



extra**red**



Dalle applicazioni tradizionali ai microservizi in cloud

---

Extra Red 2024



# Indice dei contenuti

01. L'eredità delle applicazioni tradizionali
02. Primi passi verso il cloud
03. Un approccio differente: i microservizi
04. Applicazioni cloud-native



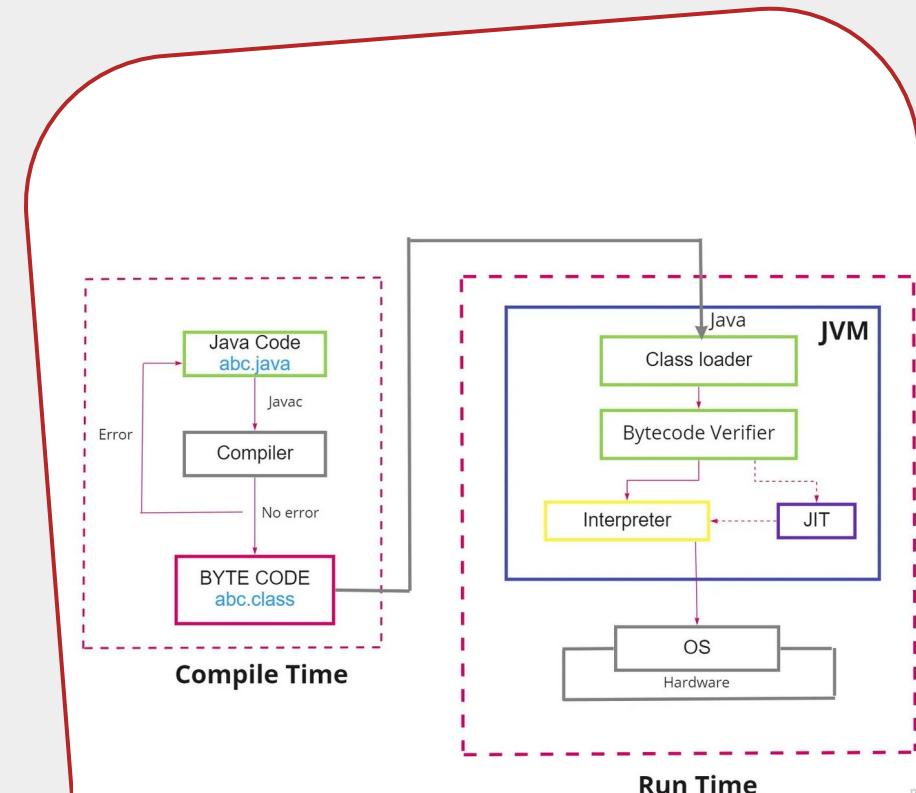
## L'eredità delle applicazioni tradizionali





# Un problema delle applicazioni tradizionali

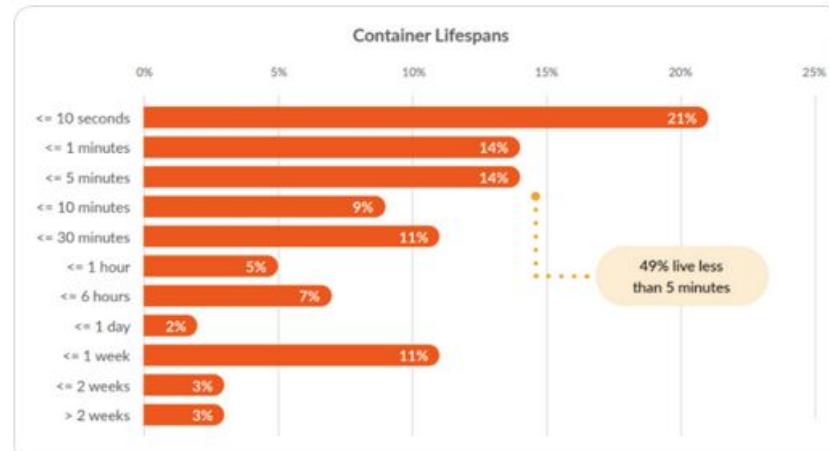
- Una macchina virtuale (Java) permette di far girare un programma compilato in byte code
- Il programma gira nella macchina virtuale, non sulla macchina fisica (non è codice **nativo**)
- La macchina virtuale esegue il codice mediante una combinazione di interprete e **JIT compiler**
- Il JIT compiler trasforma il byte code che ricorre più di frequente in codice nativo
- Un programma compilato in byte code impiega del tempo a girare in modalità nativa (**warm-up**)
- Solo un **long-running process** raggiunge una performance ottimale





# Applicazioni tradizionali in contesti dinamici

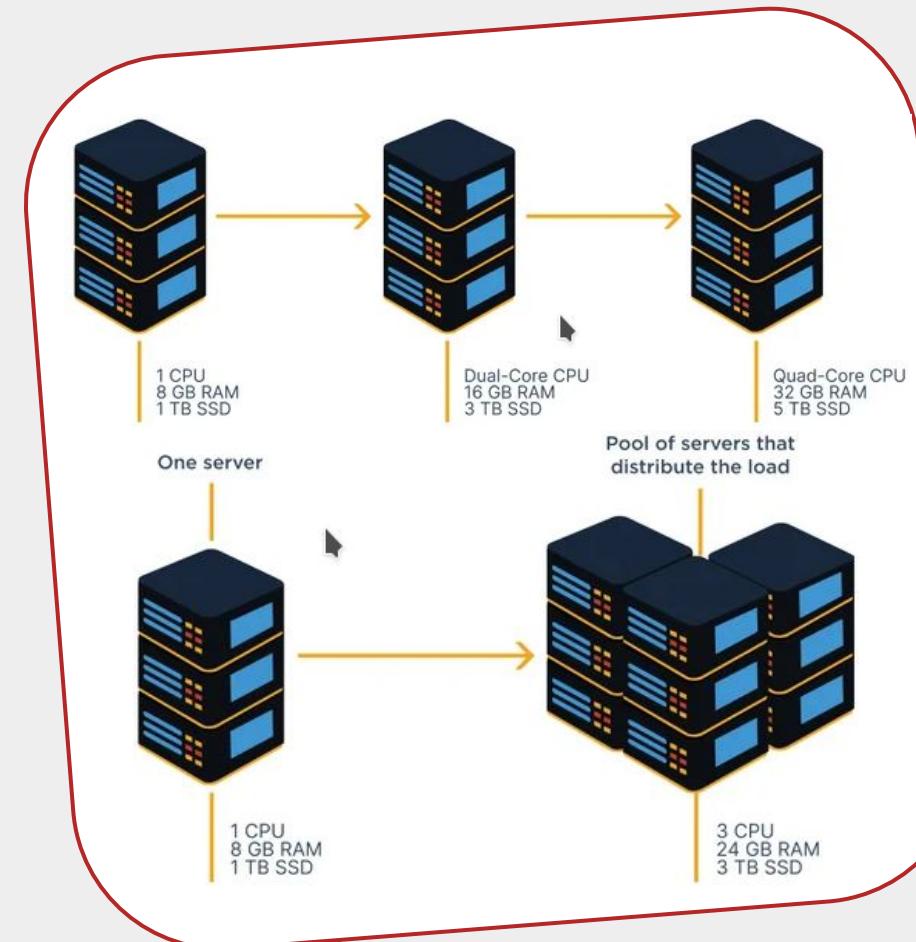
- Se secondo le statistiche di Sysdig, il 74% dei container rimane in vita per meno di 1 ora
- In una situazione del genere, le app Java (o comunque quelle basate su compilatore JIT) girano sempre in modalità interpretata o "CI"
- I linguaggi dotati di compilatore AOT (Go, Rust...) non soffrono di questo problema
- E' possibile compilare Java in modalità AOT? (Le nuove app sì!... GraalVM, Quarkus)
- Le applicazioni Java legacy tendono a girare **peggio** se vengono fatte girare in un ambiente altamente dinamico





# Modelli di scalabilità

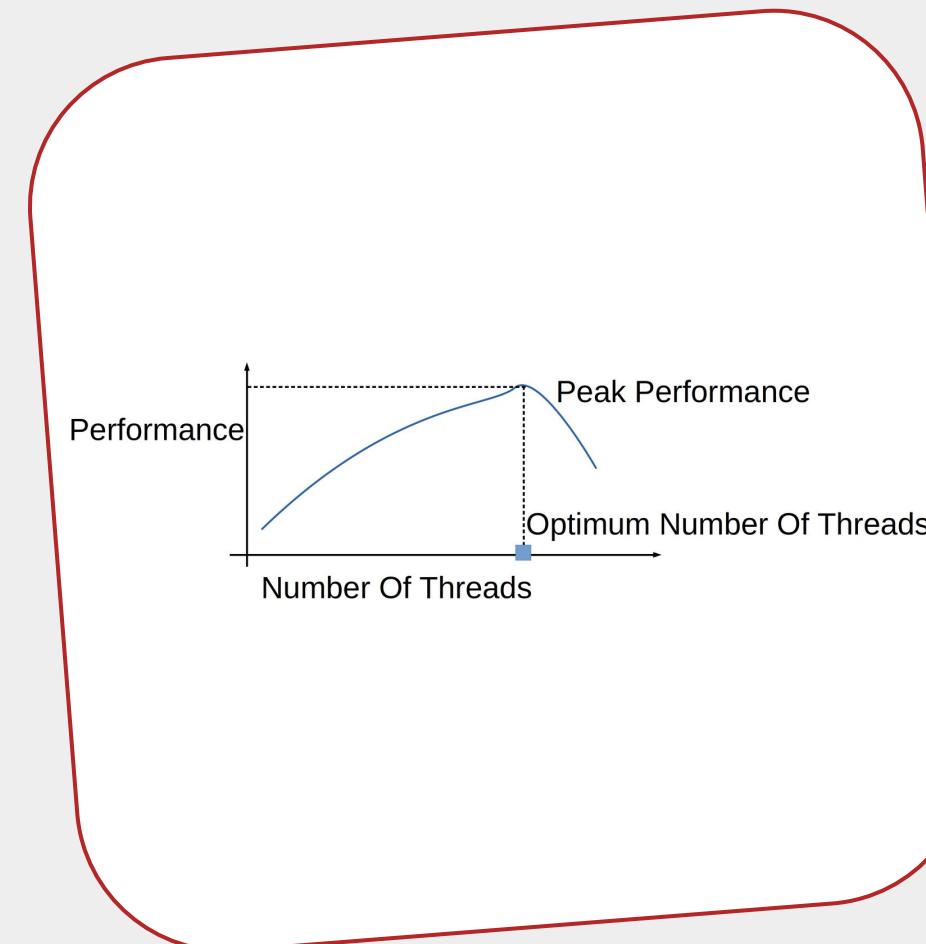
- Esistono 2 modelli di scalabilità: scalabilità verticale (ovvero scaling up/down) e scalabilità orizzontale (scaling out/in)
- La scalabilità orizzontale offre diversi vantaggi: elasticità, resilienza, fault tolerance, downtime ridotto o assente, pay per use, virtualmente illimitata
- Svantaggi: è complessa da gestire e spesso è necessario modificare il software per potersene avvalere a pieno (app stateless lato server)
- Viceversa, la scalabilità verticale è semplice ma implica downtime e non funziona bene con le applicazioni scritte in determinati linguaggi, oltre a un certo limite





