretto di energia rinnovabile negli USA ha cominciato ad aumentare, superando 3.4GW nel 2015.

Tuttavia, ci sono dei problemi:

1. **Mancanza di trasparenza**: dati veritieri o non completi.
 - a. Amazon ha aumentato il numero di Datacenter in Virginia.
2. **Mancanza di accesso alle energie rinnovabili**: in Cina solo 5%, in Corea del sud solo 1.1%, in Virginia (USA) meno dell'1%.

Apple e Google utilizzano sempre più energia rinnovabile. Amazon non è né trasparente né veritiero.

8.7. TRE AZIONI PRINCIPALI DA PORTARE AVANTI

MINIMIZZARE IL CONSUMO DI ELETTRICITÀ: limitare il numero di dispositivi in standby in casa, mentre per quanto riguarda i datacenters migliorare il raffreddamento (es. raffreddamento liquido).

RENDERE PIU' EFFICIENTE LO SFRUTTAMENTO DELL'ENERGIA ELETTRICA DEI DISPOSITIVI:

DISPOSITIVI: designare hardware efficienti, introdurre l'energia come un requisito nella produzione, nel testing, nella progettazione e nella manutenzione di software, minimizzare la comunicazione tra processi/microservizi, eliminare controlli e loop non necessari, avere applicazioni che si adattano all'ambiente se hanno meno energia a disposizione, scegliere attentamente i linguaggi di programmazione, minimizzare il movimento dei dati (usando efficientemente il caching).

RIDURRE L'E-WASTE.

8.8. IOT ENVIRONMENTAL IMPACT CALCULATOR

Modello analitico per stimare l'impatto ambientale della distribuzione di IoT per distribuzioni all'aperto alimentate da tecnologie di raccolta di energia solare. Permette:

1. Una stima dell'impatto **sull'energia e sui rifiuti** durante l'intero ciclo di vita del sistema:

Produzione → Uso → Mantenimento → Smaltimento

2. Comparare l'implementazione solita dell'IoT con una **soluzione equivalente, ma più green**.

8.9. CONCLUSIONI

Il greencomputing è un po' in contrasto con gli obiettivi dell'ICT: rischiamo infatti di creare un'applicazione poco sicura, e la scalabilità e la replicazione di istanze consumano più energia.

Inoltre, applicare le tecniche per il risparmio energetico non conviene a livello economico.

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

- ! Ogni anno ci sono 50 tonnellate di e-waste.

8.10. LABORATORIO: INGEGNERIA DEL SOFTWARE SOSTENIBILE

8.10.1. NOZIONI DI BASE

CO₂	<ul style="list-style-type: none"> • Fondamentale per la fotosintesi clorofilliana. • È il principale gas serra nell'atmosfera terrestre (da una sua produzione incontrollata deriverebbe un aumento dell'effetto serra, il quale contribuisce al surriscaldamento globale per il 70%).
CO₂-eq	Esprime l'impatto sul riscaldamento globale di una certa quantità di gas serra rispetto alla stessa quantità di anidride carbonica.
Joule [J]	Misura l'energia.
Watt [W]	Misura la potenza, corrisponde a J/s.
Wattora [Wh]	Misura l'energia (1Wh = 3600J).
Efficienza	<p>Da tenere in mente che</p> $\varepsilon = \frac{\#calcoli}{elettricità}$ <p style="text-align: center;"><i>Efficienza ≠ Correttezza</i></p>

8.10.2. 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**.



Impronta ecologica massima consentita secondo l'accordo:
2000kg di CO₂ annui per persona.

Quanta CO₂ produce un cittadino europeo ogni anno? **10.000kg**.

8.10.3. SOFTWARE E SOSTENIBILITÀ'

Il Digitale impatta sulla CO₂ prodotta: aumenta l'impronta ecologica, ma può anche aiutare a ridurla, rendendo più efficiente l'uso delle risorse che usiamo quotidianamente.

È fondamentale approfondire il rapporto tra digitalizzazione e sostenibilità, quindi parlare di **ingegneria del software sostenibile** è importante.

BLOCKCHAIN (BITCOIN)

Transazioni economiche senza intermediari, investimento redditizio (1 ₿ = 38.000€).

Single Bitcoin Transaction Footprints

Carbon Footprint	Electrical Energy	Electronic Waste
1180.64 kgCO ₂ 	2116.75 kWh 	354.20 grams 

Equivalent to the carbon footprint of 2,616,707 VISA transactions or 196,773 hours of watching YouTube.

Equivalent to the power consumption of an average U.S. household over 72.55 days.

Equivalent to the weight of 2.16 iPhones 12 or 0.72 iPads. (Find more info on e-waste [here](#).)

Annualized Total Bitcoin Footprints

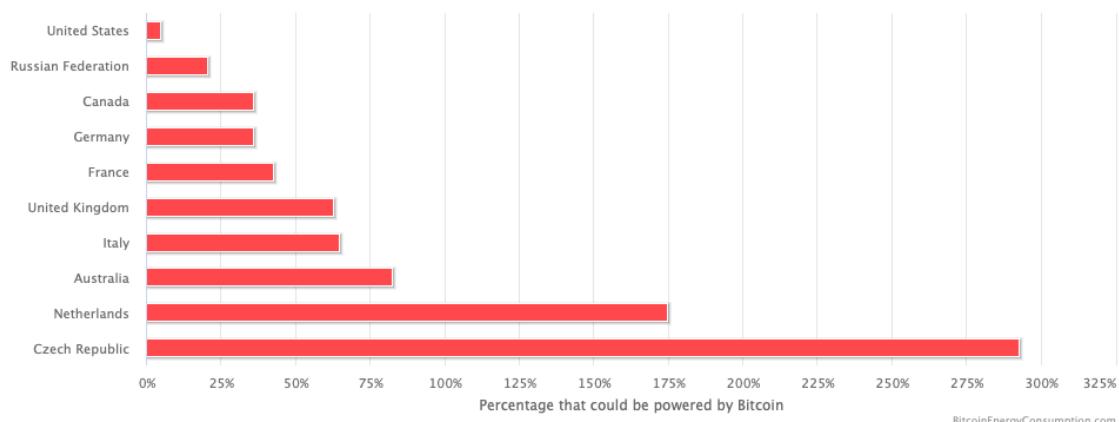
Carbon Footprint	Electrical Energy	Electronic Waste
114.06 Mt CO ₂ 	204.50 TWh 	34.22 kt 

Comparable to the carbon footprint of the Czech Republic.

Comparable to the power consumption of Thailand.

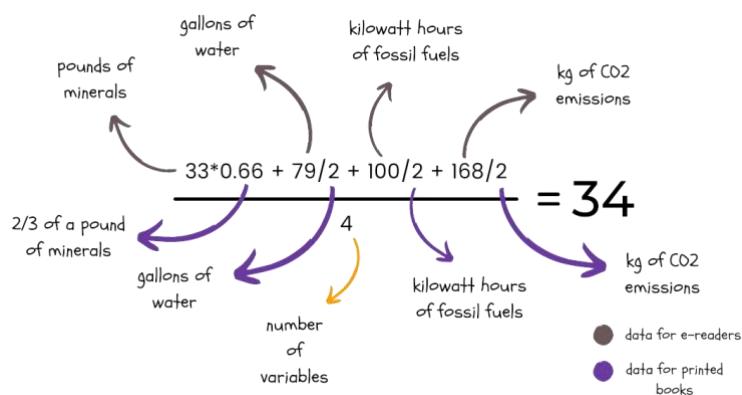
Comparable to the small IT equipment waste of the Netherlands.

Bitcoin Energy Consumption Relative to Several Countries



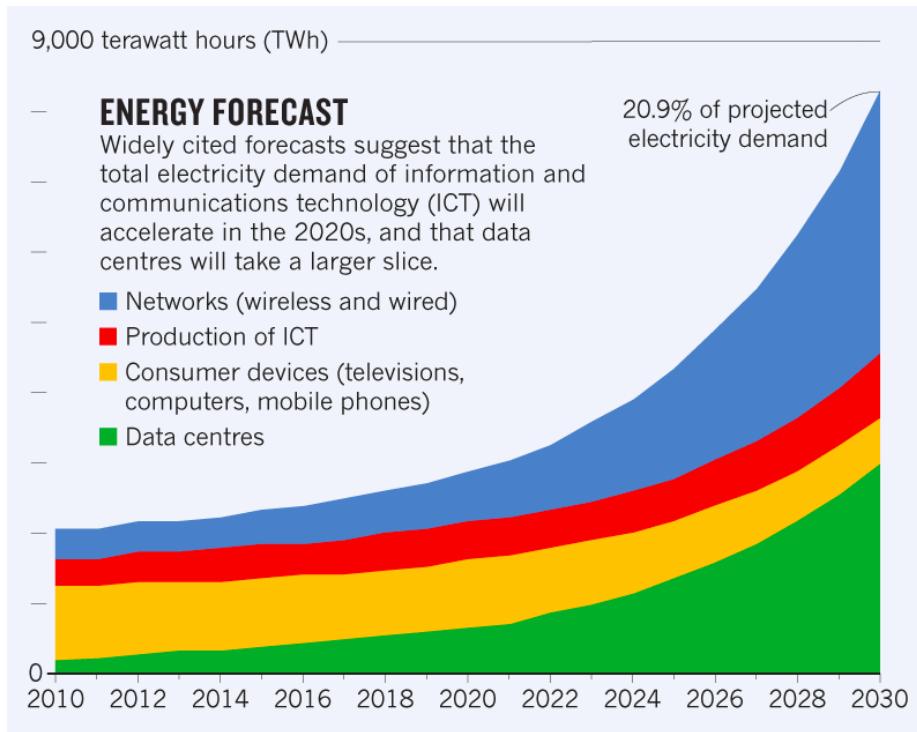
EBOOK READER

Infinite letture a disposizione senza dover stampare e senza peso/volume.

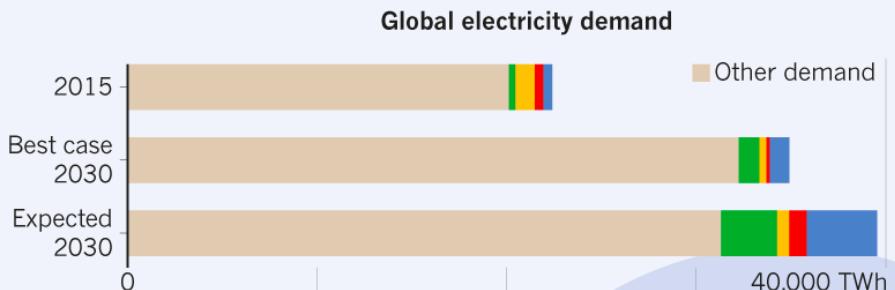


8.10.4. CONSUMO ICT

- Attualmente è tra il 5% e il 10% della domanda energetica globale.
- Produce il 2% delle emissioni di CO₂ globali.
- I data center potranno arrivare al 28% della domanda energetica per ICT.
- Le reti supereranno il 30%.



The chart above is an ‘expected case’ projection from Anders Andrae, a specialist in sustainable ICT. In his ‘best case’ scenario, ICT grows to only 8% of total electricity demand by 2030, rather than to 21%.



INTERNET EXPLOSION

Internet traffic* is growing exponentially, and reached more than a zettabyte (ZB, 1×10^{21} bytes) in 2017.



*Traffic to and from data centres.

†TB, terabyte (10^{12} bytes); PB, petabyte (10^{15} bytes); EB, exabyte (10^{18} bytes).

Quanto CO₂ produce l'invio di una mail? **20 g.**

Quanta CO₂ produce una ricerca sul web? **1 g.**

8.10.5. INGEGNERIA DEL SOFTWARE E MODELLO GREENSOFT

INGEGNERE DEL SOFTWARE: una persona che applica i principi dell'ingegneria del software per progettare, sviluppare, mantenere, testare e valutare il software informatico.

IL TRIANGOLO DI FERRO: gli ingegneri del software devono perseguire obiettivi contrastanti continuamente.

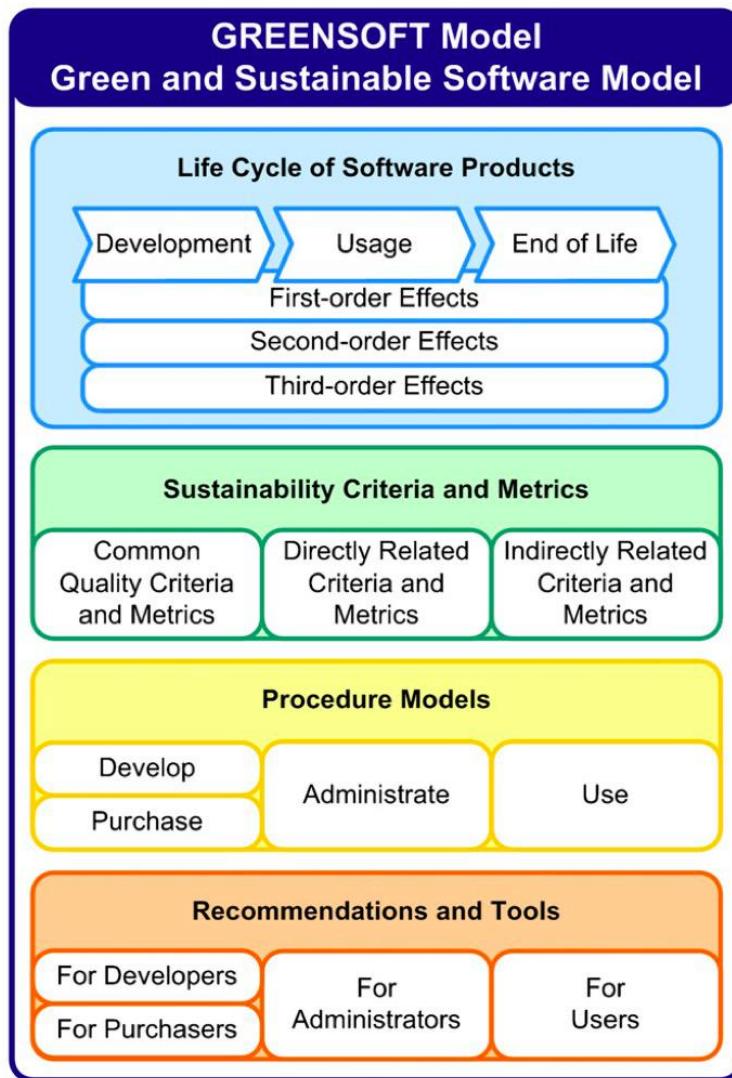
- Tipicamente: **costo, tempo e qualità.**

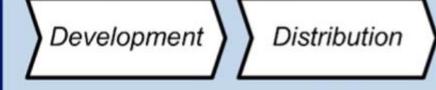
Devono inoltre soddisfare i requisiti funzionali e non-funzionali del loro prodotto.

Come considerare la sostenibilità?

1. Come un nuovo obiettivo: complica ulteriormente le decisioni.
2. Allineando lo sviluppo ai principi dell'ingegneria del software sostenibile.