# Applicazioni tradizionali e modelli di scalabilità

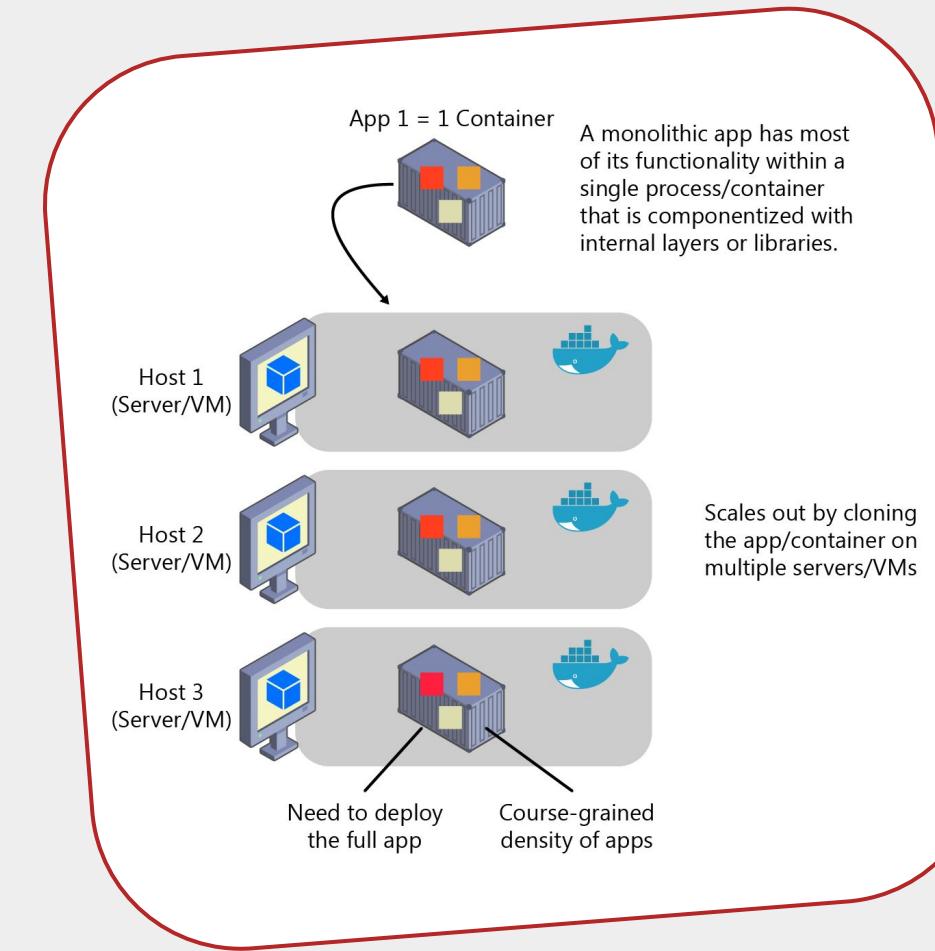
- Il modello di parallelismo offerto dal Java si basa su thread e lock: è semplice ma scala in modo limitato
- Con un numero di processori superiore a 5-6, il numero dei thread Java che competono sui lock supera una soglia critica (più CPU = più **competizione**)
- Le applicazioni, specialmente quelle complesse, non riescono a scalare linearmente oltre questa soglia
- Si ottiene un comportamento migliore con un insieme di piccole applicazioni separate e più semplici, ciascuna delle quali genera un minor carico di CPU





# Limiti delle applicazioni tradizionali

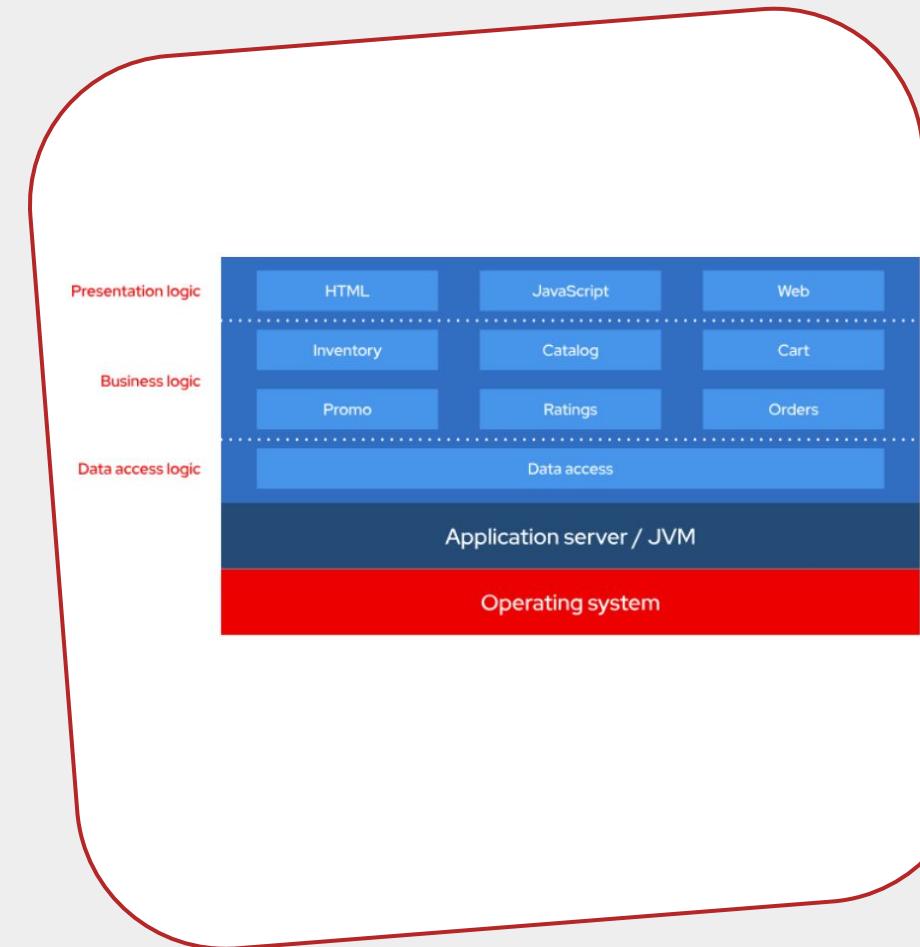
- Anche la scalabilità **orizzontale** è un problema in un contesto **dinamico**: le operazioni di scale out/scale in creano e distruggono container, comportando di conseguenza una forte dinamicità che compromette la resa di un'applicazione tradizionale (warm up!)
- Le grandi applicazioni legacy tipicamente **non** sono scalabili in un contesto dinamico (né verticalmente né orizzontalmente)
- Tali applicazioni non sfruttano i vantaggi di una container platform e del cloud





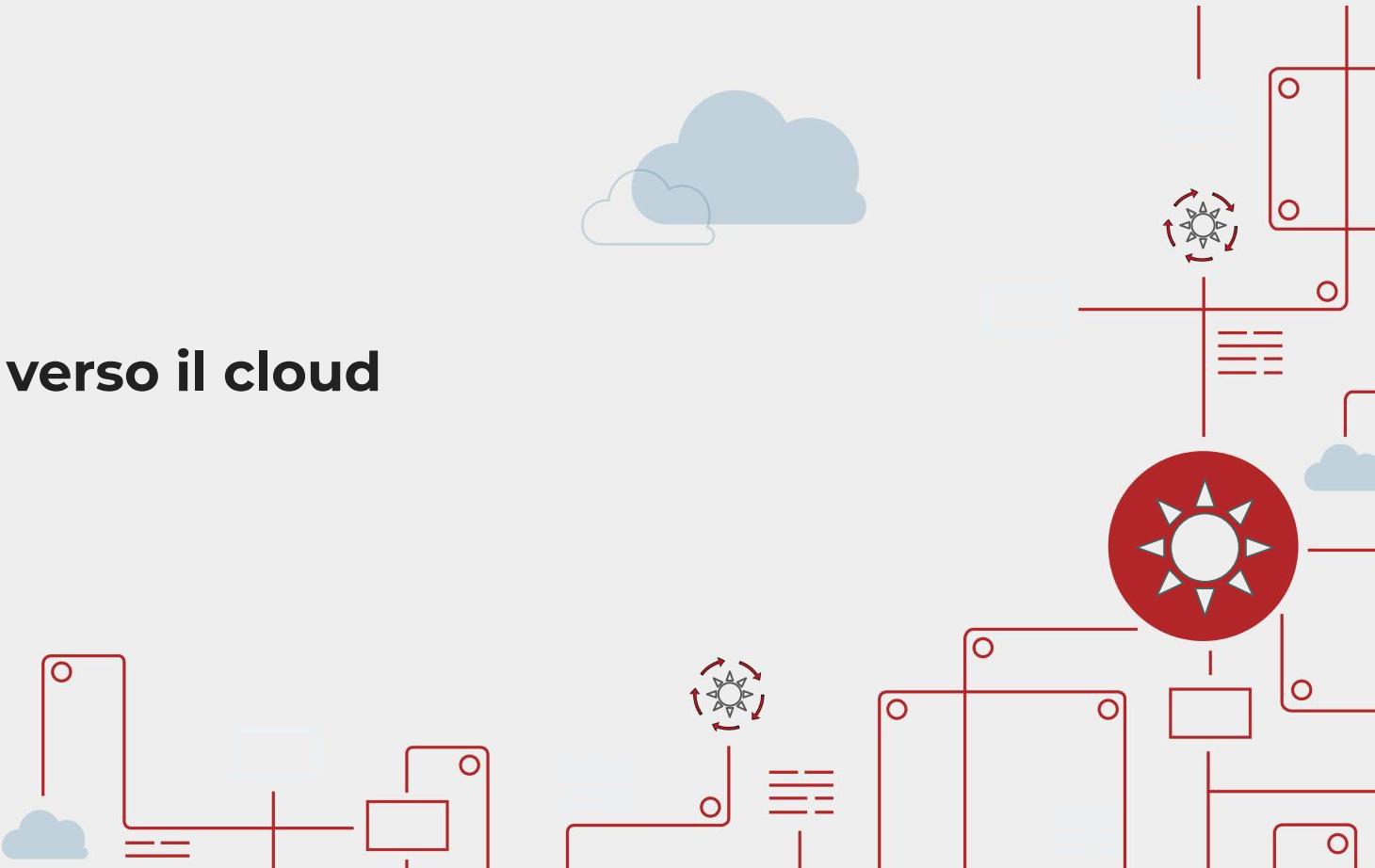
# L'ecosistema Java in un contesto dinamico

- Java è stato progettato per applicazioni monolitiche e complesse, long running, destinate a girare su server potenti
- Gli Application Server servivano appunto a concentrare molte applicazioni su un singolo server molto potente (on premise)
- Un altro aspetto della natura long-running del Java è rappresentato dal Garbage Collector, considerato un male necessario
- Il GC e il JIT girano nello stesso processo dell'applicazione e competono con essa per le risorse; le applicazioni che girano sullo stesso AS non sono isolate tra loro a sufficienza
- GC, JIT e AS non sono pensati per un contesto dinamico, con app effimere





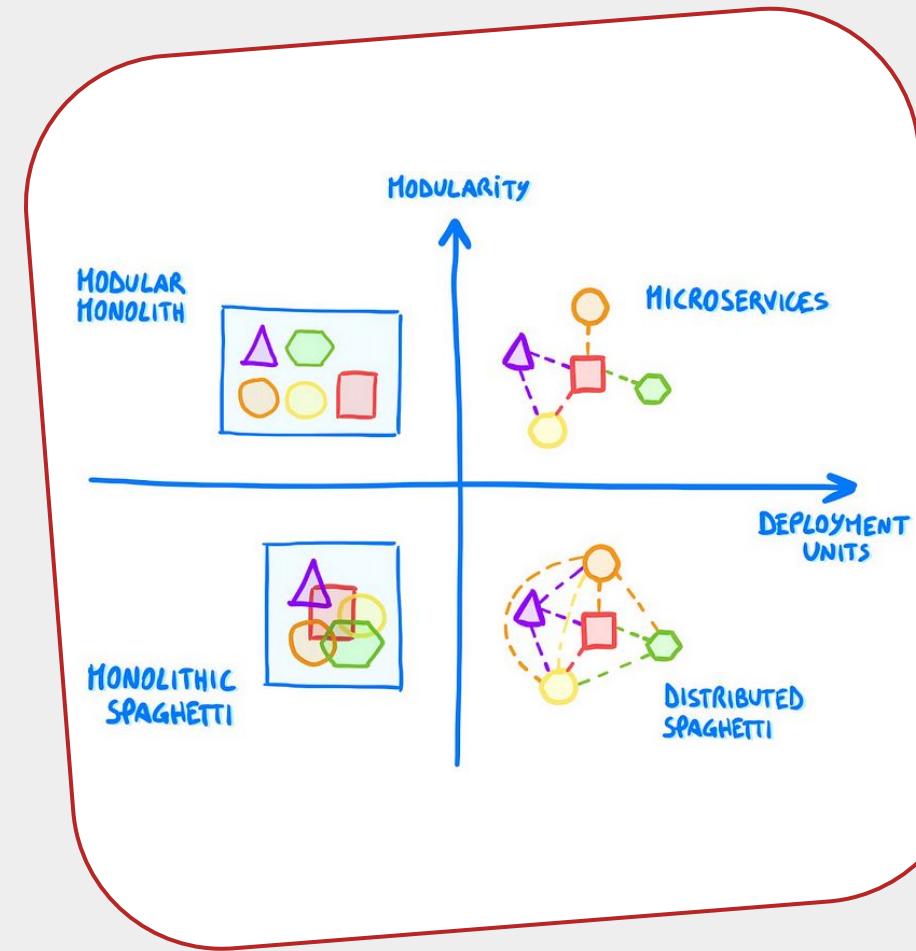
## Primi passi verso il cloud





# Una prima trasformazione: il "Majestic Monolith"

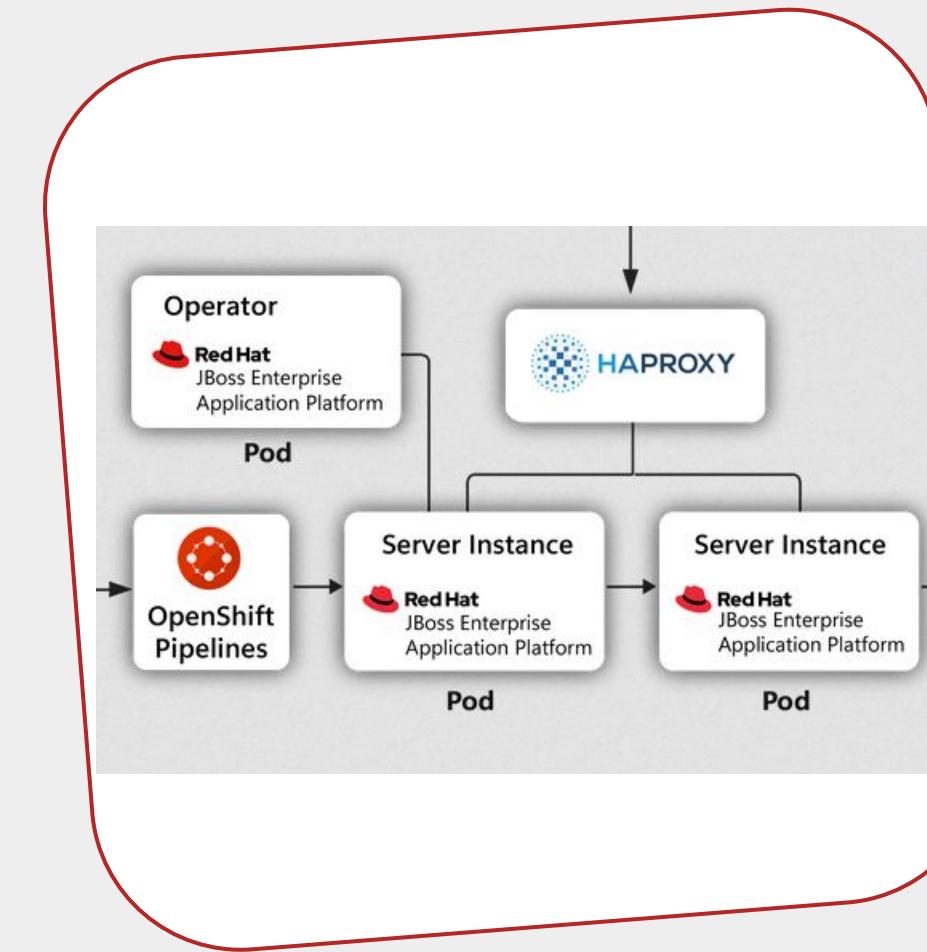
- Le aziende hanno investito enormi quantità di risorse nei monoliti, ma come beneficiare della container orchestration?
- E' possibile migliorare un monolite **preservando** la maggior parte del codice
- Il monolite va migrato su un nuovo **contenitore** e su una **moderna piattaforma di deployment**, che supporti processi evoluti di sviluppo e rilascio del software (AS moderno, container-aware)
- In questo modo le app vengono migrate in un contesto applicativo più **agile**
- E' possibile aggiungere nuove funzionalità grazie all'integrazione con i **servizi specifici del cloud** (nuovi componenti e integrazioni, nuove tecnologie)





# La versione moderna dell'AS (fast-moving AS)

- Gli AS moderni hanno caratteristiche che li rendono buoni “cloud citizen”
- Tempi di boot molto ridotti
- Architettura modulare, che comporta un forte risparmio di risorse (CPU, RAM)
- Configurazione container-aware
- Disponibilità di immagini ottimizzate, snellite ed aggiornate continuamente
- Supporto nativo alle probe di Kubernetes
- Disponibilità di Operator Kubernetes specifici che facilitano configurazione, deployment, clustering, diagnostica ecc.
- Supporto dell'aggiornamento a caldo (dev mode) e del testing in-container





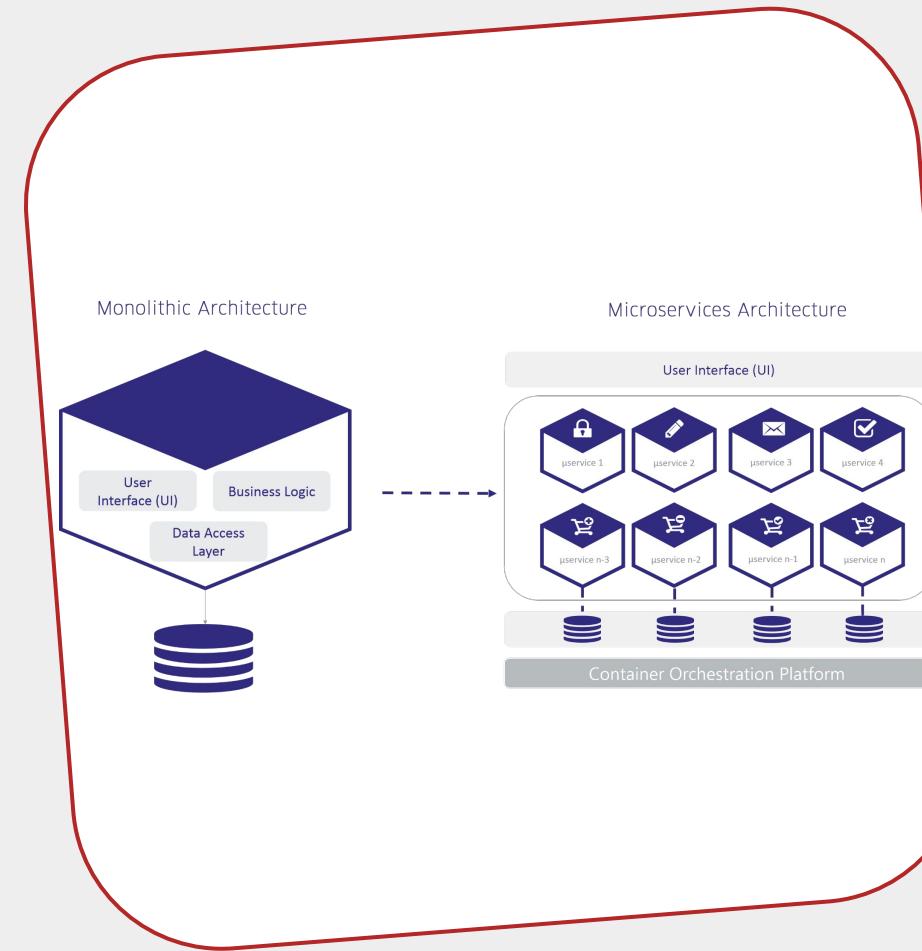
## Un approccio differente: i microservizi