IL MODELLO GREENSOFT: un **modello olistico per lo sviluppo, utilizzo e dismissione di software sostenibile.**

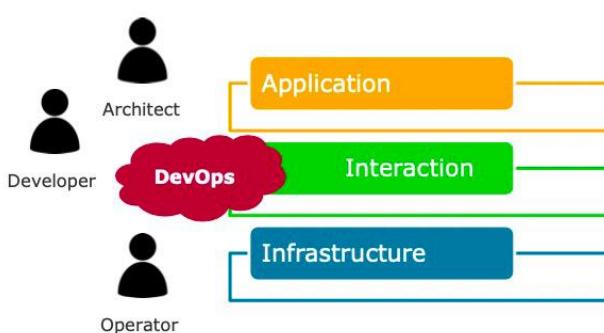


	Development	Usage	End of Life
Third-order Effects	<ul style="list-style-type: none"> - ... - Changes in software development methods - Changes in corporate organizations - Changes in life style 	<ul style="list-style-type: none"> - ... - Rebound effects - Changes of business processes 	<ul style="list-style-type: none"> - ... - Demand for new software products
Second-order Effects	<ul style="list-style-type: none"> - ... - Globally distributed development - Telework - Higher motivation of team members 	<ul style="list-style-type: none"> - ... - Smart grids - Smart metering - Smart buildings - Smart logistics - Dematerialization 	<ul style="list-style-type: none"> - ... - Media disruptions
First-order Effects	<ul style="list-style-type: none"> - ... - Daily way to work - Working conditions - Business trips - Energy for ICT - Office HVAC - Office lighting <ul style="list-style-type: none"> - ... - Manuals - Transportation - Packaging - Data medium - Download size 	<ul style="list-style-type: none"> - ... - Accessibility - Hardware requirements - Software induced resource consumption - Software induced energy consumption 	<ul style="list-style-type: none"> - ... - Backup size - Long term storage of data (due to legal issues) - Data conversion (for future use) <ul style="list-style-type: none"> - ... - Manuals - Data medium - Packaging
			

8.10.6. CASO DI STUDIO: AGGIORNAMENTO DI UN SISTEMA OPERATIVO



RUOLI E COLLABORAZIONE



Diversi ruoli possono essere ricoperti da una stessa persona.

Ruoli ricoperti da persone diverse necessitano collaborazione tra queste.

DevOps fonde i ruoli di sviluppatore, architetto e operatore.

PROSPETTIVA UTENTE

Gli utenti sono coloro che utilizzeranno il nostro sistema di aggiornamento. Cosa desiderano dal software? Il nostro sistema attualmente presenta le seguenti caratteristiche:

1. Durata aggiornamento: 10 minuti (assumendo che il dispositivo faccia solo l'aggiornamento).
2. Consumo medio: $50W \rightarrow 50W * \frac{1}{6}h = 8.3Wh$.

Un aggiornamento al mese implica 100Wh in un anno, ovvero 0.1kWh.

1 miliardo di utenti Windows: $0.1 * 10^9 kWh = 10^8 kWh = 100GWh$ in un anno.

Abbiamo quindi 43.259 tonnellate di CO₂-eq.

Inoltre, abbiamo i seguenti costi:

1. Costo elettricità: 0.3€/kWh.
2. Costo aggiornamenti annui: 0.03€.
3. Costo produttività: 10min/mese * 12 mesi = 2h.

Gli utenti non saranno interessati a ridurre il costo elettrico, saranno piuttosto interessati a ridurre il tempo di aggiornamento.

Cosa può fare un utente?

1. Essere consapevole del proprio impatto ambientale.
2. Richiedere del software di qualità migliore per evitare patch.
3. Richiedere che la sostenibilità sfrutti requisiti esistenti.

L'azienda dovrà di conseguenza

1. Produrre software di qualità migliore.
2. Evitare il più possibile aggiornamenti che richiedano un riavvio, preferendo la modalità in background.

PROSPETTIVA ARCHITETTO

Gli architetti sono coloro che determinano i requisiti funzionali e non-funzionali del sistema e che progettano le interazioni tra componenti.

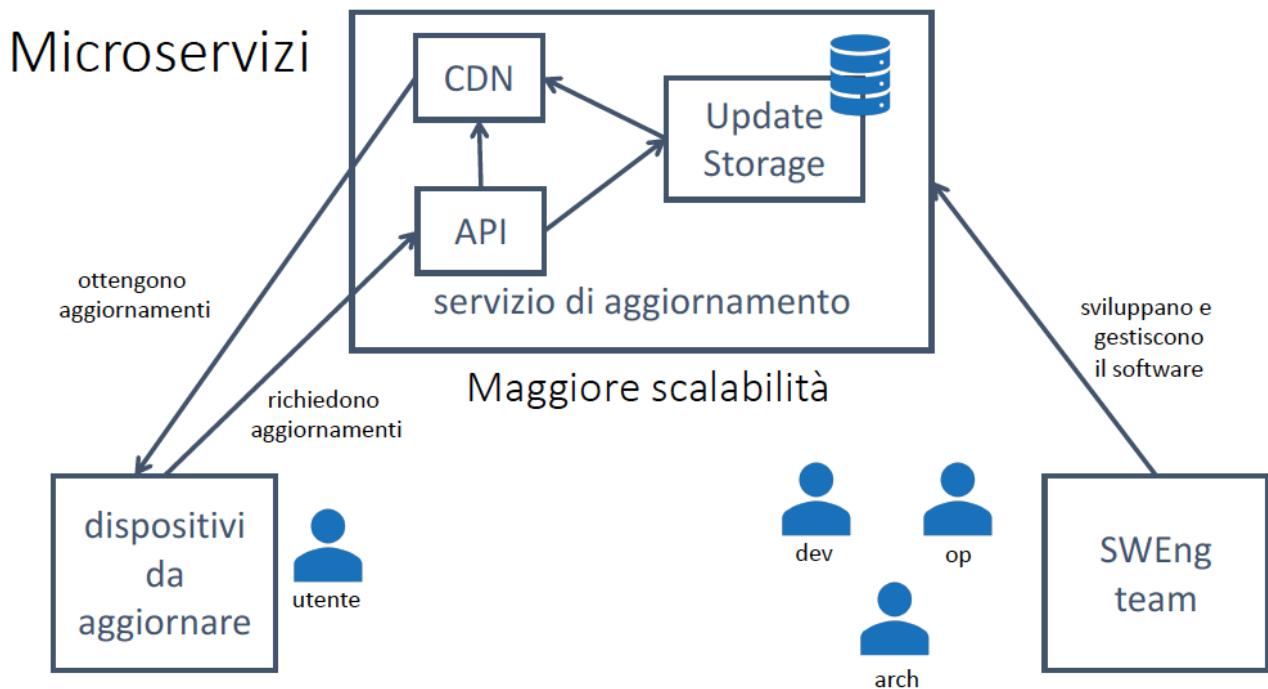
Gli architetti possono introdurre la sostenibilità nei requisiti non-funzionali:

- Scalabilità: possibilità di adattare il sistema alla domanda.
- Performance: tempo di risposta nel Service Level Agreement.

Come? Scomponendo i servizi in unità scalabili “per risorsa” (calcolo, rete, storage), tramite CDN o microservizi, o rendendo possibile un bin packing ottimo sull'utilizzo delle risorse di calcolo.

CDN 101: mantenere i dati dell'aggiornamento in “punti di presenza” locali.

- Minor distanza per raggiungere i dispositivi cliente.
- Minore utilizzo della banda, meno nodi infrastrutturali.
- Miglior utilizzo della rete, mantenendo contenuti popolari in cache.



BIN PACKING: mettere N oggetti di taglie diverse in K contenitori di capacità C, minimizzando il numero di contenitori.

- È un problema NP-hard.
- Corrisponde a minimizzare il numero di server per far girare i servizi che compongono il sistema.
- L'architetto può disegnare componenti più piccole che consentono di risolvere il problema del bin packing evitando la frammentazione.
- La soluzione del problema verrà poi delegata a livelli più bassi gestiti da altri ruoli.

Cosa può fare l'architetto?

1. Definire requisiti che consentano di essere sostenibili (scegliere un tempo di caricamento che può essere tollerato dall'utente e gestito con poche risorse).
2. Scomporre un servizio in più servizi scalabili a seconda del carico di risorse, identificando i fattori di scalabilità.
3. Scegliere una grana fine di scomposizione per facilitare il bin packing a livello più bassi.

PROSPETTIVA SVILUPPATORE

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

Affrontano delle scelte tecniche, quali linguaggio di programmazione (C++ o Python, ad esempio), scelta di librerie con algoritmi efficienti oppure scelta di pattern efficienti (“a eventi” o polling, ad esempio).

SCELTE IMPLEMENTATIVE:

1. Implementare servizi che richiedano un quantitativo ragionevole di risorse anche per favorire il bin packing.

JAVA	PYTHON
<ul style="list-style-type: none"> • Compilazione just-in-time. • Garbage collector efficiente. • Multi-thread. • VM: 2 vCPU e 2GB RAM. 	<ul style="list-style-type: none"> • Interpretato. • Reference counting. • Limiti di concorrenza. • VM: 1 vCPU e 128 GB RAM.

2. Disaccoppiare implementazione da infrastruttura in modo da facilitare la migrazione a Cloud provider differenti (ad esempio K8S che permette di spostare l'applicazione su cluster diversi).

Lo sviluppatore deve scegliere un linguaggio e trovare quindi un bilanciamento tra energia consumata, tempo di esecuzione e memoria utilizzata.

- ! Dipende anche dall'esperienza del team di sviluppo.

Table 5. Pareto optimal sets for different combination of objectives.

Time & Memory	Energy & Time	Energy & Memory	Energy & Time & Memory
C • Pascal • Go	C	C • Pascal	C • Pascal • Go
Rust • C++ • Fortran	Rust	Rust • C++ • Fortran • Go	Rust • C++ • Fortran
Ada	C++	Ada	Ada
Java • Chapel • Lisp • Ocaml	Ada	Java • Chapel • Lisp	Java • Chapel • Lisp • Ocaml
Haskell • C#	Java	OCaml • Swift • Haskell	Swift • Haskell • C#
Swift • PHP	Pascal • Chapel	C# • PHP	Dart • F# • Racket • Hack • PHP
F# • Racket • Hack • Python	Lisp • Ocaml • Go	Dart • F# • Racket • Hack • Python	JavaScript • Ruby • Python
JavaScript • Ruby	Fortran • Haskell • C#	JavaScript • Ruby	TypeScript • Erlang
Dart • TypeScript • Erlang	Swift	TypeScript	Lua • JRuby • Perl
JRuby • Perl	Dart • F#	Erlang • Lua • Perl	
Lua	JavaScript Racket	JRuby	
	TypeScript • Hack PHP Erlang Lua • JRuby Ruby		

È difficile misurare la sostenibilità, ma è possibile misurare il soddisfacimento dei requisiti.

Lo sviluppatore deve istruire il codice per:

1. Rendere disponibili le metriche funzionali (aggiornamenti consegnati).
2. Rendere disponibili le metriche non-funzionali (tempo medio di download).

È utile identificare identificatori di performance come rapporti, ad esempio il tempo di uso della CPU rispetto agli aggiornamenti consegnati.

Anche durante lo sviluppo si consumano risorse. Occorre quindi:

1. Non sprecare energie: implementare la cosa giusta e farlo bene.
2. Configurare pipeline che facilitino lo sviluppo del software.
3. Testare automaticamente per evitare difetti software.

Cosa può fare lo sviluppatore?

1. Cercare di ottimizzare l'efficienza del software.
2. Disaccoppiare applicazione da dettagli sull'infrastruttura.
3. Istrumentare e misurare.
4. Ridurre lo spreco di risorse durante lo sviluppo.

PROSPETTIVA OPERATORE

Gli operatori sono coloro che gestiscono l'infrastruttura e garantiscono la disponibilità dell'applicazione. Devono:

1. Offrire risorse infrastrutturali adeguate a garantire le performance desiderate.
2. Utilizzare al meglio le risorse per contenere i costi e favorire la sostenibilità.
3. Garantire backup e ripristino.
4. Implementare log e monitoraggio.

Per quanto riguarda le piattaforme, devono:

1. Scegliere piattaforme che consentano un bin packing vicino all'ottimo (virtualizzazione, container, serverless).
2. Monitorare KPI infrastrutturali e KPI aziendali.
3. Automatizzare lo scaling dei servizi (autoscaling).
4. Sfruttare la località dei dati per minimizzare la trasmissione di rete e velocizzare la performance.
5. Evitare backup/cronjob periodici se non necessari.
6. Usare istanze "spot" per task non prioritari (ottimizza l'uso delle risorse).

Cosa può fare un operatore?

1. Fare scelte infrastrutturali che garantiscano performance e facilitino il bin packing.
2. Monitoraggio, log, backup.
3. Ottimizzare l'uso delle risorse.

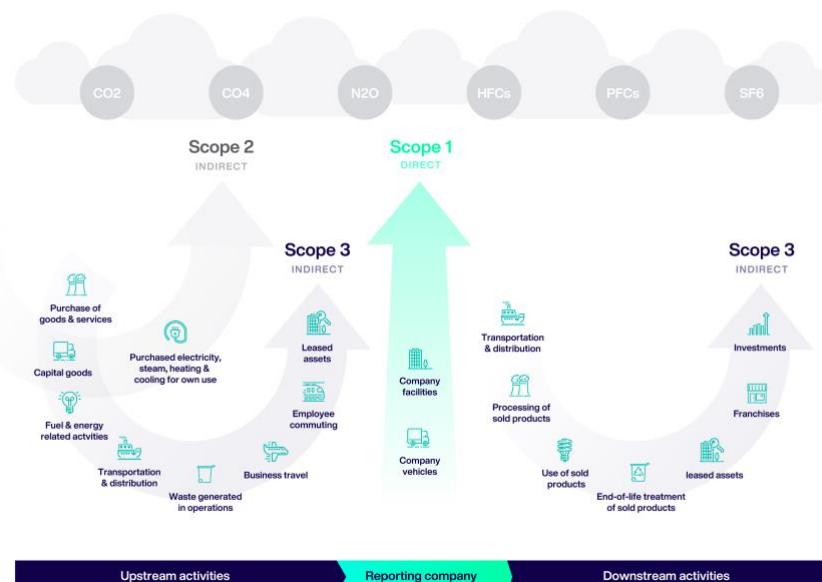
8.10.7. QUADRO NORMATIVO

Le leggi e i regolamenti possono favorire una transizione sostenibile.

- Le emissioni dovute al ciclo di vita di un software sono una esternalità negativa: è un costo indiretto che viene pagato da terzi non coinvolti nel ciclo di vita di quel software.

Le leggi possono:

1. "Tassare" le esternalità negative (dallo sviluppo alla dismissione).
2. Obbligare la trasparenza per creare consapevolezza tra i clienti.



L'Unione Europea ha definito una direttiva su "Corporate sustainability reporting", all'interno del Green Deal europeo.

- Obbliga le aziende a relazionare sulla sostenibilità ambientale.
- Sarà in vigore dal 2023.

8.11. LABORATORIO GREEN COMPUTING: JAVA COLLECTION – CONSUMO ENERGETICO ED EMISSIONI

In questo laboratorio si sono svolti semplici esperimenti per stimare il consumo energetico e le emissioni di CO₂ di alcune strutture dati fornire nelle librerie standard Java.

Prerequisiti:

1. Java JDK >= 17.
2. Java IDE (esempio NetBeans).
3. Conoscenza base del linguaggio Java.

8.11.1. STRUTTURE DATI

Le strutture dati forniscono un modo per organizzare (e processare) l'informazione.