# Cosa sono le architetture a microservizi (MSA) / 1

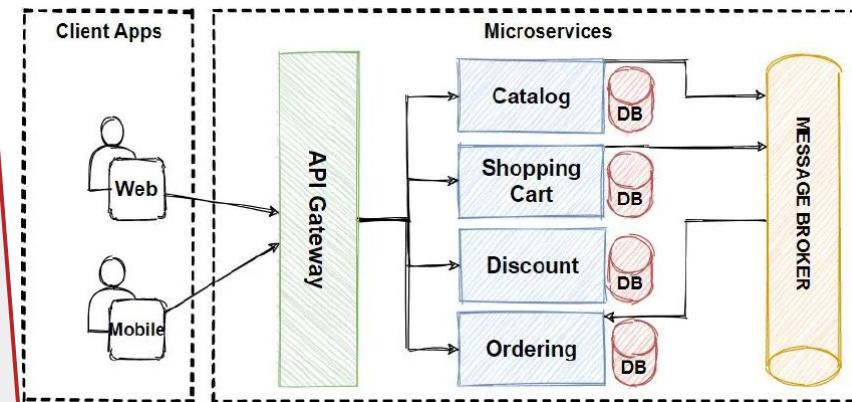
- I microservizi sono sia un'**architettura** che un **approccio** alla produzione del software
- L'approccio a microservizi consiste **in generale** nello “spezzare entità” (software, team, processi) in pezzi più piccoli, autonomi e che possono funzionare in parallelo, **riducendo il tempo necessario a produrre valore**
- Dal punto di vista **architetturale** (MSA) si passa dal monolite, in cui ogni componente viene creato all'interno di un'unica entità applicativa, a un insieme di app che interagiscono per completare le stesse attività, restando **indipendenti** gli uni dagli altri a livello di **logica** di business, di **dati** e di **processo** di OS





# Cosa sono le architetture a microservizi (MSA) / 2

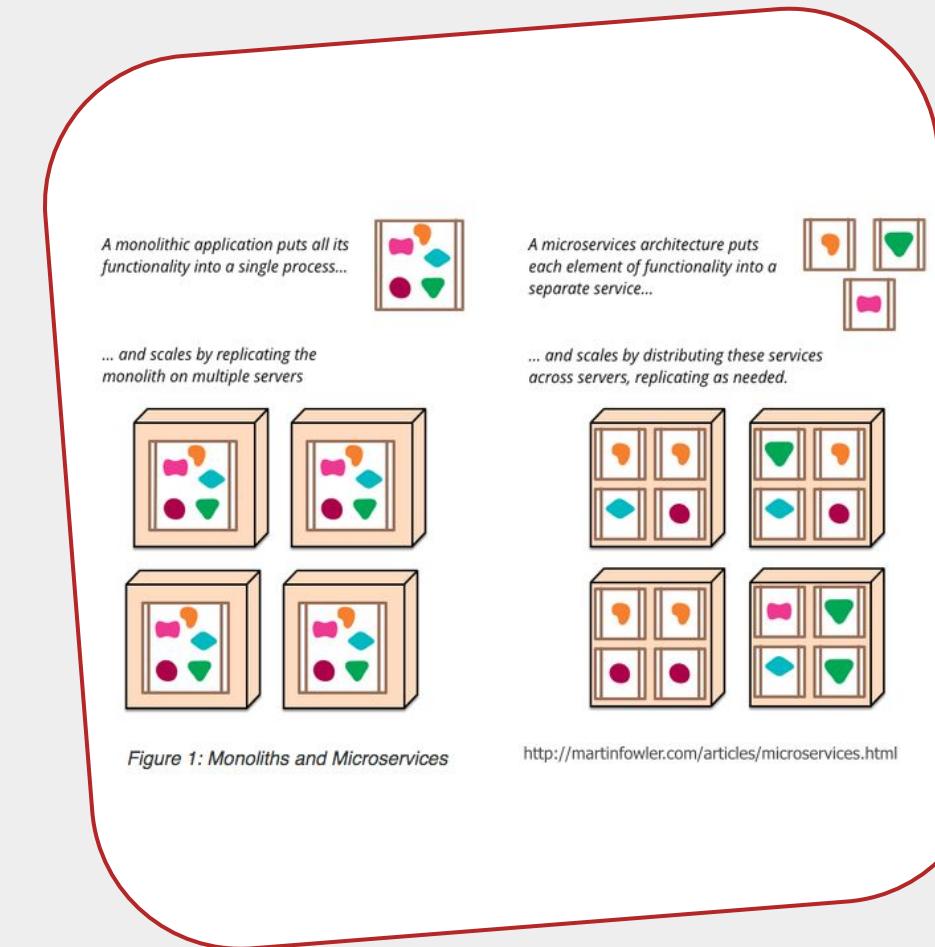
- Ciascun componente è dedicato a svolgere al meglio uno, e soltanto uno, specifico compito (forte coesione interna)
- La comunicazione tra microservizi avviene attraverso interfacce standard (API), in modo sincrono oppure asincrono
- Rispetto alle applicazioni monolitiche, i **(singoli)** microservizi sono più facili da compilare, testare, distribuire e aggiornare





# Problemi risolti dai microservizi / 1

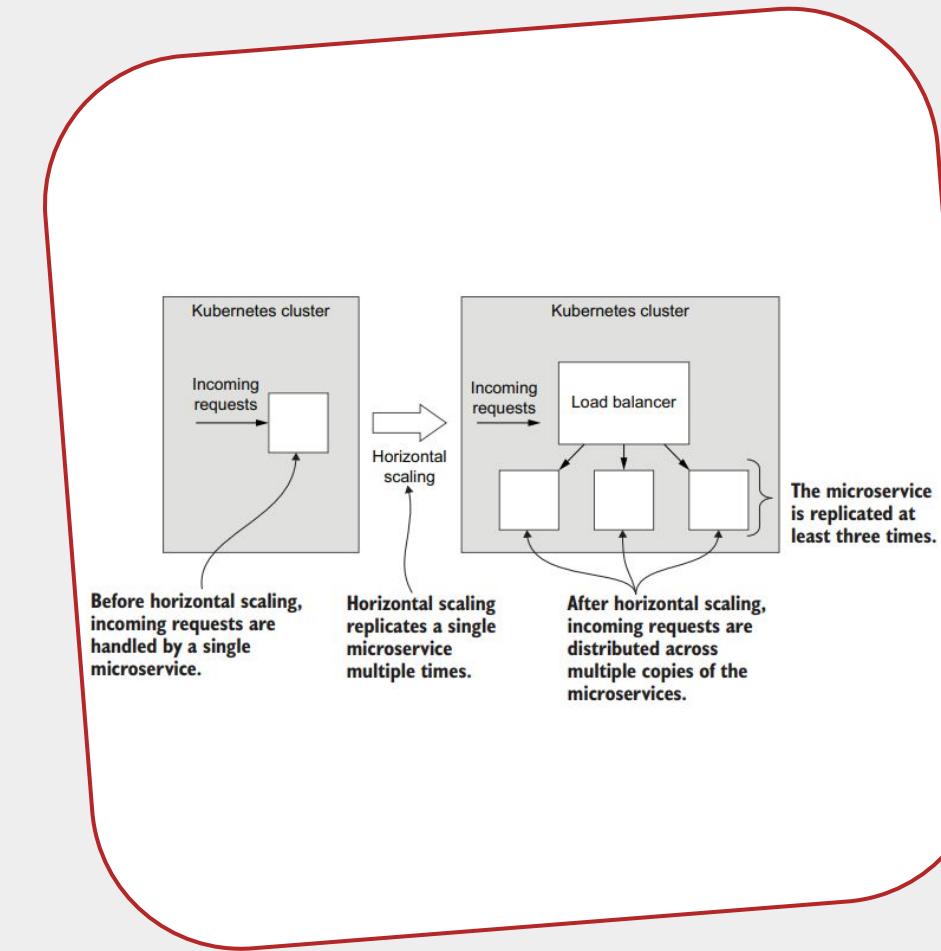
- Ripercorriamo i pain point delle app legacy
- **Tempo di startup:** un microservizio in genere parte più rapidamente di un monolite
- **Tempo di warm-up:** un microservizio contiene meno codice ed è più facile da compilare AOT (anche in Java)
- **Self-locking (scalabilità verticale limitata):** un microservizio è più semplice di un monolite ed è più facile scriverlo in modo da evitare competizione sui lock: la scalabilità verticale migliora
- **Scalabilità a grana grossa:** scalare selettivamente alcuni microservizi è meno dispendioso che scalare orizzontalmente un monolite





# Problemi risolti dai microservizi / 2

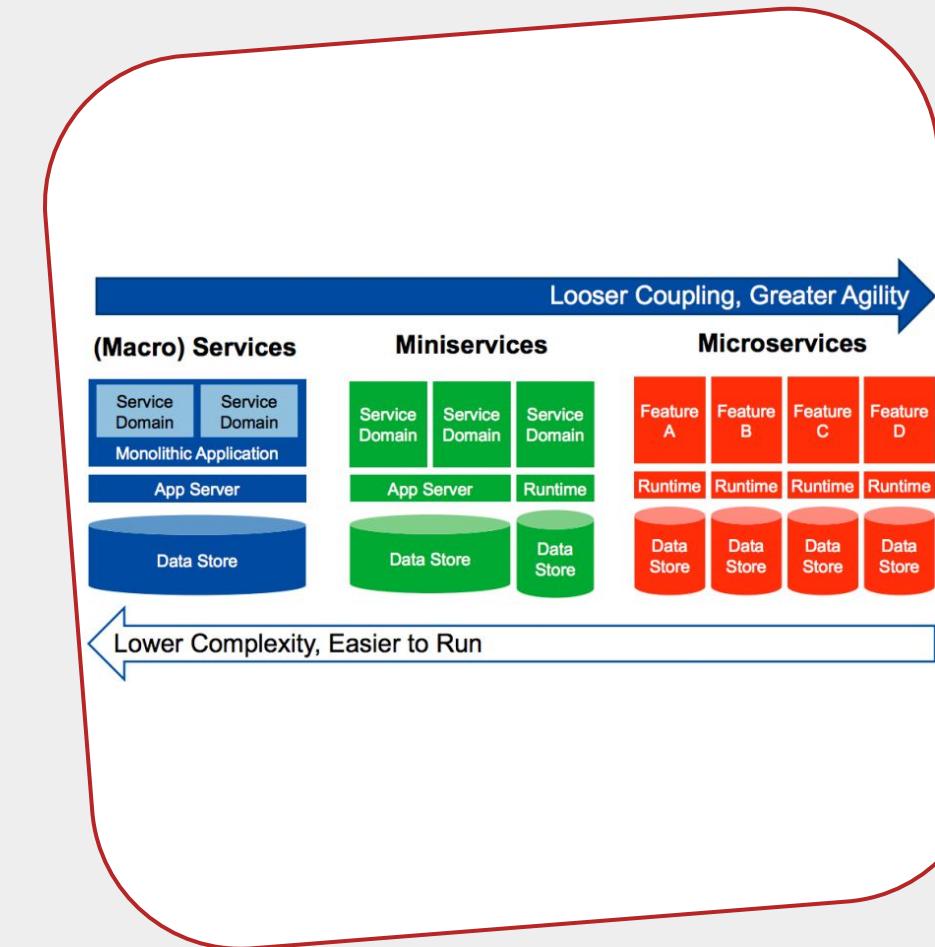
- **Scalabilità orizzontale limitata:** un microservizio ha meno “inerzia” in termini di tempo (startup e shutdown) e di spazio (memory footprint) rispetto a un monolite. Scale out/in migliore e più efficiente
- **Configurazione di request/limit e capacity planning difficoltosi:**
  - Il comportamento di un microservizio è più semplice da analizzare rispetto a quello di un monolite
  - E' più facile riempire in modo efficiente uno zaino con molti oggetti piccoli piuttosto che con pochi oggetti grandi





# Problemi risolti dai microservizi / 3

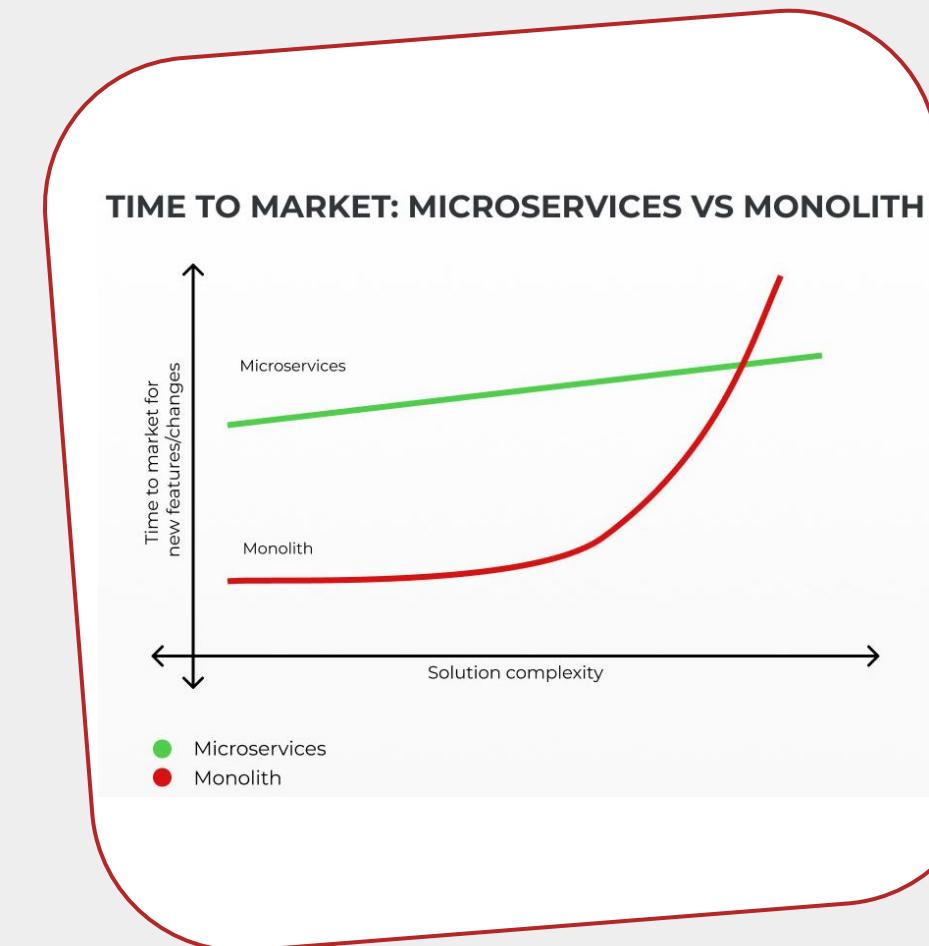
- **Manutenibilità delle applicazioni:** un microservizio è tendenzialmente più piccolo e semplice di un monolite (*ignoriamo per ora la complessità di un'applicazione distribuita*)
- **Scarsa agilità di business:** i microservizi permettono di adottare una strategia “divide et impera”. **Modificare** i singoli microservizi in base a nuovi requisiti di business è più facile che modificare un monolite
- **Tempi di sviluppo:** grazie ai tempi di **startup** rapidi, è possibile adottare tool e framework di **sviluppo** che accelerano in modo considerevole lo sviluppo e il test del codice





# Problemi risolti dai microservizi / 4

- **Tempi (e rischi) di rilascio:** rilasciare **selettivamente** i singoli microservizi è più rapido e meno rischioso che rilasciare un monolite (Big Bang)
- **Scarsa resilienza:** un monolite esibisce facilmente dei “single point of failure”; i microservizi sono in grado di mutuare la resilienza del cloud
- **Costi elevati:** le risorse di un monolite vanno dimensionate per il picco massimo di consumo; i microservizi possono avvalersi dell'autoscaling orizzontale
- **Complessità di gestione; mancanza di standardizzazione; mancanza di skill:** prego, attendere





# Obiettivo: un nuovo ecosistema applicativo

- Rifattorizzare a microservizi significa creare un nuovo **ecosistema** di app (per una nuova infrastruttura)
- Emergono alcune caratteristiche:
  - App ad alto consumo di risorse ⇒ App a basso consumo di risorse
  - App long-running ⇒ App effimere
  - App “ad alta inerzia” (in termini di tempi di startup/shutdown) ⇒ App a bassa inerzia
- L'applicazione originale viene decomposta in tante app, più piccole sia in termini di “spazio” (codice, risorse) che in termini di “tempo” (inerzia, durata, ciclo di vita)
- **Il nuovo ecosistema applicativo è adatto a un ambiente dinamico**





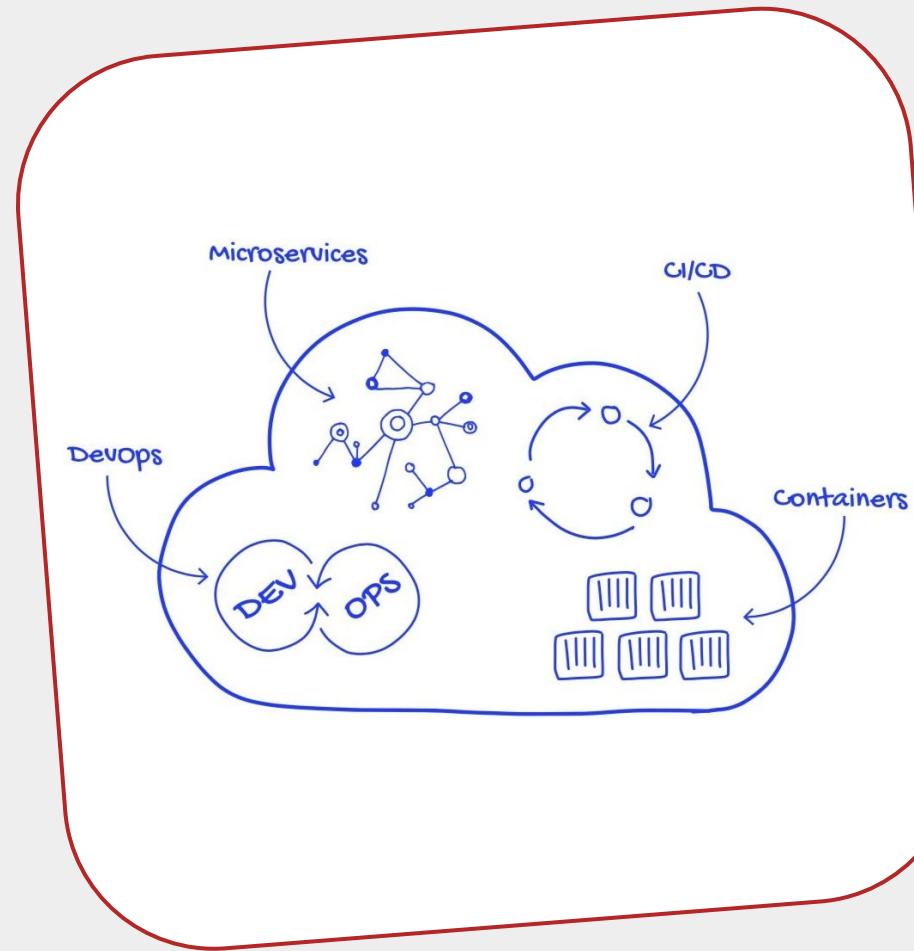
## Applicazioni cloud-native





# Caratteristiche delle applicazioni cloud-native

- L'ambiente cloud, dinamico ed elastico, esalta l'efficacia dei microservizi ma non è invece adatto alle app tradizionali
- Le caratteristiche dei microservizi viste finora sono **solo una parte** di quelle che un'app cloud-native è tenuta ad avere
- Prima ancora che al pattern architetturale dei microservizi, il concetto di cloud-native è riconducibile a una serie di principi noti come 12 Factor. Una vera *cloud app* è sia MSA che 12 Factor
- Per poter sfruttare le caratteristiche del cloud bisogna rifactorizzare le app legacy in MSA / 12 Factor, trasformando l'architettura e **lo sviluppo applicativo**