Una struttura dati *CRUD* consente:

- L'aggiunta (*create*),
- la lettura (*read*),
- l'aggiornamento (*update*) e
- l'eliminazione (*delete*) dei dati che contiene.

Java fornisce alcune strutture dati di libreria, in particolare riferibili ai tipi astratti *List* (liste, collezioni di elementi ordinate che può contenere duplicati), *Set* (insiemi, collezioni di elementi non duplicati senza ordine) e *Map* (mappe chiave-valore, associazione chiave-valore in cui vengono inseriti casualmente gli elementi e serve per alleggerire il carico, un esempio è una tabella hash).

8.11.2. COMPLESSITÀ

Strutture dati diverse offrono diverse performance in tempo per le operazioni di aggiunta, lettura, aggiornamento ed eliminazione.

Tale performance dipende dalla complessità in tempo associata alle operazioni.

Nel nostro esperimento, considereremo 4 implementazioni di strutture dati *List* e *Set*:

- **ArrayList** – create $O(1)$, read $O(1)$, update $O(N)$, delete $O(N)$.
 - La *create* è costante perché aggiunge sempre in testa o in coda.
 - Java considera una lista come un array, quindi *update* e *delete* devono scorrere tutto l'array.
- **LinkedList** – create $O(N)$, read $O(N)$, update $O(N)$, delete $O(N)$.
- **HashSet** – create $O(1)$, read $O(1)$, update $O(1)$, delete $O(1)$.
 - Le operazioni sono costanti nel caso medio.
- **TreeSet (albero rosso-nero)** – create $O(\log N)$, read $O(\log N)$, update $O(\log N)$, delete $O(\log N)$.

8.11.3. SCALE UP! – OPERAZIONI PRELIMINARI

1. Scaricare primer.zip e decomprimerlo.
2. Aprire il progetto nell'IDE di Java. Contiene 3 file:
 - a. GreenMeter.java – per eseguire gli esperimenti con diverse strutture dati.
 - b. CRUDCollectionIF – definisce un'interfaccia CRUD per incapsulare le diverse strutture dati per l'esperimento.
 - c. CRUDArrayList – implementa l'interfaccia di cui sopra.
3. Eseguire il main contenuto in GreenMeter.java. Si dovrebbe avere un output del tipo:
CPU time: 6.901457584 s
Energy: 2.4541583168704002E-5 kWh
CO2: 0.006135395792176 g

GreenMeter.java

```
public class GreenMeter {  
    public static void main(String[] args) {  
        double POWER = 20.0; // Watt consumption of host machine  
        double CO2perKWh = 250; // CO2 g/kWh  
        int MAX = 10000; // universe's max  
        int N = 250000; // elements  
        int L = 100000; // number of CRUD operations  
  
        // CRUD data structure to be tested  
        CRUDCollectionIF numbers = new CRUDArrayList(N, MAX);  
        Random rand = new Random();  
        char[] operations = {'C', 'R', 'U', 'D'}; // operations to run  
  
        double before = System.nanoTime(); // start timing  
  
        for(int s = 0; s < L; s++) {  
            switch (getRandom(operations)) {  
                case 'C' -> numbers.create(rand.nextInt(MAX));  
                case 'R' -> {  
                    if (numbers.size() > 0)  
                        numbers.read(rand.nextInt(MAX));  
                }  
                case 'U' -> {  
                    if (numbers.size() > 0)  
                        numbers.update(rand.nextInt(MAX), rand.nextInt(MAX));  
                }  
                case 'D' -> {  
                    if (numbers.size() > 0)  
                        numbers.delete(rand.nextInt(MAX));  
                }  
                default -> {}  
            }  
        }  
  
        double after = System.nanoTime(); // end timing  
  
        // we do the maths here:  
        // to seconds  
        double time = (cpuAfter - cpuBefore) / Math.pow(10.0, 9.0);
```

```

        //J/s to kWh
        double consumption = (time * POWER) / (3600 * Math.pow(10,3));
        double co2 = CO2perKWh * consumption;
        /* [...] Print the above :-) */
    }
    /* [...] */
}

```

8.11.4. SCALE UP! – ESERCIZIO

1. Seguendo l'esempio in `CRUDArrayList`, **incapsulate** `LinkedList`, `HashSet` e `TreeSet` all'interno dell'interfaccia `CRUDCollectionIF`.
2. Configurate `GreenMeter.java` con il consumo energetico della vostra macchina, lasciate invariati gli altri elementi. **Stiamo stimando il consumo energetico e le emissioni di un mix bilanciato di 1000 operazioni CRUD su collezioni di lunghezza 250K con valori interi in [0, 10000).**
3. Per ciascuna delle strutture dati, eseguite `GreenMeter.java` e prendete nota del risultato ottenuto.

8.11.5. RISULTATI

`ArrayList`

```

Execution time: 2.193630125 s
Energy: 1.8280251041666666E-5 kWh
CO2 emissions: 0.004570062760416667 g

```

`LinkedList`

```

Execution time: 7.032041375 s
Energy: 5.860034479166667E-5 kWh
CO2 emissions: 0.014650086197916667 g

```

Quale è più sostenibile tra `ArrayList` e `LinkedList`? Perché?

- `ArrayList` è più sostenibile di `LinkedList` nei nostri esperimenti perché sfrutta un vettore ad accesso diretto per velocizzare la lettura dei dati.

`TreeSet`

```

Execution time: 0.037318583 s
Energy: 3.1098819166666666E-7 kWh
CO2 emissions: 7.774704791666666E-5 g

```

`HashSet`

```

Execution time: 0.022017208 s
Energy: 1.8347673333333333E-7 kWh
CO2 emissions: 4.5869183333333335E-5 g

```

Quale è più sostenibile tra TreeSet e HashSet? Perché?

- La struttura dati più sostenibile nei nostri esperimenti è HashSet (CRUD: O(1)).

8.11.6. CONSIDERAZIONI SUI RISULTATI

- Le strutture di tipo List offrono performance peggiori rispetto a quelle di tipo Set, ma possono contenere elementi duplicati.
- Se si prevede un mix bilanciato di operazioni CRUD, ArrayList risulta migliore di LinkedList. Tuttavia, ArrayList è meno efficiente per quanto riguarda le operazioni di modifica ed eliminazione dei dati perché gli array non consentono l'allocazione dinamica della memoria. D'altro canto, LinkedList implementa anche l'interfaccia Queue che si rivela utile in alcuni scenari.
- Se HashSet è la struttura più sostenibile, perché non usarla sempre? Dipende dal contesto d'uso. TreeSet contiene collezioni ordinate e fornisce operazioni quali first() e last(). Viceversa, HashSet richiede $O(N \log N)$ per ordinare i dati. Se è importante mantenere i dati ordinati, è meglio optare per TreeSet.

8.11.7. CONCLUSIONI

- La complessità in tempo delle operazioni su una struttura dati CRUD è fortemente correlata al consumo di energia e alle emissioni di CO₂ di un programma.
- Per scegliere la struttura dati migliore bisogna determinare un compromesso bilanciato tra il suo utilizzo nel programma e la sua sostenibilità/efficienza.
- Implementazioni diverse di una stessa struttura dati possono offrire performance e funzionalità molto diverse (esempio ArrayList vs LinkedList, TreeSet vs HashSet). È quindi importante scegliere anche l'implementazione più adeguata ai nostri scopi.

9. MICROSERVIZI

9.1. COSA SONO E PERCHE' USARLI

Tecnologia molto importante nello sviluppo software, soprattutto utilizzando Cloud.

- **Riducono il *lead time* per nuove feature e aggiornamenti.**
Lead time: tempo che passa da quando un progettista incomincia ad immaginare una funzionalità al momento in cui il primo utente fa “click” su quella funzionalità → Tempo che ci vuole per rendere utilizzabile una nuova funzionalità o un nuovo aggiornamento.
- **Riuscire a scalare in maniera efficace.**
Aumentare la capacità di risposta dell’utente, ad esempio creando delle repliche della nostra applicazione.
 - Replica di un singolo microservizio: **scalabilità orizzontale**.
 - Replica di tutta l’applicazione: **scalabilità verticale**.

Le dimensioni non sono sempre modeste.

9.2. IL PASSATO: MONOLITI

La tipica architettura di un’applicazione enterprise di vecchia generazione è composta da 3 livelli:

1. Interfaccia lato client.
2. Applicazione lato server.
3. Database.

L’applicazione lato server viene chiamato **monolite**: un unico grande eseguibile in grado di rispondere alle richieste.

PROBLEMI:

1. Un piccolo cambiamento nell’applicazione richiede il rebuilding e redeploying dell’intero monolite.
2. Impossibilità di scalare soltanto una parte dell’applicazione.

9.3. ESSENZA DEI MICROSERVIZI

SERVICE-ORIENTATION: le applicazioni sono sviluppate come insiemi di servizi, ognuno su un Container diverso, che comunicano tra di loro con meccanismi semplici (piccoli REST, richieste-risposta HTTP).

Oltre alle chiamate sincrone ci sono dei *dumb message bus* (code asincrone).

I servizi sono poliglotti, ovvero possono essere scritti in linguaggi diversi.

Nella prima versione dei microservizi venivano usati degli **EBS**, cioè dei connettori che permettevano ai servizi di collaborare tra loro (non erano code asincrone) e usavano come standard di sicurezza gli **standard WS** (Web Service).

ORGANIZZARE I SERVIZI INTORNO A DELLE BUSINESS CAPABILITIES: la *legge di Conway 1986* dice che le organizzazioni che progettano sistemi finiscono con l’essere vincolate nel produrre progettazioni che riflettono la struttura delle proprie organizzazioni.

I microservizi dicono di 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.

- Problemi di consistenza e integrità.

Eventual consistency: invece di mantenere i database sempre consistenti, accettare che per un periodo non lo siano, ma che lo saranno, prima o poi, in futuro.

- Si sacrifica la consistenza per migliorare l'efficienza.

INDEPENDENTLY DEPLOYABLE SERVICE: ogni servizio è indipendente e può essere lanciato senza lanciare gli altri servizi, si fa il rebuilt solo di una parte e non di tutta l'applicazione.

SCALABILITÀ ORIZZONTALE: si vuole scalare solamente quel servizio che ha problemi di performance.

RESISTENZA AI FALLIMENTI: se fallisce l'API Gateway, potrebbe causare un fallimento a cascata. Il problema da risolvere è che chi ha fatto la richiesta deve *rispondere al fallimento nel modo migliore possibile*.

La *fault tolerance* deve essere un requisito del sistema: 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 chiama il proxy che manda la richiesta a S. Se dovessero aumentare i tempi di risposta o non riuscisse ad ottenerla, il circuit breaker fornisce comunque una risposta.

ESEMPIO: barra di ricerca di Spotify. Se l'utente digita qualcosa, e la ricerca non funziona, semplicemente veniva svuotata la barra di ricerca, così che l'utente ridigitasse la cosa da cercare.

9.3.1. PERIODI DI DOWNTIME

Le chiamate sincrone tra i servizi inducono un effetto di downtime che si moltiplicherà nel tempo.

ESEMPIO (1): se ho 30 dipendenze (servizi), ciascuna con il 99.99% di uptime (tempo in cui il sistema funziona), io avrò che il sistema funziona quando **tutti** i servizi funzionano contemporaneamente. Quindi

$$99.99\%^{30} \times 24 \times 30 = 99.7\% \times 24 \times 30 \rightarrow \text{più di 2 ore di downtime al mese}$$

ESEMPIO (2): per offrire una funzionalità F, un microservizio invoca in modo sincrono altri due microservizi tra loro indipendenti, ciascuno dei quali con il 90% di uptime. Possiamo predire che F non sarà probabilmente disponibile per circa 6 giorni ogni mese. La probabilità che non funzioni è infatti

$$p = 1 - (0.9 \times 0.9) = 0.19$$

Calcolato su 30 giorni, vediamo quindi che i giorni di downtime sono 6.

9.3.2. TEST BRAVELY

Chaos monkey: permette di testare in modo coraggioso il funzionamento di una applicazione. Lo possiamo configurare in modo 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.

9.4. DEVOPS

Per decenni lo sviluppo del software veniva fatto da un gruppo di development; ottenuto il prodotto finale, lo passavano agli operatori che gestivano il rapporto con i clienti.

Ora con DevOps non si fanno progetti ma *prodotti*: tu lo hai costruito e tu lo mandi in esecuzione.

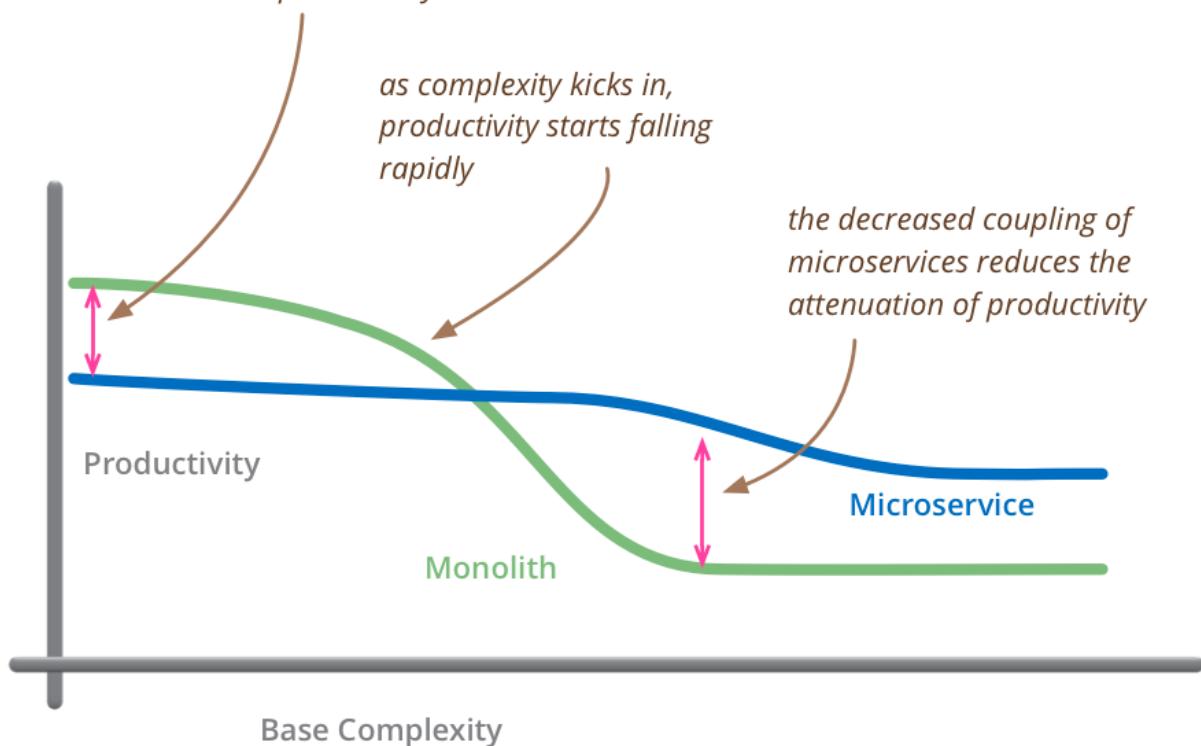
- Il team full stack che si occupa di un servizio lo progetta, lo crea e se lo gestisce.