# Applicazioni 12 Factor

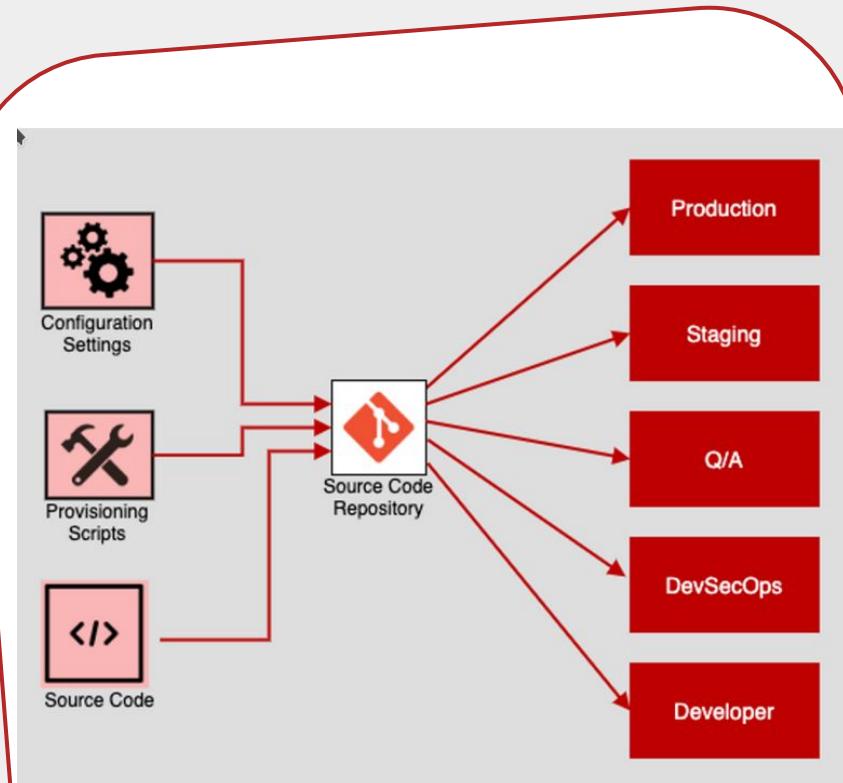
- Il “12 Factor App” è un insieme di principi che descrive un particolare approccio allo sviluppo del software
- L'obiettivo è creare app **portabili** tra ambienti, che possano essere rilasciate con **pochi rischi** (pochi difetti e possibilità di rollback), facilmente **scalabili**, con startup e shutdown ordinati e **rapidi, resilienti** a crash e problemi di rete (altro aspetto dinamico), aggiornabili in modo **rapido** e sempre in uno stato **predicibile** (noto e documentato)
- Seguendo tali linee guida è possibile creare app particolarmente adatte a girare in ambiente cloud, aumentando la qualità del software (**valore** percepito dagli utenti)





# Alcuni principi 12 Factor: codebase

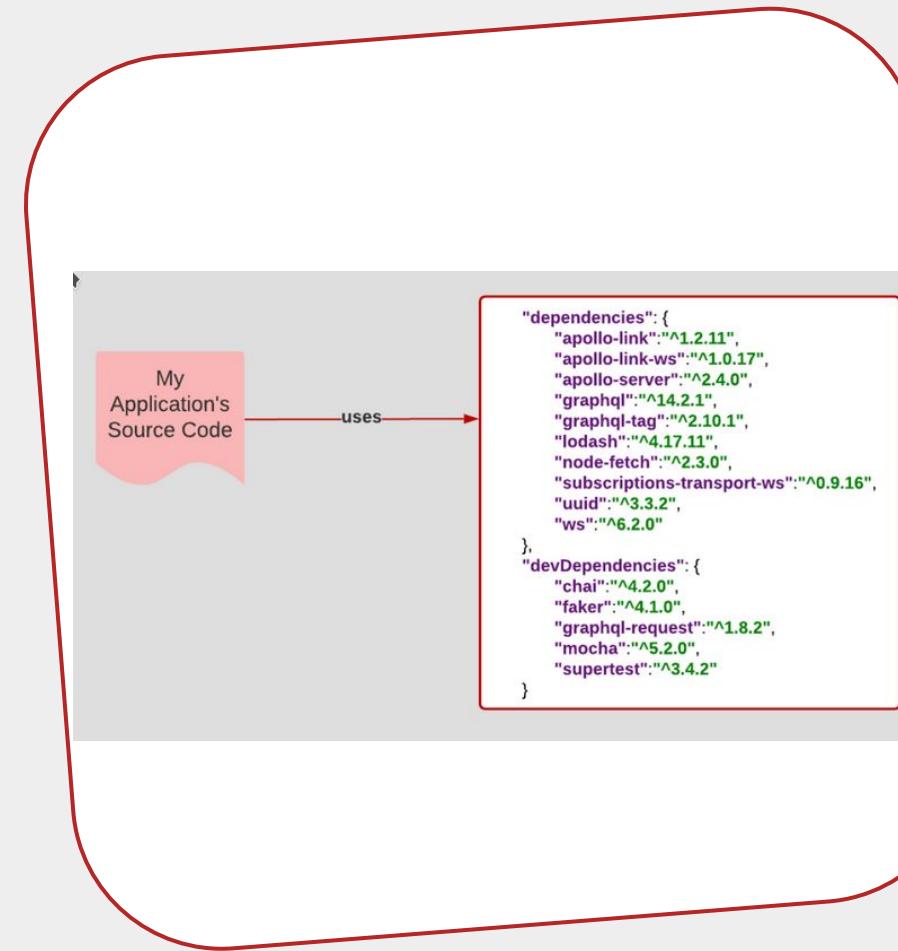
- Esiste una sola fonte di verità per tutto: il codice mantenuto in un sistema di versionamento (**DVCS**)
- Il “Codice” comprende: codice applicativo, configurazioni, manifest di deployment, script di automazione, IaC ecc.
- Sia persone (Dev, Ops, Sec) che processi di automazione (CICD) accedono alla stessa base di codice
- Dal codice si producono delle release **immutabili** (build artifact + configurazione per il deployment in un certo ambiente). Tutti i deployment afferiscono a una precisa **release** della codebase
- La codebase garantisce la **predicibilità** degli aggiornamenti





# Alcuni principi 12 Factor: dipendenze

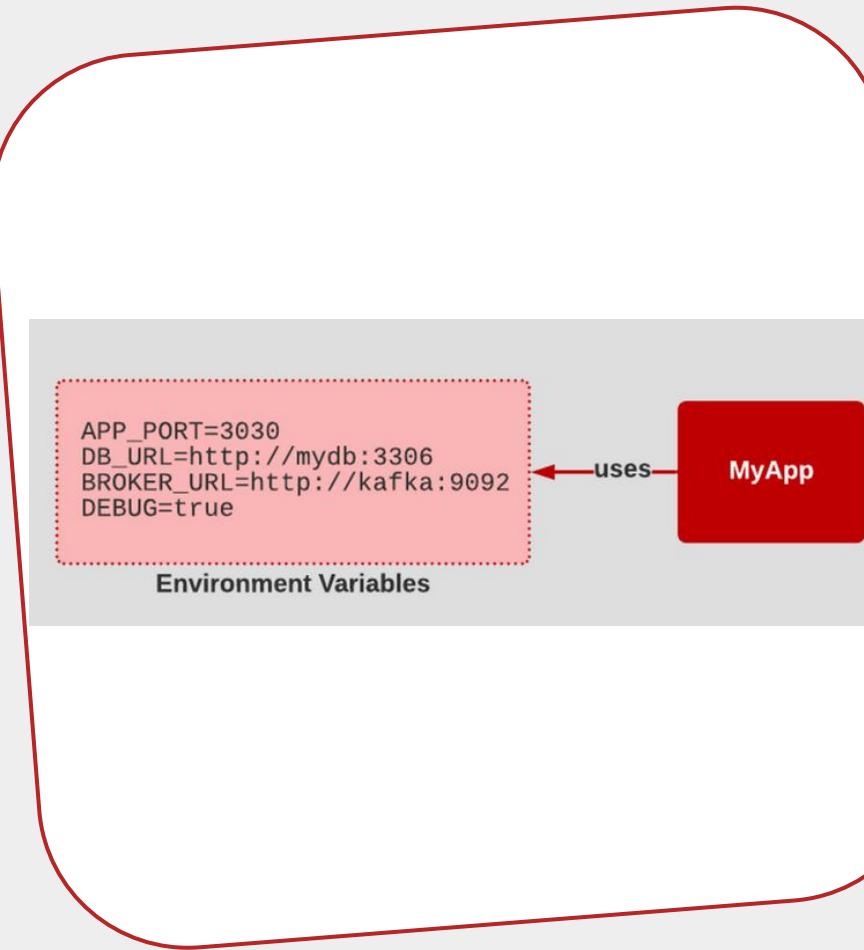
- **Esplicitazione delle dipendenze:** qualsiasi dipendenza deve essere dichiarata in modo esplicito mediante un manifest mantenuto nella codebase e recuperato a build time. Le dipendenze in sé **non** fanno parte della codebase
- **Isolamento:** l'app deve essere indipendente dai pacchetti di sistema e auto-contenuta. L'app e le sue dipendenze vanno incapsulate insieme in un pacchetto portatile tra i vari ambienti
- Corretta gestione delle dipendenze e isolamento garantiscono **portabilità** e **predicibilità**