9.5. CONCLUSIONE

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

PRO	CONTRO
<ul style="list-style-type: none"> • Minore lead time. • Scalabilità. 	<ul style="list-style-type: none"> • Eccessiva comunicazione tra i servizi. • Complessità. • Facile sbagliarsi. • Non è semplice decentralizzare i dati. • Se un team è poco preparato, avrà problemi ad utilizzare i microservizi.

for less-complex systems, the extra baggage required to manage microservices reduces productivity



10. ARCHITECTURAL SMELL E SOLUZIONI

“Cattivo odore architetturale”: un’architettura che molto probabilmente non dovrebbe essere così.

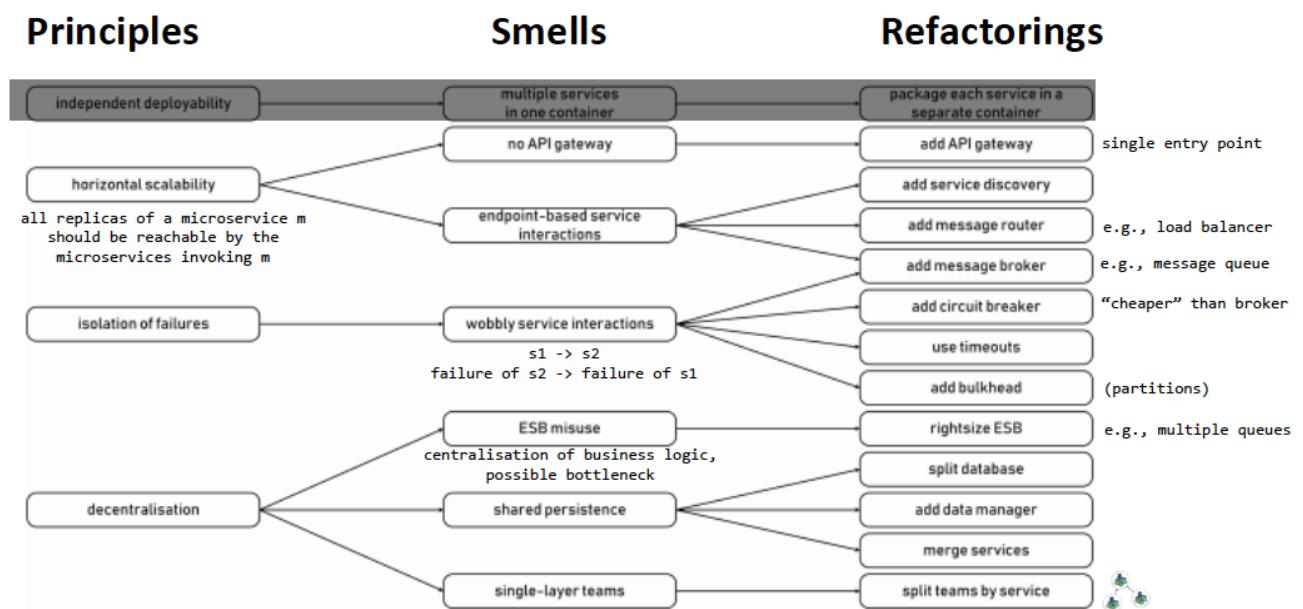
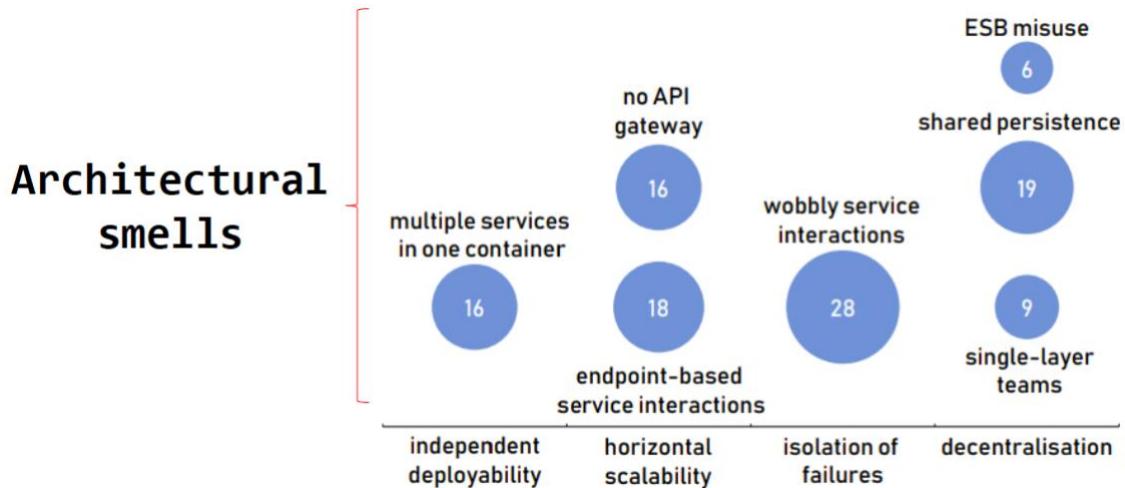
Come è possibile risolvere delle smell architetturali?

10.1. DESIGN PRINCIPLES

Abbiamo 4 design principles:

1. **Independent deployability**: i microservizi che formano un’applicazione devono poter essere sviluppati indipendentemente.
2. **Horizontal scalability**: i microservizi che formano un’applicazione devono poter scalare indipendentemente dagli altri.
3. **Isolation of failure**: dobbiamo isolare i fallimenti.
4. **Decentralization**: dobbiamo decentralizzare ogni aspetto dell’implementazione delle applicazioni basate sui microservizi.

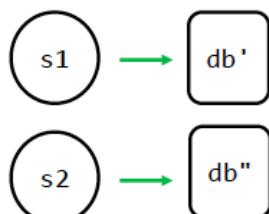
10.2. ARCHITECTURAL SMELLS



Andiamo ad analizzarli singolarmente:

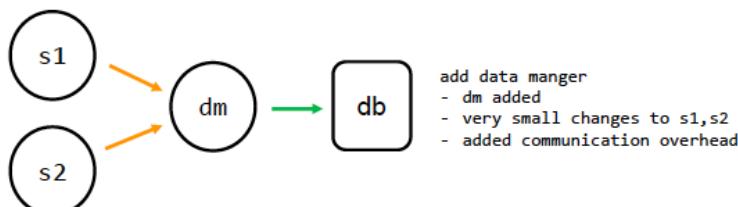
1. **Multiple services in one container:** se devo usare un servizio è difficile perché sono tutti nello stesso container → Divido i vari servizi in container separati.
2. **Endpoint-based service interactions:** potenzialmente viola la scalabilità orizzontale. Se ho un servizio che ne invoca un altro, se quest'ultimo lo faccio scalare (creando altre 2 istanze), il chiamante continua a invocare sempre lo stesso di prima.
Refactoring: uso un “intermediario” che invoca l’istanza migliore, oppure una coda in cui viene messa la richiesta e viene “consumato” il messaggio dalla coda.
3. **No API Gateway:** con un API Gateway il cliente non chiama direttamente i servizi interni, ma chiama l’API Gateway che gestisce la sua richiesta.
Refactoring: dobbiamo aggiungere l’API Gateway, in modo che i clienti non possano fare invocazioni all’interno della nostra architettura. Viene utilizzato anche per l’autenticazione.
4. **Shared persistence:** più servizi accedono e condiviscono lo stesso DB.
Refactoring:

- a. Dividere il database se possibile.



split database
- db splitted
- small changes to s1,s2
- not always possible/easy to implement
- eventual data consistency for replicated data

- b. Introdurre un data manager: s1 e s2 invocano le operazioni offerte dal data manager, così quando dobbiamo modificare il database s1 e s2 non lo sanno. Tuttavia, abbiamo 2 chiamate: una per il dm e una per il db.



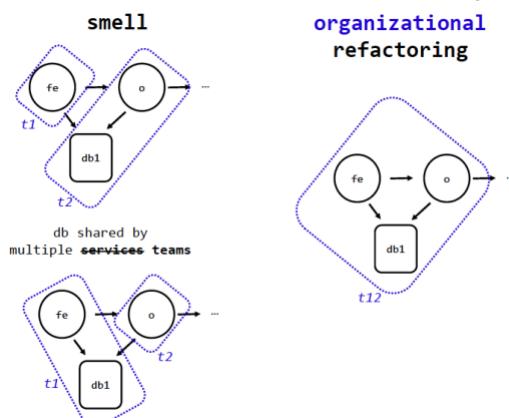
add data manager
- dm added
- very small changes to s1,s2
- added communication overhead

- c. Fare il merge dei servizi: s1 e s2 condividono lo stesso database (potrebbe essere che s1 e s2 siano lo stesso microservizio, e debbano essere uniti).



merge services
- s1 and s2 merged into single service
- not always easy to implement

5. **ESB misuse:** parte dell’intelligenza e della business logic va a finire in questo punto centralizzato (ESB) violando l’idea dei microservizi di endpoint e dumb message.
6. **Single-layer teams:** dobbiamo avere esperti di cose diverse in ogni team.



7. **Wobbly service interaction:** un'interazione è “traballante” quando un fallimento di m1 può scatenare un fallimento su m2.

Refactoring:

- a. Aggiungiamo un circuit breaker: il proxy chiude il circuito in modo che m2 non fallisca.
- b. Message broker, coda asincrona: i produttori mandano messaggi alla coda e i consumatori si limitano a leggere dalla coda → Un fallimento del produttore non genera un fallimento del consumatore.
A chiama il message broker e mette la richiesta, B prende la richiesta dal message broker.
- c. Timeouts: meccanismo con cui il chiamante ha una chiamata timed e se non arriva una risposta può interrompere l'attesa e non fallire.
- d. Bulkhead: definiamo il confine più grande (paratia) all'esterno del quale non ci deve essere condivisione dei dati e chiamate singole dei servizi
All'interno della paratia condivido dati ma quando l'attraverso non devo condividere per avere un isolamento dei fallimenti.

10.3. uFreshener (microFreshener)

Ci offre una visione grafica di queste soluzioni e di avere una visione teambased nell'ultimo aggiornamento.

Possiamo scegliere di vedere solo alcuni microservizi.

11. AZIENDE E MICROSERVIZI

11.1. SPOTIFY

Perché Spotify ha deciso di passare ai microservizi?

RIDURRE IL LEAD TIME

1. Deve innovare il proprio servizio e reagire in maniera rapida per essere competitivo in un mercato che si muove velocemente.
2. Inoltre, deve essere supportato su molte piattaforme: ogni funzionalità aggiunta deve essere supportata su ogni dispositivo.
3. Vecchie versioni del servizio devono essere funzionanti.

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

11.1.1. QUANDO E COME È AVVENUTO IL PASSAGGIO AI MICROSERVIZI?

All'aumentare del numero di richieste l'interazione con il **Database** di Spotify ha cominciato a rallentare e non scalava.

- Soluzione iniziale: replicare le istanze dei Database (scalare il backend).

All'aumentare nuovamente degli utenti il Database resisteva ma il **Server** no.

- Soluzione: hanno scalato il Server replicandolo, mettendo davanti dei Load Balance per sostenere il carico.

PROBLEMA: aumentando ancora di più gli utenti e avendo la necessità di modificare l'applicazione con tanti devide e con tante versioni questa architettura non permetteva di raggiungere entrambi gli obiettivi.

- Soluzione: passaggio ad un'organizzazione a **microservizi** e riorganizzazione dei propri team (**full-stack autonomous team**).

11.1.2. VANTAGGI DEI MICROSERVIZI

1. **Facile scalare.**
2. Più **facile fare i test.**
3. Più **facile fare il deploy.**
4. Più **facile monitorare:** possiamo monitorare parti diverse e più piccole.
5. **Versionato in maniera indipendente:** è molto importante mantenere diverse versioni della propria applicazione.
6. **Meno suscettibile ai grandi fallimenti.**

11.1.3. SVANTAGGI DEI MICROSERVIZI

1. **Difficile da monitorare** se i servizi sono tanti e le istanze dei servizi ancora di più.
2. **Bisogno di buona documentazione e discovery tools.**
3. La **latenza aumenta**, si generano più comunicazione e aumentano i ritardi.

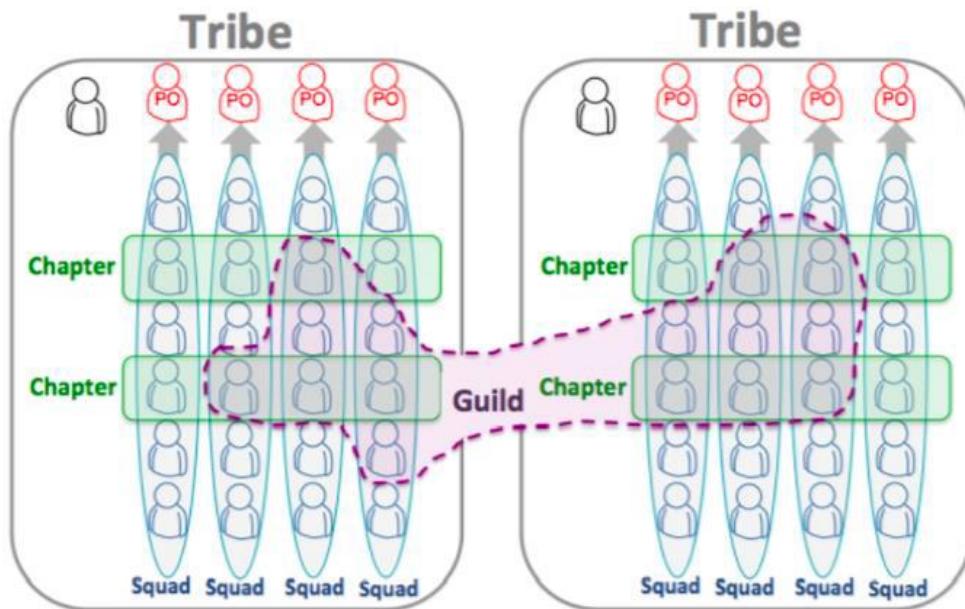
11.1.4. ALCUNI DATI

- Più di **90 teams**.
- **600 sviluppatori**.

- **5 uffici di sviluppo.**
- **1 prodotto.**
- **810 servizi attivi.**
- **10 servizi per squad.**

11.1.5. STRUTTURA DEI TEAM

La struttura dei team di Spotify si basa sull'**Agile Software Engineering**.



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

TRIBU': raccolta di squadre all'interno della stessa area commerciale. Le squadre all'interno di una tribù risiedono nella stessa area.

- Spotify implementa incontri condivisi per creare interazioni tra squadre e eventi informali dove le squadre condividono ciò su cui stanno lavorando. Il leader della tribù è responsabile di fornire e creare un ambiente ideale per tutte le squadre.

CAPITOLI: membri del team che lavorano in un'area speciale. Si scambiano idee per promuovere l'innovazione e "l'impollinazione incrociata" tra i team. Si occupano soprattutto della review del codice: quando una squadra vuole produrre una nuova versione di un servizio, chiede una review ad un paio di esperti dell'altro team che fanno parte del capitolo.

GILDA: è una comunità di membri all'interno dell'organizzazione con interessi condivisi che vogliono condividere conoscenze, codice degli strumenti e tecniche.

11.1.6. SPOTIFY ENGINEERING CULTURE

1. **Usa i principi dell'Agile Software Engineering:** teams organizzati come sopra, release frequenti e piccoli, focalizzandosi sulla motivazione, community e fiducia.
2. **Fail fast → Learn fast → Improve fast:** ci saranno sicuramente dei fallimenti, dopo i quali si fanno delle analisi e si cerca di imparare qualcosa, meglio fallire che evitare i fallimenti.
 - a. Si lavora con il **Limited blast radius**, un raggio di danno limitato: il problema di Spotify è che avendo milioni di utenti, quando si sviluppa un nuovo servizio potremmo causare

problemi a tutti → Rilasciando gradualmente prima solo ad alcuni utenti, si limitano malfunzionamenti.

3. **Approccio allo sviluppo:** principi di **Lean Startup**. L'idea è di cercare di dare più importanza all'impatto che alla velocità dell'innovazione, oltre che a dare più importanza all'innovazione rispetto alla prevedibilità. Infine, delivery value > plan fulfillment.
4. **Hack Time:** si vuole che i dipendenti dedichino il 10% del tempo in Hack Time, dove possono dedicarsi a fare cose innovative per favorire la creatività.
5. **Waste-Repellent culture:** ciò che non interessa, tutto ciò che non crea valore, si butta.
 - a. Lavagna dei miglioramenti: si scrive tutto ciò che si vuole migliorare del prodotto.
6. **Minimizzare il bisogno di “Big Projects”:** aumentano i rischi, meglio dividere in piccoli progetti in modo da minimizzare anche la burocrazia e la documentazione.
7. **Health culture:** creare un ambiente sano e piacevole per i dipendenti.

11.2. NETFLIX

Perché Netflix ha deciso di passare ai microservizi?

RIDURRE IL LEAD TIME

1. Bisogno di automatizzare e velocizzare gli aggiornamenti.
2. Supportare più di 1000 tipi di device differenti.

SCALARE

1. Più di **86 milioni di membri, 150 milioni di ore** di streaming.
2. **15% di tutto il traffico in downstream di Internet**.

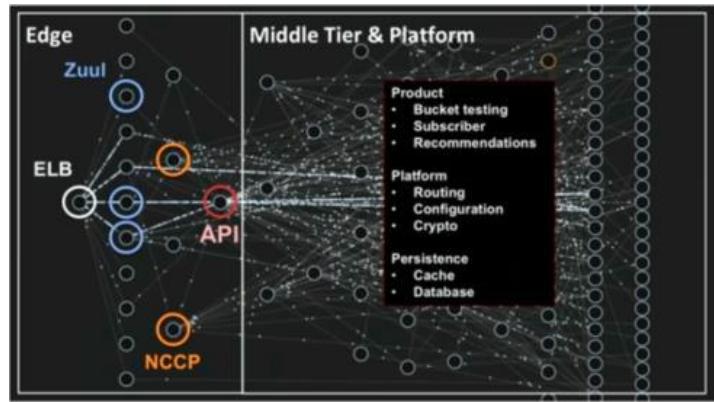
Netflix usa **100.000 istanze su Amazon Web Services**, dove si sono spostati nel periodo **2008-2016 chiudendo tutti i loro data center**.

Possiede circa **500 microservizi** ed è uno dei più grandi distributori di open source.

11.2.1. PASSAGGIO AI MICROSERVIZI

L'introduzione dei container ha creato problemi al supporto tecnologico: hanno quindi deciso di utilizzare AWS Blox di Amazon.

Netflix ha quindi deciso di passare dal monolite ai microservizi.



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

- **ELB:** Elastic Load Balancing di AWS per instradare il traffico attraverso il primo livello.

- **Zuul**: proxy per le richieste che vengono dal Load Balancing, effettua un routing dinamico fino ad arrivare all'API di Netflix.
- **API**: chiama tutti gli altri servizi.
- **NCCP**: mantiene le vecchie versioni → Legacy tier.

11.2.2. PROBLEMI DI NETFLIX CON I MICROSERVIZI

RICHIESTE TRA DIVERSI SERVIZI: avere tanti microservizi comporta che, se il servizio A chiama il servizio B, abbiamo quello che viene descritto come “attraversare un abisso”, e si possono causare fallimenti o congestione della rete.

! Se cadono tutti i servizi a catena fino all'API, è un grosso problema.

Che cosa ha fatto Netflix? Ha sviluppato **Hystrix**: permette di introdurre tecniche come Timeouts e Retries, Fallback (il cliente può continuare il servizio) e Pool, oltre a circuiti di thread isolati.

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

In generale, dobbiamo testare una visione complessiva del sistema. Abbiamo 2 livelli di testing: il livello con critical microservices, invece di testare tutti i microservizi, supervisiona solo quelli che mi servono.

11.2.3. OBIETTIVI

LIBRERIE CLIENTI: se usiamo troppe librerie clienti rischiamo di rendere l'architettura un monolite.

- **SOLUZIONE:** cercare di tenere le librerie clienti solo se sono davvero utili, ma con moderazione e mantenendole molto semplici.

PERSISTENZA:

MOLTO IMPORTANTE

CAP THEOREM

In presence of a network **Partition**, you cannot have both **Availability** and **Consistency**.

In un sistema informatico non possiamo garantire contemporaneamente:

1. **Coerenza/Consistenza**: tutti i nodi vedano gli stessi dati nello stesso momento.
2. **Disponibilità/Availability**: ogni richiesta riceve una risposta su ciò che è riuscito o fallito.
3. **Tolleranza al partizionamento**: il sistema continua a funzionare nonostante arbitrarie perdite di messaggi o malfunzionamenti.

ESEMPIO: abbiamo 3 istanze di un database B, C e D. Possono arrivare 3 richieste e la richiesta su C fallisce → C non viene aggiornato.

Chi usa B e D deve continuare a non avere problemi, nonostante C non sia aggiornato: dobbiamo rinunciare alla consistenza.

Se vogliamo la consistenza, dobbiamo fare un rollback e non aggiornare né B, né D, né C.

Cosa ha fatto Netflix? Ha scelto la strada dell'**eventual consistency** (con Cassandra): scrivi dove puoi scrivere, prima o poi dovrà aggiornare anche i database che non sono stati aggiornati.

- ! Viene data priorità all'availability.
- ! C'è un quorum minimo: almeno un tot di database deve essere aggiornato.

INFRASTRUTTURE: per evitare che Netflix non sia disponibile a causa di down dei server di Amazon (24 dicembre 2012, avevano una sola regione di server), usa una strategia multiregioni su più regioni di Amazon.

SCALARE SERVIZI SENZA STATO: un servizio senza stato è un tipo di comunicazione in cui le richieste tra client e server sono indipendenti e non memorizzano uno “stato”.

- ! La perdita di un nodo è un “non evento”.

Per i servizi che non portano a uno stato la strategia di Netflix è replicare le istanze e poi testare con Chaos Monkey.

SCALARE SERVIZI CON STATO: quando deve essere fatta la scrittura di un dato, viene fatta in zone multiple. La read cerca di leggere dalla zona più vicina il dato, ma nel caso di fallimento la lettura viene ripetuta sulle copie, anche se si trovano in zone di availability diverse.

- ! Se c’è un disastro bisogna recuperare i dati da un clone.

11.2.4. ALCUNI DATI

- **30 milioni di richieste al secondo.**
- **2 trilioni di richieste al giorno.**
- Tantissimi oggetti e istanze.
- **Latenza nell’ordine dei millisecondi.**

11.3. COMPARETHEMARKET.COM

Permette di comparare i prezzi delle assicurazioni. Utilizza i microservizi perché ha bisogno di rimanere aggiornato in tempo reale con i cambiamenti dei prezzi e perché ha circa 10 milioni di visite mensili.

11.4. UBER

78 milioni di visite al mese e tanti servizi diversi (Uber Eats, Uber Bike...).

11.5. COLLEGAMENTO TRA MICROSERVIZI E CLOUD

Potrei avere la mia applicazione a microservizi e non usare il cloud, ma molto spesso i microservizi vengono deployati su Cloud.

PERCHE’?

Independently scalable services + On-demand, pay-per-use infrastructure → **Cost optimization**.

Stack proliferation + Stack Cloud services → **Minimised management challenges**.

1. I microservizi mi permettono di avere parte della mia applicazione indipendente e scalabile: se posso usare il Cloud in modalità on-demand e pay-per-use, ottengo un’ottimizzazione dei costi. La scalabilità è teoricamente illimitata con un pagamento proporzionale all’uso.
2. Il poliglottismo porta alla proliferazione degli stack: posso rendere più efficiente e flessibile la mia applicazione; tuttavia, devo gestire molteplici stack.
 - a. Se posso usare servizi Cloud che si occupano autonomamente di gestire gli stack allora questo mi porta molti benefici.

12. KUBERNETES

Prima abbiamo visto cosa sono e come sono fatti i Container. Ora dobbiamo però porci delle domande:

1. Cosa succede se un Container muore?
2. Cosa succede se la macchina su cui è in esecuzione il tuo Container fallisce?
3. Cosa facciamo se più Container hanno bisogno di comunicare tra di loro? Come stabiliamo una connessione tra di loro?
4. Se il tuo ambiente di produzione consiste in più di una macchina, come decidiamo quale macchina deve eseguire il Container?

12.1. ORCHESTRAZIONE DEI CONTAINER

Introduciamo quindi una nuova tecnica di orchestrazione dei Container (la prima l'abbiamo già vista, ovvero Docker Swarm) chiamata **Kubernetes**.

Possiede molte funzionalità simili a Docker swarm, ma è più efficiente: i due si integrano a vicenda. K8s gestisce l'intero ciclo di vita dei singoli Container, avviando e arrestando le risorse in base alle necessità.

- Se un Container si arresta inaspettatamente, K8s reagisce lanciando un altro Container al suo posto.

K8s fornisce un meccanismo per consentire alle applicazioni di comunicare tra di loro anche se i singoli Container sottostanti vengono creati o distrutti.

Dato un set di carichi di lavoro dei Container e un set di macchine su un cluster, K8s analizza ogni Container e determina la macchina migliore per schedulare quel carico di lavoro.

12.2. K8S DESIGN PRINCIPLES

12.2.1. DICHIARATIVITÀ

Permettiamo all'utente di dire **cosa** vuole e non come vuole che sia fatto. Noi utenti, quindi, definiamo quale è lo **stato desiderato** del sistema. Kubernetes poi:

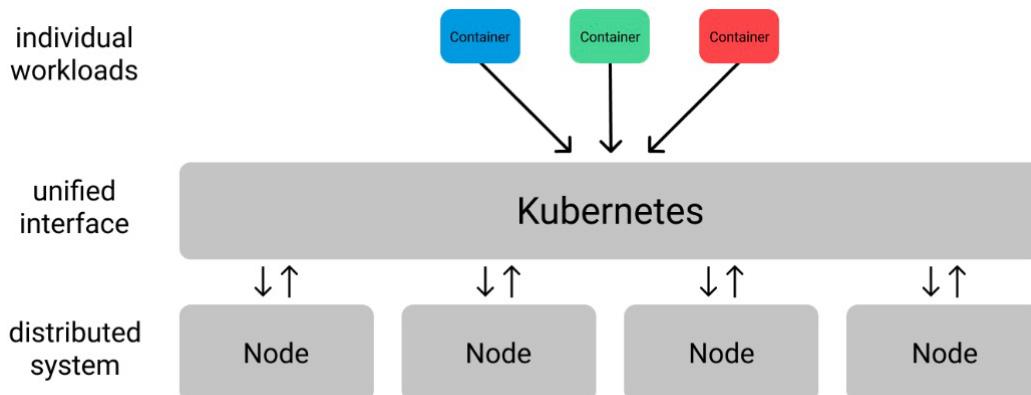
1. Monitora costantemente lo stato del sistema e capisce rileva quando questo non rispetta le nostre aspettative.
2. Interviene per aggiustare il problema, rendendo il nostro sistema auto-curante.

Lo stato desiderato è definito tramite una serie di oggetti.

- Ogni oggetto ha una specifica in cui fornisci lo stato desiderato e uno status che riflette lo stato attuale dell'oggetto.
- K8s esegue costantemente il polling di ogni oggetto per garantire che il suo stato sia uguale alle specifiche.
 - Se un oggetto non risponde, K8s produrrà una nuova versione per sostituirlo.
 - Se lo stato di un oggetto si è allontanato dalle specifiche, K8s eseguirà i comandi necessari per riportare quell'oggetto allo stato desiderato.

12.2.2. DISTRIBUZIONE

K8s fornisce un'interfaccia unificata per interagire con un **cluster di macchine**.



Non dobbiamo preoccuparci di comunicare con ogni singola macchina separatamente.

12.2.3. DECOUPLING

Abbiamo visto come i Container dovrebbero svilupparsi basandosi su un'architettura a microservizi.

K8s supporta quindi, per sua natura, l'idea di **servizi “disaccoppiati”**, separati l'uno dall'altro, che possono scalare ed essere aggiornati indipendentemente l'uno dall'altro.

12.2.4. INFRASTRUTTURA IMMUTABILE

Il nostro obiettivo dovrebbe essere quello di disporre la cosiddetta **infrastruttura immutabile**.

- Esempio: invece di fare il login in un Container per fare l'upload di aggiornamenti, noi vogliamo costruire una nuova immagine di Container, distribuire la nuova immagine e cancellare il vecchio Container.

Durante il ciclo di vita di un progetto sarebbe meglio usare la stessa immagine del Container e modificare solamente le configurazioni esterne all'immagine del Container, ad esempio montando un file di configurazione.

I Container sono progettati per essere effimeri, pronti per essere sostituiti da un'altra istanza di Container da un momento all'altro.

Il mantenimento di un'infrastruttura immutabile semplifica il rollback delle applicazioni allo stato precedente: possiamo semplicemente aggiornare la nostra configurazione usando la precedente immagine del Container.

12.3. OGGETTI

MANIFESTI: gli oggetti di Kubernetes possono essere definiti nei manifesti (file YAML o JSON), che definiscono:

1. **API Version:** l'API dell'oggetto e la sua versione.
2. **Kind:** il tipo di oggetto che si sta creando.
3. **Metadata:** cosa identifica univocamente l'oggetto.
4. **Spec:** le specifiche dell'oggetto (stato desiderato, immagine da usare...).

12.3.1. POD

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

12.3.2. DEPLOYMENT

Incluse una collezione di Pod definiti da un template e un contatore di repliche (numero n di copie del template che vogliamo eseguire).

Il cluster proverà sempre ad avere n pod disponibili, mandando in esecuzione altri pod per sostituire quelli che eventualmente falliranno.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ml-model-serving
  labels:
    app: ml-model
spec:
  replicas: 10
  selector:
    matchLabels:
      app: ml-model
  template:
    metadata:
      labels:
        app: ml-model
    spec:
      containers:
        - name: ml-rest-server
          image: ml-serving:1.0
          ports:
            - containerPort: 80
```

How many Pods should be running?

How do we find Pods that belong to this Deployment?

What should a Pod look like?

Add a label to the Pods so our Deployment can find the Pods to manage.

What containers should be running in the Pod?

12.3.3. SERVICE

Ogni pod è assegnato a un unico indirizzo IP che possiamo usare per comunicare con lui. Gli oggetti service forniscono un endpoint stabile per indirizzare il traffico ai pod desiderati anche se i pod sottostanti cambiano a causa di aggiornamenti o fallimenti.

I service sanno a quali pod devono inviare il traffico tramite delle **etichette** (coppie chiave-valore) che definiamo nei metadata dei pod.