# Alcuni principi 12 Factor: configurazione

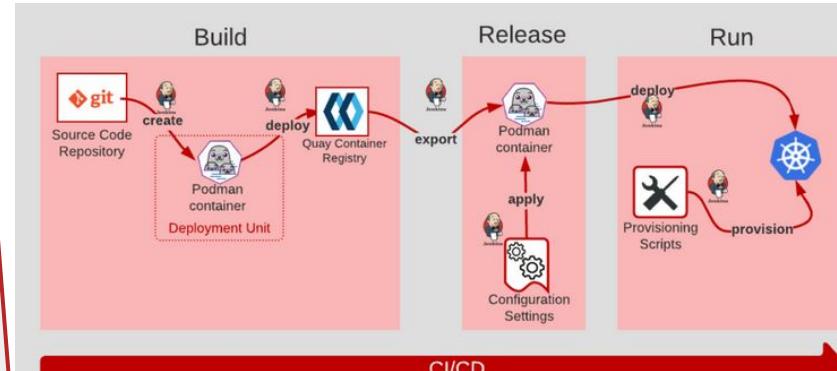
- La configurazione applicativa deve essere disaccoppiata dal codice, esclusa dalle build, e iniettata negli ambienti di runtime mediante **variabili d'ambiente**
- Lo stesso pacchetto binario può così essere installato in ambienti diversi variando la configurazione; inoltre, la configurazione risulta indipendente dal linguaggio e dal sistema operativo
- Spostare la configurazione nelle variabili d'ambiente garantisce **portabilità**





# Alcuni principi 12 Factor: build, release, run

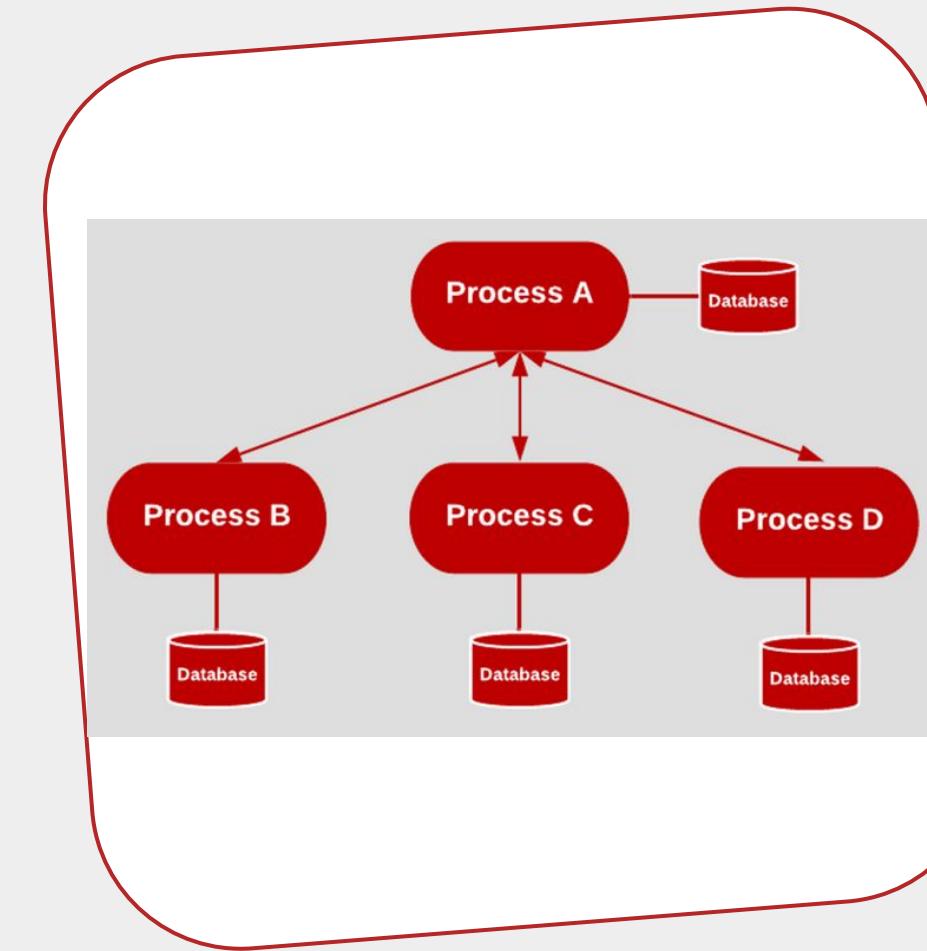
- Il processo di deployment del software va separato in tre fasi distinte: build, rilascio ed esecuzione
- La fase di build pacchettizza codice e dipendenze e produce un **artefatto** di build
- La fase di rilascio combina l'artefatto di build con la configurazione di un ambiente specifico, producendo una **release**
- La fase di esecuzione prevede la creazione dell'ambiente e l'avvio della release
- Questo principio garantisce **predicibilità**, **rapidità** di aggiornamento e **affidabilità** (prevedibilità di comportamento)





# Alcuni principi 12 Factor: processi stateless

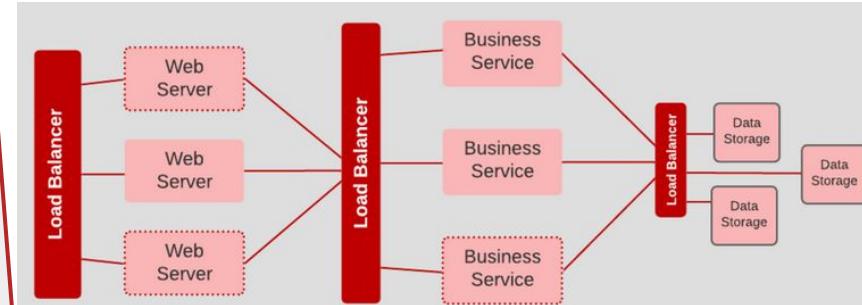
- Una app deve girare come un insieme di processi **stateless**, che non condividono dati tra loro
- Stateless: ciascun processo non può mantenere dati sullo spazio di archiviazione disponibile localmente né può contare sul fatto che qualsiasi richiesta successiva venga gestita dallo stesso processo (memoria)
- Ciascun processo è **effimero** e può essere rimosso in qualsiasi momento, perdendo i dati che sta gestendo (nessuna memoria)
- I dati vanno eventualmente mantenuti su un servizio di supporto esterno (database, cache distribuita ecc.)
- Questo principio garantisce **scalabilità**





# Alcuni principi 12 Factor: concorrenza

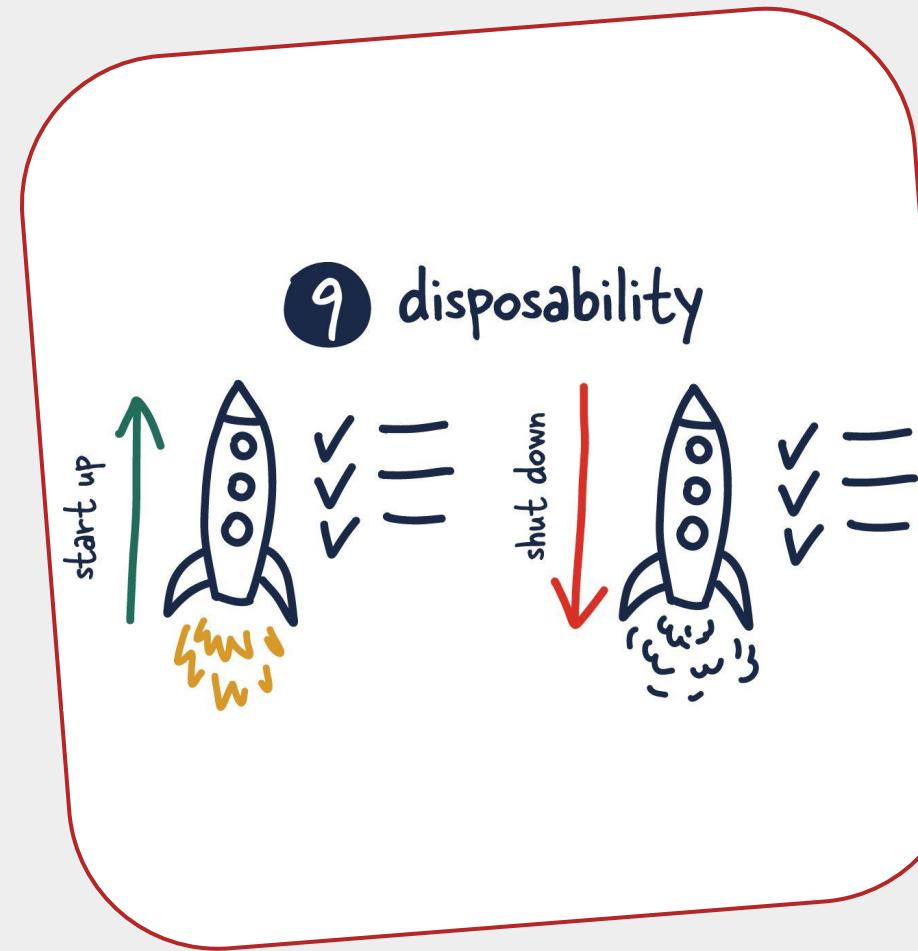
- Un'applicazione deve girare come un insieme di processi concorrenti
- I ms girano come processi suddivisi in gruppi di servizio (**Service**) in base alla tipologia e allo scopo
- Ciascun gruppo (**repliche di un ms**) opera dietro un load balancer e può essere scalato indipendentemente dagli altri
- La capacità di un'applicazione può essere incrementata aggiungendo nuove istanze piuttosto che aumentando le risorse di una singola macchina
- Questo principio abilita la **scalabilità selettiva** (piuttosto che la scalabilità a grana grossa o la scalabilità verticale) e la **resilienza**