```
apiVersion: v1
kind: Service
metadata:
  name: ml-model-svc
  labels:
    app: ml-model
spec:
  type: ClusterIP
  selector:
    app: ml-model
  ports:
    - protocol: TCP
      port: 80
```

How do we want to expose our endpoint?

How do we find Pods to direct traffic to?

How will clients talk to our Service?

12.3.4. INGRESS

Gli oggetti service ci permettono di esporre le applicazioni solo dietro un endpoint stabile e disponibile solo dal traffico interno al cluster.

Per esporre all'esterno del cluster la nostra applicazione, dobbiamo definire un oggetto **Ingress**.

- Possiamo selezionare quali service rendere disponibili al pubblico.

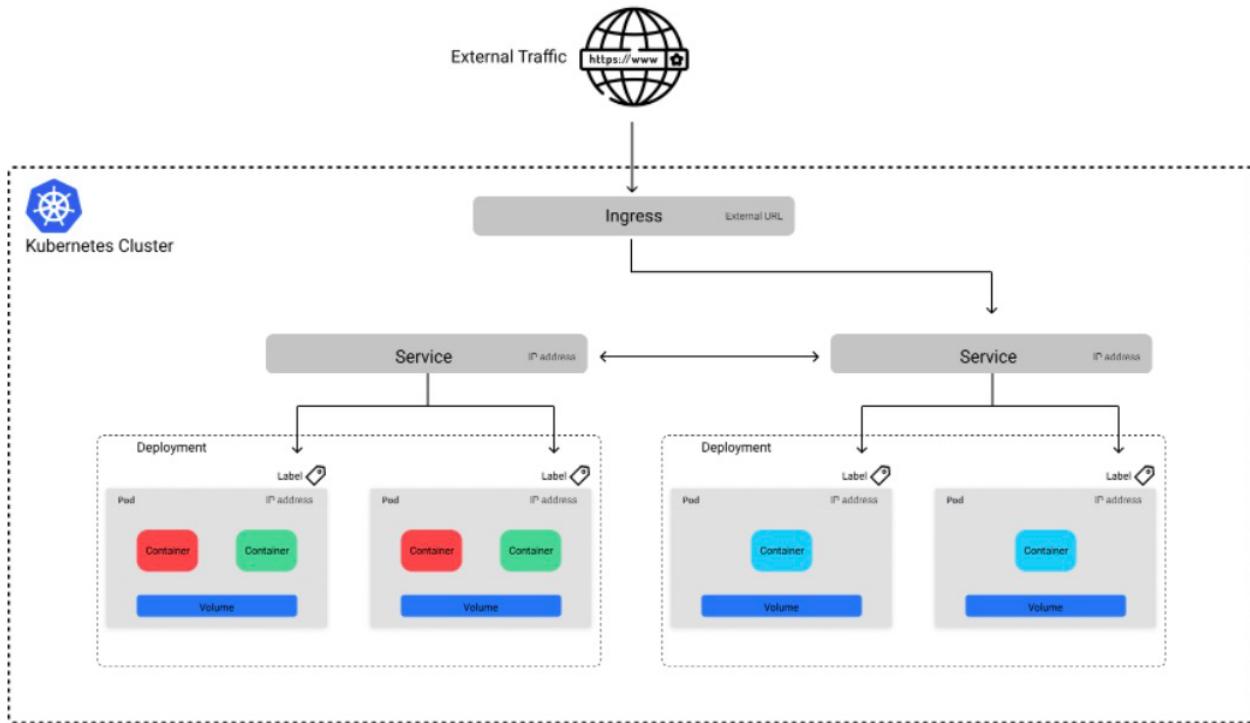
```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ml-product-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /app
        backend:
          serviceName: user-interface-svc
          servicePort: 80
```

Configure options for the Ingress controller.

How should external traffic access the service?

What Service should we direct traffic to?

Di seguito abbiamo un'immagine completa della struttura di un cluster di Kubernetes.



12.4. CONTROL PLANE

Abbiamo due tipi di macchine fisiche in un cluster:

- Nodo master:** contiene la maggior parte delle componenti del control plane.
- Nodo worker:** eseguono il carico di lavoro delle applicazioni.

12.4.1. NODO MASTER

L'utente fornisce oggetti nuovi o aggiornati all'**API server** del nodo master:

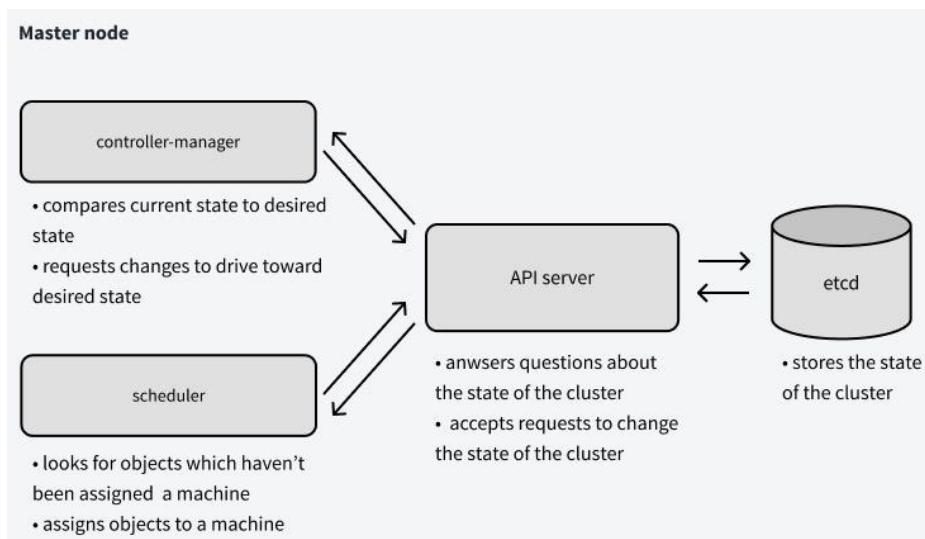
1. L'API server valida la richiesta che gli viene data e fa da interfaccia per le domande relative allo stato corrente del cluster.
2. Lo stato del cluster è memorizzato in uno store (con coppie chiave-valore) chiamato **etcd**.

Lo **scheduler** determina dove dovrebbe essere eseguito l'oggetto.

1. Chiede all'API server quali oggetti non sono ancora stati schedulati.
2. Determina a quali macchine quegli oggetti dovrebbero essere assegnati.
3. Risponde all'API server.

Il **controller manager** monitora lo stato del cluster tramite l'API server.

Se lo stato attuale è diverso dallo stato desiderato, il controller manager farà le sue modifiche tramite l'API server.



12.4.2. NODI WORKER

KUBELET: *node agent* che è in esecuzione su ogni nodo che prende un insieme di specifiche pod e assicura che i Container descritti in tali specifiche siano in esecuzione e "sani", comunicando con l'API server. Inoltre, quando un nodo si unisce al cluster, kubelet manderà un "annuncio" all'API server.

- **Kube proxy**: abilita la comunicazione tra i vari Container tramite i vari nodi del cluster.



DOCKER SWARM	KUBERNETES
<ul style="list-style-type: none"> • Più semplice da installare. • Abbastanza semplice da imparare. 	<ul style="list-style-type: none"> • Offre il meccanismo di auto-scaling. • Più robusto in termini di tolleranza ai guasti. • Grande community. • Supportato da CNCF.
La scelta migliore se l'applicazione che si deve gestire è relativamente semplice.	La scelta migliore se l'applicazione raggiunge un livello di complessità considerevole e la dimensione dei cluster non è banale.

12.5. LABORATORIO KUBERNETES

In questo laboratorio useremo **Kubernetes** per creare un Pod contenente container che eseguono due immagini Docker.

- Prerequisito: installare i tool `minikube` e `kubectl`.

12.5.1. MINIKUBE E KUBECTL

MINIKUBE: è uno strumento che ti permette di eseguire Kubernetes in locale.

- Esegue un cluster Kubernetes all-in-one oppure multi-nodo sul tuo personal computer in modo che tu possa provare Kubernetes, oppure per il tuo lavoro di sviluppo quotidiano.

KUBECTL: è uno strumento da riga di comando che ti consente di eseguire comandi sui cluster Kubernetes.

- Si può usare per distribuire applicazioni, ispezionare e gestire le risorse del cluster e visualizzare i log.

12.5.2. STEPS

1. Creare un **cluster** in locale con Minikube: scaricare quindi Minikube e farlo partire con il comando

```
minikube start
```

2. Partendo dalle due immagini

- a. <https://hub.docker.com/r/yeasy/simple-web> (in ascolto sulla porta 80)
- b. <https://hub.docker.com/r/scottyc/webapp> (in ascolto sulla porta 3000)

Scrivere

- a. Un file `pod.yaml` per configurare il pod che gestisce le due immagini.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod1
  labels:
    app: myApp      # etichetta necessaria per ricollegarci al
                     # Service
spec:
  containers:
    - name: cont1
      image: scottyc/webapp
      ports:
        - containerPort: 3000
```

```

- name: cont2
  image: yeasy/simple-web
  ports:
    - containerPort: 80

```

- b. Un file `service.yaml` per esporre il pod come servizio raggiungibile.

```

apiVersion: v1
kind: Service
metadata:
  name: myservice1
spec:
  type: LoadBalancer
  selector:
    app: myApp
  ports:
    - protocol: TCP
      name: scotty-web-app
      port: 5000 # porta che useremo noi in localhost
      targetPort: 3000 # porta del Container

    - protocol: TCP
      name: yeasy-web-app
      port: 5002 # porta che useremo noi in localhost
      targetPort: 80 # porta del Container

```

3. Dispiegare con `kubectl` il **pod** e il **servizio**.

```
--pod--
kubectl create -f PATH/pod.yaml
kubectl apply -f PATH/pod.yaml
```

```
--service--
kubectl create -f PATH/service.yaml
kubectl apply -f PATH/service.yaml
```

4. Testare il risultato. Per farlo, abbiamo bisogno di esporre il nostro servizio, e lo facciamo tramite il comando

```
minikube tunnel
```

L'altro metodo (non affrontato al laboratorio) è utilizzando ingress, un ulteriore involucro che fa da gateway. Per essere utilizzato, però, ha bisogno di un ingress control.

13. DATACENTER

Strutture complesse i cui server rappresentano solo una piccola parte. Ci sono generatori di energia, batterie, unità di raffreddamento, connessioni Internet, sezioni per emergenze, telecamere di sicurezza, porte di sicurezza, backup dei dati in un'altra sede.

13.1. SAP DATACENTER

Si preoccupano anche della salvaguardia dell'ambiente: strategie per ridurre l'inquinamento.

13.2. GOOGLE DATACENTER

Sicurezza: scanner dell'iride o laser sotto i pavimenti. Energia distribuita in modo "overhead" (dall'alto). Tecnologia di raffreddamento con torri e tubi in cui scorre l'acqua: tiene la temperatura più alta rispetto alle altre aziende. Quando cambiano i drive, dopo averli sostituiti, li distruggono. Cerca di ridurre l'emissione di CO₂.

13.3. UNIPI DATACENTER

Ci sono 3 nodi DCI (*DataCenter Interconnect*) in città a 100Gbit/sec. La connessione è *no single point of failure*.

Traffico est-ovest: traffico di dati all'interno della rete di datacenter, vi è un *spine-leaf topology* che contribuisce all'aspetto di ridondanza.

Traffico nord-sud: traffico con l'esterno.

Sistema di raffreddamento **adiabatico**: permette di avere un indice PUE < 1.3.

- Raffreddamento adiabatico: mettiamo insieme il raffreddamento evaporativo con il raffreddamento dell'aria. In particolare, usiamo l'evaporazione dell'acqua per pre-raffreddare l'aria. Quindi, usiamo il raffreddamento evaporativo solo nelle parti più calde del giorno e dell'anno.

PUE: indice per l'efficienza energetica, superiore a 1 ma il più vicino possibile a 1, si calcola come

$$\frac{\text{total facility power}}{\text{IT equipment power}}$$

13.4. DATACENTER INFRASTRUCTURE MANAGEMENT

Azioni principali: planning, installazione e gestione dei server e delle connessioni, mantenere il cablaggio pulito, sicurezza.

Per monitorare i datacenter si ha una **visione grafica (DCIM)**: grafici e reports con notifiche immediate in caso di errori o fallimenti.

- **Documentazione:** se la documentazione non viene svolta in modo regolare, può portare a vari problemi perché non si conoscono le informazioni né storiche né aggiornate su qual è lo stato dell'infrastruttura.

13.4.1. COSA NON FARE ALL'INTERNO DI UN DATACENTER

1. Gestione disordinata del cabling.
2. Introdurre cibo o bevande all'interno del datacenter.
3. Tralasciare la documentazione.

4. Lasciare la porta aperta del datacenter tra un accesso e l'altro.
5. Non prevenire click accidentali di interruttori.

13.4.2. COME REAGISCE IL PERSONALE DI FRONTE A UN PROBLEMA

1. Rendersi pronto immediatamente.
2. Gestire il panico.
3. Seguire la **checklist**: serve sia all'operatore che al gestore, altrimenti ognuno si comporta in maniera diversa.

13.4.3. BUSINESS CONTINUITY & DISASTER RECOVERY

1. Avere una copia degli stati *lontana* dall'altra.
2. Il piano di evacuazione/incendio va prima testato.
3. Avere personale preparato.
4. Avere piani di recupero nel caso il primo piano fallisca.

13.5. HYPERCONVERGENCE

Struttura tradizionale: server, networking e storage separati tra loro e poi montati assieme.

- Può causare l'incompatibilità tra le componenti.

Struttura converged: fornisce subito un insieme di componenti pretestate e validate con un sistema di gestione integrato con cui poter eseguire le operazioni principali.

Struttura hyper-converged: sfrutta la virtualizzazione → Combina la virtualizzazione del server con il networking e lo storage in un singolo box.

- Riduce di molto i costi hardware.
- Ha alcune limitazioni: non si può aumentare lo storage senza aumentare anche la potenza di calcolo, non sono supportate tutte le applicazioni, i relativi software non sono economici.

14. INTERNET OF THINGS & FOG COMPUTING (DAL CLOUD ALL'IOT)

14.1. INTERNET OF THINGS

Area in cui si studia come dispositivi elettronici, se dotati di connettività di rete, creano una serie di applicazioni innovative.

- Circa 10 miliardi di dispositivi IoT attivi nel 2021, in aumento di più di 25.4 miliardi nel 2030.

Producono una quantità di dati enorme che, grazie al Cloud, è possibile raccogliere.

PROBLEMA: per alcune applicazioni questo tipo di architettura non è efficace, e c'è bisogno di processare e filtrare i dati prima di salvarli nel Cloud.

14.2. DEPLOYMENT MODELS

IoT + Edge: elaborazione locale dei dati, al confine tra il Cloud e i dispositivi locali.

- **Vantaggio:** elaborare i dati direttamente nello stesso luogo dove vengono prodotti (bassa latenza).
- **Svantaggio:** capacità limitata e difficoltà nella condivisione dei dati tra applicazioni.

IoT + Cloud: i dati vengono mandati sul Cloud per l'elaborazione dove vi è una potenza di calcolo enorme.

- **Vantaggio:** grande potenza di calcolo.
- **Svantaggi:** le parti dell'applicazione devono essere collegate per poter raggiungere la rete. Alta latenza e bottleneck per l'ampiezza di banda.