# Alcuni principi 12 Factor: sostituibilità

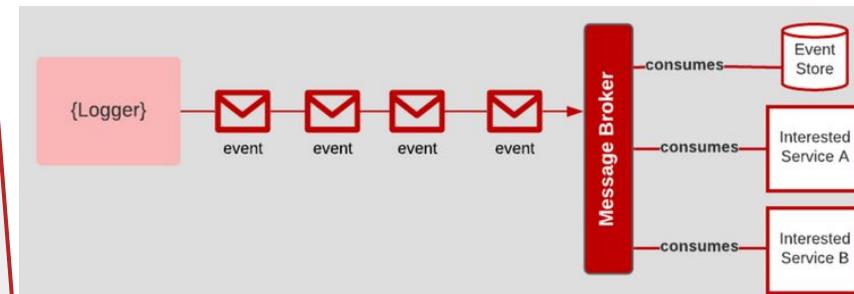
- Una app deve avviarsi nel più breve tempo possibile e accettare traffico solo quando pienamente operativa (**graceful startup**)
- Una app deve arrestarsi in modo pulito, tipicamente gestendo SIGTERM in modo corretto. L'applicazione deve completare le operazioni in corso, non accettare ulteriori richieste e liberare tutte le risorse prima di uscire (**graceful shutdown**)
- App con tali proprietà sono dette **sostituibili (disposable)**
- Le app sono effimere e possono essere terminate in un momento qualsiasi, ma se sono **sostituibili** ciò non comporta effetti di bordo e possono essere riavviate senza problemi





# Alcuni principi 12 Factor: gestione dei log

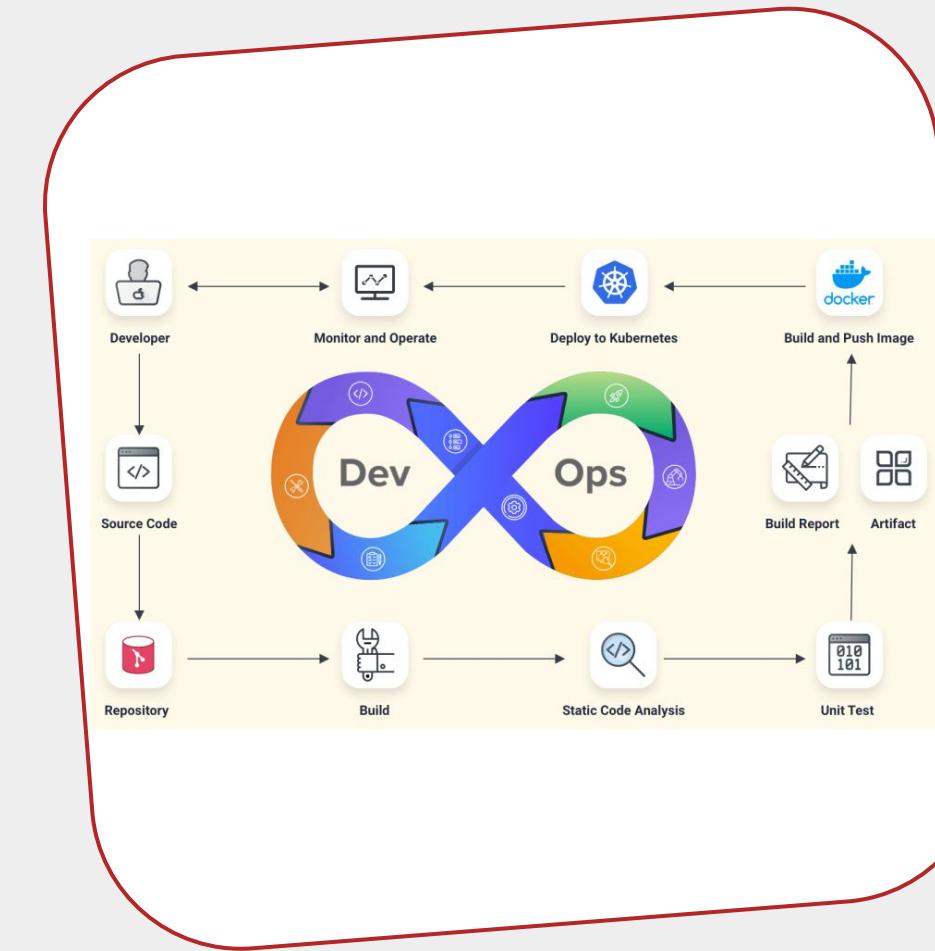
- Un'app non deve produrre i log scrivendo nel proprio file-system locale ma deve inviarli a STDOUT/STDERR
- La raccolta, l'instradamento e l'elaborazione dei log vengono gestiti da processi disaccoppiati dalle app
- I log possono così essere consumati da vari tool, aggregati e analizzati; inoltre non sono legati alle singole macchine e sopravvivono alla terminazione dell'app
- App effimere e scalabili, se non vogliono avere problemi, devono gestire i log in modo “agnostico”, secondo questo principio





# 12 Factor, MSA, container e DevOps

- App 12 Factor non implica per forza MSA: una app può essere resa cloud-native senza ricorrere a un refactoring architettonico (Majestic Monolith)
- L'adozione di container e Kubernetes agevola molto la realizzazione di App 12 Factor (cloud-native)
- Una **container platform** con servizi self-service, integrata con dei workflow CICD, permette di implementare completamente i principi 12 Factor (di sfruttare al massimo le caratteristiche del cloud) e di ottimizzare la cooperazione tra Dev e Ops (DevOps)
- Il **processo di produzione** del software migliora di pari passo con le performance





# Biomi applicativi

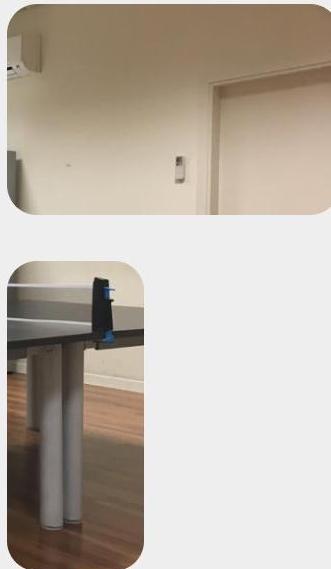
- Le app MSA sono **complesse**; si giustificano solo in un contesto altamente dinamico come il cloud, dotato di servizi avanzati che aiutano a gestire la complessità
- Un contesto tradizionale, mono-server e senza automazioni, è inadatto a una MSA almeno quanto il cloud è inadatto a una app tradizionale
- Il **refactoring** di una app tradizionale in MSA ha lo scopo di introdurre proprietà desiderabili che rendono le applicazioni realmente cloud-native e quindi in grado di sfruttare i vantaggi del cloud





# Bibliografia e link utili

- 3 Cloud Migration Approaches and Their Pros and Cons  
<https://bluexp.netapp.com/blog/cvo-blg-cloud-migration-approach-rehost-refactor-or-replatform>
- Red Hat - What is Java application modernization?  
<https://www.redhat.com/en/topics/application-modernization/what-is-java-application-modernization>
- Horizontal Vs. Vertical Scaling: Which Is Right For Your App?  
<https://www.missioncloud.com/blog/horizontal-vs-vertical-scaling-which-is-right-for-your-app>
- Microservices  
<https://martinfowler.com/articles/microservices.html>
- A cosa servono i microservizi?  
<https://www.redhat.com/it/topics/microservices>
- 12 Factor App meets Kubernetes  
<https://www.redhat.com/architect/12-factor-app-containers>



extra**red**



Via Salvo D'Acquisto 40/P  
56025, Pontedera (Pisa), Italia



0587 975800



[www.extrasys.it/it/red](http://www.extrasys.it/it/red)

**Grazie per l'attenzione**

**Davide Carmelo Costanza**

Lead Solution Architect, PM  
[davide.costanza@extrasys.it](mailto:davide.costanza@extrasys.it)

# EXTRA RED

Engineering Group -Digital Technology



# Extracted in a snapshot

>50 clients

in customer  
baseline

~5M€  
revenues  
in 2023

>70 HCs

highly certified  
consultants



## CLOUD & INFRA

- Red Hat Openshift
- Red Hat Ansible Automation Platf.
- Red Hat Fuse + Red Hat 3Scale
- IBM BAM OE (ex Red Hat PAM)

Cloud, Orchestration, DevOps, Security  
Cloud-Native Apps, Integration, Stream  
Processing, Application Migration &  
Modernization

## DIGITAL EXP.

- Cloud-Native Apps
- Liferay DXP
- Angular & React

Portal Development,  
frontend development

## DATA & AI

- IBM Watson
- ELK & Grafana
- InfluxDB
- Microsoft PowerBI

Big Data Analytics, NLP,  
Monitoring, Imagine  
Recognition, Machine  
Learning

## IN PARTNERSHIP WITH:



SERVICE PARTNER  
SILVER





# TECHNOLOGY SERVICE PROVIDER



Premier  
Business Partner

The leading company in the world of Open Source

DevOps ←

Cloud & Infrastructure ←

Middleware ←

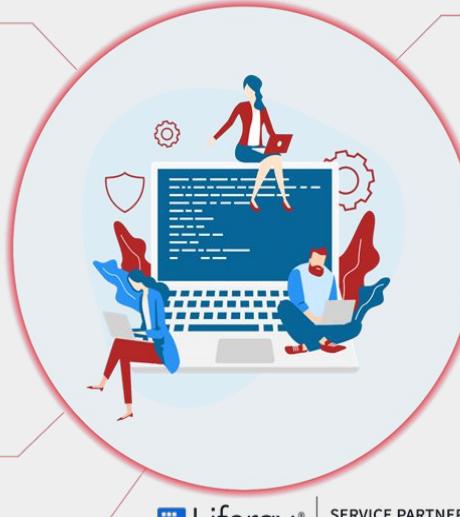
AMM ←



Open Source Time Series DB platform for Metrics and Events

Application/infrastructure monitoring & assistance ←

IoT & Industria 4.0 ←



Digital Experience Platform

DXP (Digital Experience Platform) & App Mobile ←



Public & Hybrid Cloud, Cloud Paks, Watson, Augmented Intelligence

→ AI, BI & Data

→ Cloud



World leading company in Cloud services

→ AI, BI & Data

→ Cloud

→ IoT



# I NOSTRI TEAM



## PLATFORM TEAM

Cloud, Orchestration, DevOps,  
Security



## MIDDLEWARE TEAM

Integration, Stream Processing,  
Application Migration &  
Modernization



## AI & DATA TEAM

Big Data Analytics, NLP, Monitoring,  
Imagine Recognition, Machine  
Learning



## DXP TEAM

Cloud-Native Apps, DXP, Mobile

## OUR TECHNOLOGIES