14.3. FOG COMPUTING

Infrastruttura di nodi eterogenei (piccoli datacenter, server, laptop...) distribuiti in mezzo tra IoT e Cloud per migliorare la capacità di calcolo e di storage.

Il fog 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.

ADAPTIVE APPLICATION DEPLOYMENT: mandare in esecuzione un'applicazione multiservizio in una infrastruttura grande, distribuita ed eterogenea non è semplice se si vuole tenere conto dei vincoli.

APPLICATION MANAGEMENT: gestione delle applicazioni durante il ciclo di vita del sistema.

LIGHTWEIGHT MONITORING: monitoraggio distribuito su infrastrutture offerte da provider diversi.

PRIVACY/SECURITY/TRUST: se le infrastrutture sono offerte da provider diversi ci possono essere problemi di sicurezza e privacy.

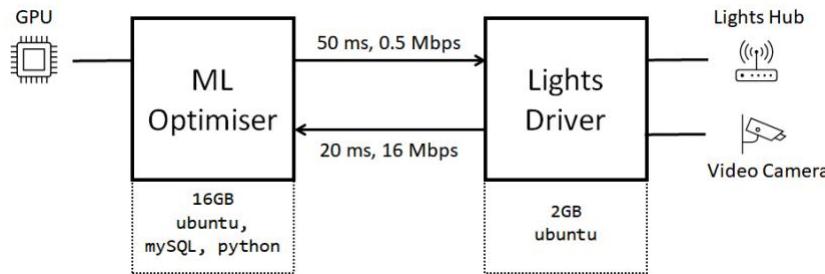
14.3.1. DEPLOYING APPLICATIONS THROUGH THE FOG

Le applicazioni di nuova generazione sono tipicamente applicazioni multiservizio, composte da più pezzi che devono essere mandati in esecuzione sui nodi di una infrastruttura eterogenea, enorme e dinamica: ci sono dei nodi che possono entrare e uscire liberamente.

Vediamo un esempio di applicazione.

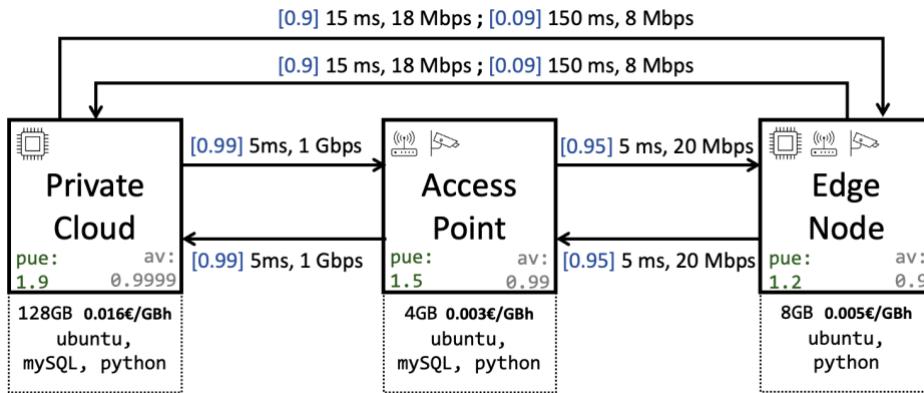
SMART LIGHTING: applicazione che gestisce le luci pubbliche in una strada sfruttando tecniche di Machine Learning per ridurre il consumo di energia.

Abbiamo due servizi che gestiscono l'accensione dei dispositivi e l'ottimizzazione della GPU. Questi due moduli devono poter comunicare, oltre che con i dispositivi "esterni" ai quali sono collegati, anche tra di loro con un tempo e una banda ragionevoli.



Abbiamo poi un'infrastruttura Fog, composta da 3 nodi (cloud privato, access point e nodo edge).

- I numeri tra parentesi quadre ci dicono la probabilità con cui si avranno quelle statistiche di trasferimento dei dati.



Nella nostra infrastruttura aggiungiamo i nostri stakeholder, che saranno responsabili del corretto funzionamento delle varie parti dell'applicazione. Tra di loro ci saranno dei legami di fiducia.

Sorgono alcuni problemi:

- PLACEMENT:** come si fa a posizionare i servizi di un'applicazione in maniera opportuna su una infrastruttura, tenendo conto di un insieme di parametri. È un problema NP-hard.
 - Declarative placement:** linguaggio dichiarativo, basato su logica del primo ordine. Il linguaggio si chiama **prolog** e la notazione `: -` significa se. Con questo linguaggio possiamo scrivere i nostri programmi in termini di fatti e relazioni tra questi fatti per posizionare i servizi dell'infrastruttura tenendo conto di costi, qualità del servizio, consumo di risorse e sicurezza.
- DINAMICITA' 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 lo stesso.
 - Per tener conto di questa variazione si possono usare dei **modelli probabilistici** utilizzando delle distribuzioni di probabilità.
 - Vantaggio:** possiamo stimare quanto è buono il nostro piazzamento del servizio rispetto a variazioni infrastrutturali.

- c. **Svantaggio:** il backtracking è in genere esponenziale.
- 3. **CONTINUOUS REASONING:** quando si hanno sistemi di grandi dimensioni si utilizza la proprietà di *composizionalità* per analizzare in maniera differenziale i sistemi → Ci si focalizza solo sulle ultime cose che sono cambiate.
 - a. I programmati di Facebook, dopo aver aggiornato un servizio, utilizzano un motore chiamato **infer** che svolge un'analisi statica per controllare se l'intero sistema è *bug-free* e rispetta le specifiche.
- 4. **TRUST:** l'infrastruttura continua tra IoT e Cloud non è di un solo proprietario, ma ci sono diversi provider → Il deployment di un'applicazione in generale può richiedere che parti dell'applicazione vadano in nodi operati da provider diversi. È importante riuscire a stabilire delle relazioni di confidenza tra i provider.

14.4. MONITORING THE FOG

I meccanismi di monitoraggio generano una degradazione della rete. L'idea è di avere un meccanismo di monitoring distribuito, leggero, resistente ai malfunzionamenti e che esegua un monitoraggio:

1. Delle risorse hardware disponibili.
2. Della qualità del servizio end-to-end tra i nodi.
3. Dei dispositivi IoT collegati.

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

15. SEMINARIO EXTRARED: HIGH TABLE

High Table è un progetto di trasformazione digitale e di migrazione al Cloud: si prende una o più vecchie applicazioni, e si effettua una “modernizzazione” di queste portandole in un Cloud.

15.1. PRESENTAZIONE EXTRARED

15.1.1. NUMERI DI EXTRARED

- 2 sedi in Italia.
- Oltre 50 clienti.
- 5 milioni di fatturato.
- Più di 75 dipendenti: 4 CXO, 6 Project Manager, 8 Sales & Pre-Sales, 60+ Delivery Teams.

15.1.2. TECHNOLOGY SERVICE PROVIDER

Servizi ad alto valore aggiunto con tecnologie leader.

- **Red Hat – OpenShift:** l’azienda leader nel mondo dell’Open Source.
 - DevOps.
 - Cloud & Infrastructure.
 - Middleware.
 - AMM.
- **IBM:** Public & Hybrid Cloud, Cloud Paks, Watson, Augmented Intelligence.
 - AI & Data.
 - Cloud.
- **Influxdata:** piattaforma DB Time Series Open Source per Metriche ed Eventi.
 - Monitoraggio & Assistenza applicativa/infrastrutturale.
 - IoT & Industria 4.0.
- **Liferay:** Digital Experience Platform.
 - DXP (Digital Experience Platform) & App Mobile.
 - Enterprise custom software provider.

15.1.3. LINEE DI BUSINESS

1. **AI & Data.**
2. **Middleware:** creare architetture e gestire scambi di messaggi tra le applicazioni.
3. **Platform (Cloud & DevOps).**
4. **Digital Experience:** team che si occupa di realizzare applicazioni di frontend e portali.

15.2. HIGH TABLE: IL CLIENTE DI EXTRARED E I SUOI REQUISITI

15.2.1. IL CLIENTE: L’ENTE PUBBLICO HIGH TABLE

È un ente che fornisce servizi a una categoria di Liberi Professionisti.

- Gli assistiti di High Table generano in Italia un giro d'affari pari a circa 220 miliardi di euro all'anno (11% del PIL).
- Hanno circa 200k iscritti, che effettuano circa 3 milioni di interventi sul territorio all'anno.

Servizi offerti dall'ente:

1. Gestione di contratti.

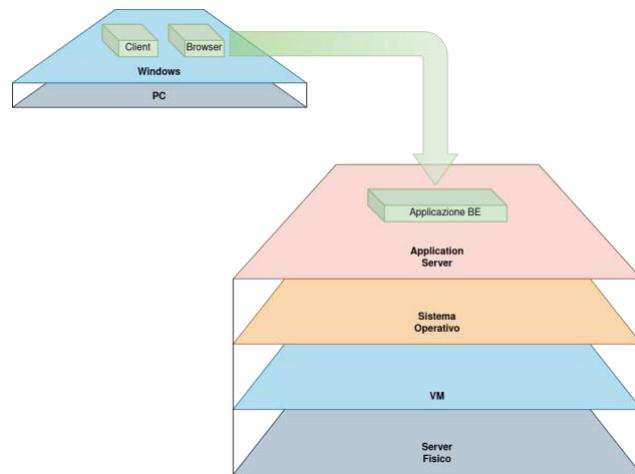
2. Fornitura strumentazione professionale.
3. Trasporti.
4. Ospitalità.
5. Cure mediche.

15.2.2. INFRASTRUTTURA INFORMATICA DELL'ENTE

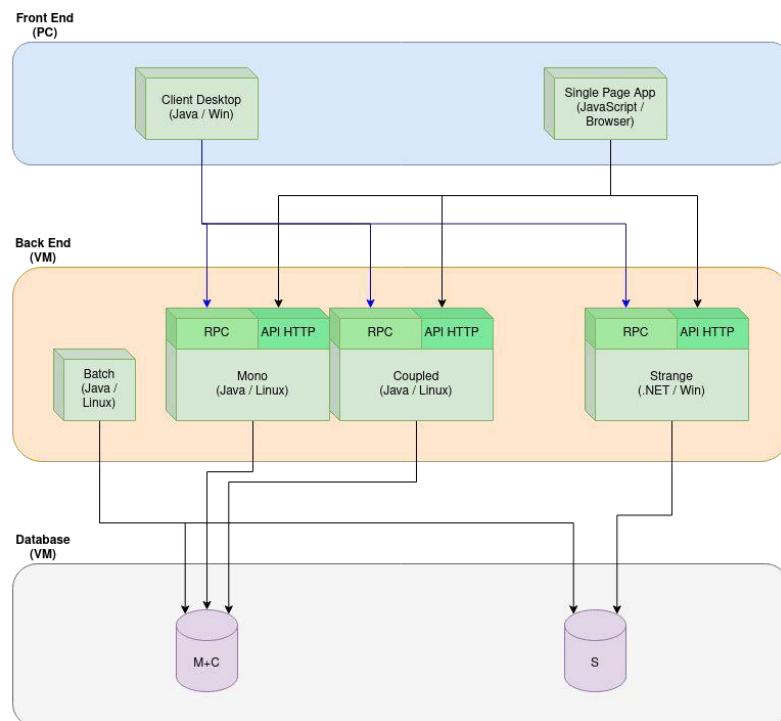
Tutte le applicazioni girano su VM e Application Server, in un CED.

- Sistema di gestione dei rapporti “M” – **Monolitico** → Monolite tradizionale, iniziato nei primi anni 2000.
- Sistema di gestione pagamenti e contratti “A” – **Accoppiato** → Altro monolite, che condivide il database con l’applicazione precedente.
- Sistema di gestione delle Informazioni Riservate “S” – **Strano** → Monolite realizzato con una tecnologia fuori dalla “comfort zone” di ExtraRed, non realizzato in Java.

APPLICAZIONE DI TIPO HIGH TABLE



ARCHITETTURA DELLE APPLICAZIONI HIGH TABLE



15.2.3. COSA VUOLE IL CLIENTE

AMM (IN CLOUD): vuole cambiare l'**ambiente** nel quale girano le applicazioni, portando queste ultime da CED a Cloud.

- Il cliente sa già che non è sufficiente un approccio “*lift & shift*”, le applicazioni andranno **modificate** (refactoring) e aggiornate.
- Il cliente non vuole solamente ammodernare l’architettura e le tecnologie ma aggiungere **nuove funzionalità**.

NUOVE FUNZIONALITA’: High Table vuole implementare una “**BI in real time**”.

- Migliorare l’esperienza utente, tramite introduzione di chatbot (**NLP**) e machine learning (**ML**).
- Introdurre logiche di workflow management (orchestrazione di servizi tramite **BPM**).

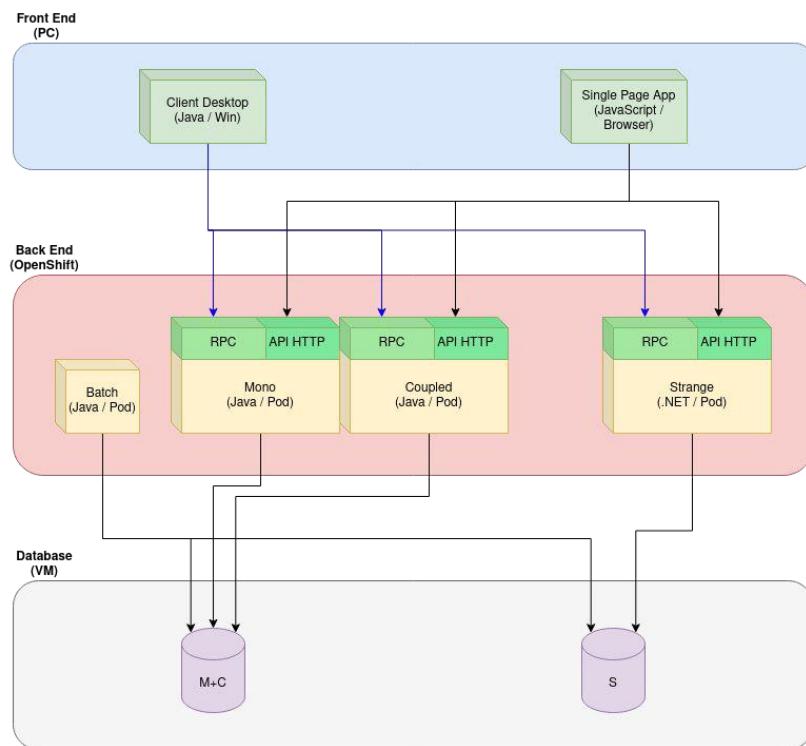
15.2.4. COSA NON VUOLE IL CLIENTE

Il cliente non vuole un approccio “Big Bang”:

- Vuole un approccio **graduale** alla trasformazione digitale (progetto in step successivi).
- Vuole mantenere un **parallelo** tra vecchi servizi e nuovi, senza disservizi.
- Vuole **riutilizzare** la logica di business già sviluppata.

15.3. IL PROCESSO DI TRASFORMAZIONE

15.3.1. DIGITAL TRASFORMATION – PASSO 1

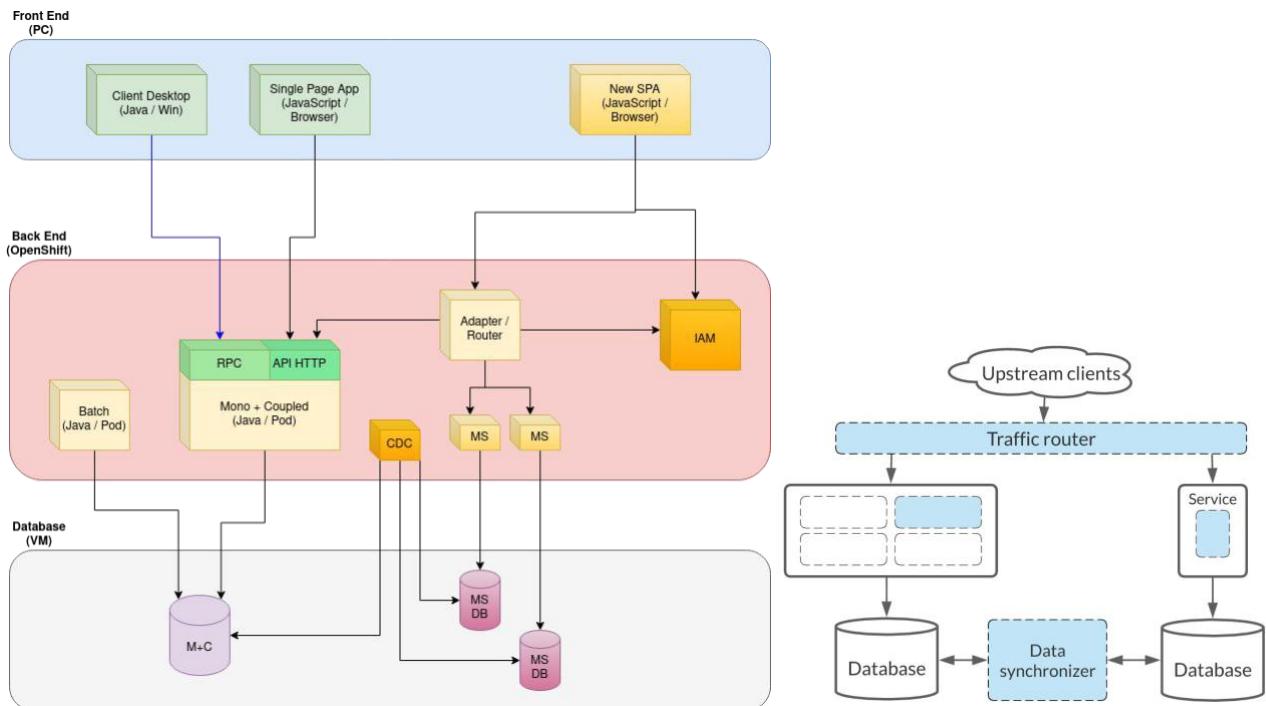


- Introduzione di **OpenShift** “on premise”.
- Migrazione delle applicazioni su OpenShift tramite **lift & shift**.
- L’applicazione *Strana* si gestisce portandola su Linux piuttosto che facendo girare le applicazioni su nodi Windows.
- I database rimangono fuori (su VM) perché la gestione dello **stato** può essere un problema.
- Il FE rimane esattamente come prima.

DETTAGLI

- Migrazione su OpenShift assistita da **MTA** (Migration Tool for Applications).
- Migrazione tra Application Server: da JBoss AS Legacy a **JBoss EAP** su OpenShift (le applicazioni rimangono pacchettizzate come prima).
- Containerizzazione dell'applicazione *Strana* e dei processi batch in modo compatibile con OpenShift (queste applicazioni sono pacchettizzate in modo diverso da prima).

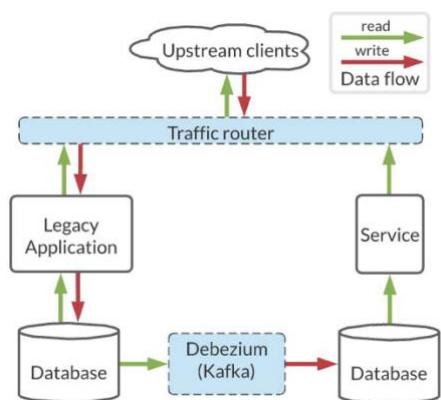
15.3.2. DIGITAL TRASFORMATION – PASSO 2



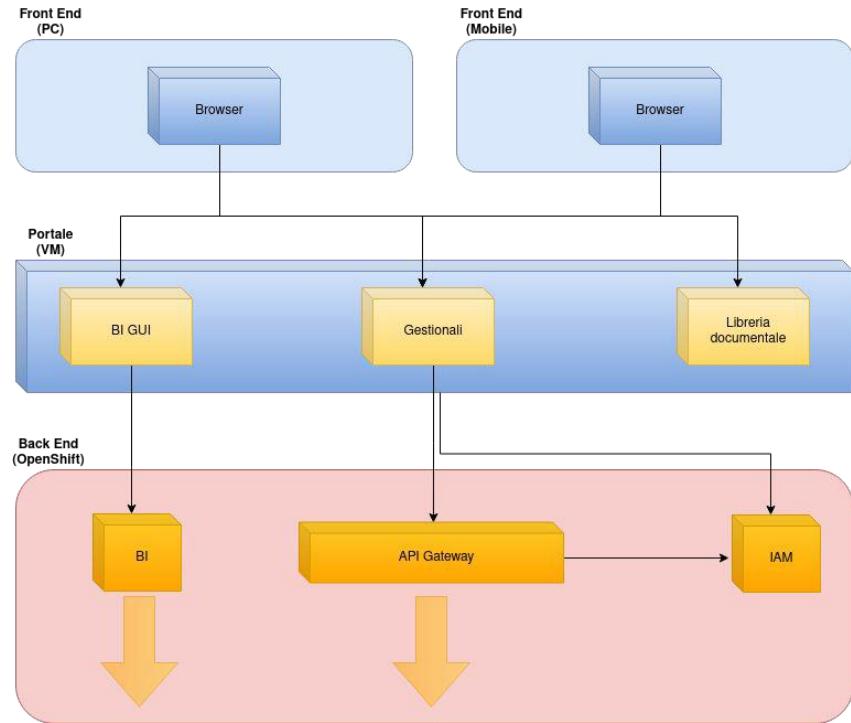
- Il vecchio FE rimane inalterato.
- Si introducono i prototipi dei nuovi FE in **parallelo**.
- Si introducono degli **Adapter/Router** davanti alle applicazioni, collegati a uno IAM.
- Si inizia ad applicare lo “*Strangler pattern*” tramite **CDC** (Change Data Capture).

DETTAGLI: “STRANGLER FIG APPLICATION”

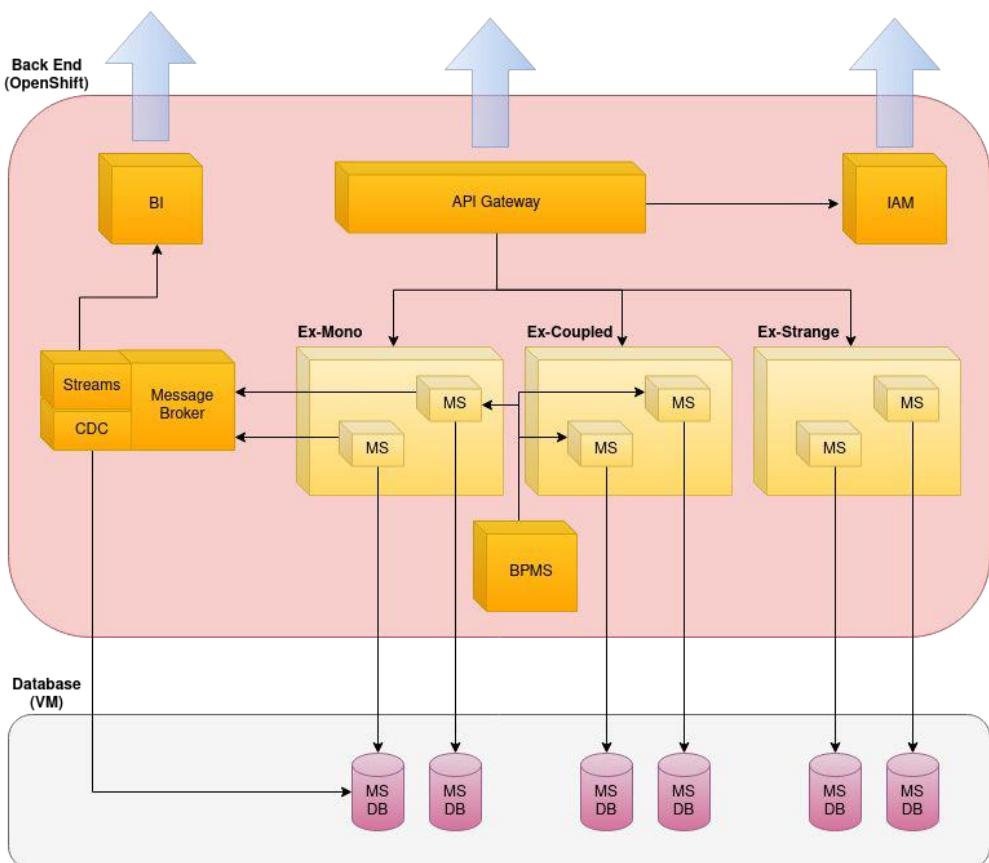
- Identificazione dei confini funzionali.
- Migrazione delle funzionalità a un nuovo (micro)servizio.
- Migrazione del database (database strangulation).



15.3.3. DIGITAL TRASFORMATION – PASSO 3

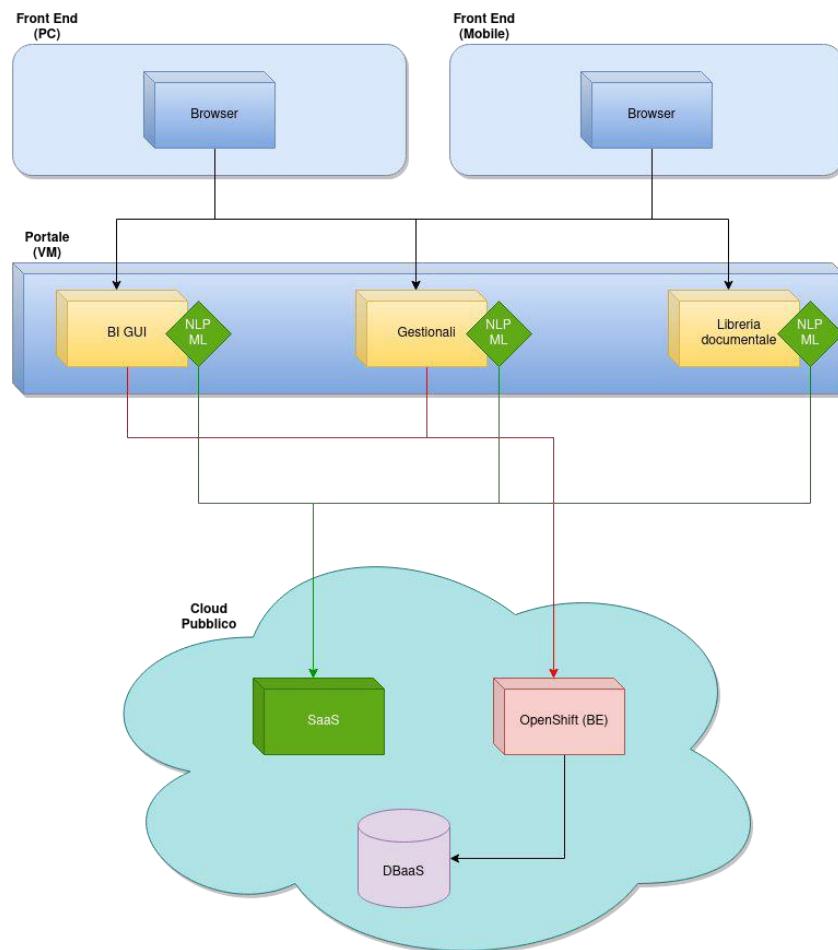


- Le applicazioni **desktop** sono state rimosse (solo applicazioni web).
- Le nuove applicazioni web di FE sono state accorpate all'interno di un **Portale**, che gira su un cluster di VM dedicato.
- Il Portale è stato collegato allo IAM introdotto al passo precedente, per gestire sia SSO che chiamate API.



- È stato introdotto un **API Gateway** davanti ai servizi, collegato allo IAM (Adapter non più necessari).
- Tra FE e BE vengono scambiati dei token di autenticazione e autorizzazione.
- Le applicazioni “M” ed “A” sono state disaccoppiate. Tutte le app sono state reindustrializzate a microservizi.
- Tutti i servizi espongono interfacce REST.
- I microservizi utilizzano un middleware di tipo “message broker” per scambiarsi messaggi in modo **asincrono**.
- Ciascun microservizio ha il proprio database, separato dagli altri.
- È stato introdotto un middleware di **BPM** (Business Process Modeling) per orchestrare i processi di business tra i servizi.
- Il sistema di CDC è stato conservato per alimentare le logiche di **ETL** (Extract/Transform/Load) che girano come applicazioni di stream processing.
- È stato introdotto un sistema di **BI** (Business Intelligence), alimentato dai flussi di stream processing.

15.3.4. DIGITAL TRASFORMATION – PASSO 4



- Le applicazioni di FE sono state integrate con alcuni servizi **SaaS** esterni (servizi di Natural Language Processing e Machine Learning).
- I servizi e i middleware sono stati portati su un OpenShift **managed** in cloud.
- I database sono stati migrati a dei **DBaaS** in cloud.
- Il Portale stesso può essere portato su un cluster di VM in cloud.

15.4. I TEAM E I LORO RUOLI

15.4.1. PLATFORM TEAM (OPS + DEVOPS)

- Installazione e configurazione infrastrutturale di OpenShift on premise.
- Installazione e configurazione di base dei middleware, dello IAM e della piattaforma BI.
- Installazione e configurazione di base del Portale.
- Configurazione dei servizi del cloud pubblico.
- Approntamento delle pipeline e delle automazioni DevOps.
- Migrazione in cloud.

15.4.2. MIDDLEWARE TEAM (MIDDLEWARE E MICROSERVIZI)

- Migrazione e refactoring delle applicazioni.
- Interfacciamento con i middleware (implementazione EIP).
- Sviluppo delle logiche specifiche ai middleware.
- Gestione del ciclo di vita dello sviluppo su OpenShift.

15.4.3. DXP TEAM (PORTALI E WEB UI)

- Realizzazione delle nuove applicazioni web di FE.
- Personalizzazione e configurazione avanzata del Portale.
- Integrazione delle applicazioni del Portale con i servizi ML e NLP.

15.4.4. AI&DATA TEAM

- Configurazione avanzata della piattaforma BI.
- Creazione dashboard e report.
- Sviluppo e configurazione dei servizi ML e NLP.

15.5. CONCLUSIONI

PARTENZA	ARRIVO (PER ORA)
<ol style="list-style-type: none">1. Applicazioni monolitiche e fortemente accoppiate.2. Tecnologie e modelli di deployment obsoleti.3. Interfacce non standard, sincrone.	<ol style="list-style-type: none">1. Applicazioni a microservizi disaccoppiati e autonomi.2. Framework e linguaggi moderni, container orchestrati.3. Interfacce standard sincrone e asincrone.4. Nuove funzionalità: BI in real time, BMP, NLP, ML.