Microservizi, lambda e container: che differenza c'è? Parliamone!

29 Febbraio 2024 - 7 min. read

AWS Lambda *Containers* *Microservices*

Container, Lambda e microservizi sono concetti tanto popolari e importanti, quanto fraintesi dai tecnici nello sviluppo software moderno. A voltesi parla di architetture a microservizi quando, in realtà, si tratta solo di un gruppo di piccoli container con monoliti fortemente accoppiati.

In questo articolo, faremo chiarezza sui microservizi, e sui servizi gestiti da AWS che ci faciliteranno nella loro implementazione.

"Usiamo i container, quindi il nostro sistema è a microservizi."

Sfortunatamente, le cose non sono così semplici: i microservizi devono aderire a regole rigorose e avere definizioni chiare.

Entraimo nel vivo della discussione.

I microservizi dovrebbero essere:

- **Strettamente definiti**: i microservizi devono implementare una singola funzione per la logica aziendale, come il logging, la registrazione dell'utente, il calcolo del prezzo.
- **Debolmente accoppiati**: un microservizio non deve dipendere da altri microservizi per implementare un'operazione, e la disponibilità di servizi esterni non deve influire sulla sua disponibilità.
- **Fortemente incapsulati**: la comunicazione tra microservizi deve essere standard, senza codice condiviso, e i dettagli sul funzionamento interno non devono essere accessibili, nemmeno per quanto riguarda la persistenza dei dati.

- **Indipendentemente distribuibili**: il rilascio di una nuova versione di un microservizio non deve influire sulla disponibilità e sull'implementazione di altri servizi.
- I**ndipendentemente scalabili**: questo è una conseguenza dalle altre esigenze. Se un servizio ha bisogno di più risorse di calcolo, deve ottenerle senza dipendere da o influenzare altri servizi.

Vediamo ora i principi del design dei container:

- **Singola responsabilità**: un container si occupa di un singolo aspetto.
- **Alta osservabilità**: ogni container deve implementare API per osservare il suo stato e gestire l'applicazione.
- **Conformità al ciclo di vita**: un container dovrebbe essere in grado di leggere gli eventi riguardanti il ciclo di vita della piattaforma che lo esegue e reagire agli eventi.
- **Immagine immutabile**: un container è immutabile; il suo codice non deve essere specializzato e cambiare tra ambienti differenti.
- **Usa e getta**: un'applicazione containerizzata è effimera, quindi non devono essere memorizzati localmente dati e utilizzati per la persistenza dello stato
- **Auto-contenimento**: le librerie necessarie per l'applicazione sono disponibili all'interno del filesystem del container. L'unica dipendenza richiesta è il kernel.
- **Confinamento del runtime**: le risorse a disposizione del container sono definite e limitate. Il runtime applica i limiti per evitare che il sistema sia in sofferenza.

Come potete vedere, definizioni e concetti si sovrappongono in alcuni punti , ma il diavolo si nasconde nei dettagli: l'uso di una tecnologia specifica non significa che il design dell'architettura avrà automaticamente tutte le proprietà che ci aspettiamo (e non abbiamo nemmeno menzionato Lambda!).

I container, infatti, **offrono un modo per impacchettare in modo portabile ed eseguire il software in un ambiente isolato**. Allo stesso tempo, un approccio a microservizi aiuta a dividere un problema complesso in più servizi più semplici, fortemente disaccoppiati, che comunicano utilizzando API e possono essere gestiti da diversi team.

Benefici e sfide dell'uso di queste tecnologie

I container possono aiutare con la scalabilità, la portabilità e l'efficienza delle risorse, ma richiedono anche un'occhio attento a orchestrazione, monitoraggio e sicurezza.

Le lambda possono aiutare con l'agilità, la convenienza economica e la scalabilità, ma possono portare problemi di prestazioni e aumentare la complessità d'implementazione.

I microservizi possono aiutare con la modularità, la flessibilità e la tolleranza ai guasti, ma introducono anche latenza di rete, coerenza dei dati e aumentano la difficoltà nell'effettuare i test.

Ecco alcuni esempi di container e lambda che non possono essere identificati come microservizi.

- Un WordPress containerizzato (non è strettamente definito)
- Una lambda che si basa interamente sull'output di un'altra lambda (non è fortemente disaccoppiata)
- Due container che accedono alla stessa tabella del database (non c'è incapsulamento)

Ci sono molti esempi e sicuramente potete pensare a qualcosa nel vostro lavoro quotidiano.

Non ci concentreremo sul discutere se un'architettura orientata ai microservizi sia migliore e per quale caso d'uso perché miriamo a semplificare la nostra vita mentre gestiamo le infrastrutture che eseguiranno le nostre applicazioni.

Ricordate che quando usiamo i container e le lambda per implementare una strategia a microservizi, dobbiamo aggiungere almeno tre componenti alla nostra architettura: un runtime, un orchestrator e un discovery service.

AWS offre diversi servizi gestiti tra cui scegliere per andare nella direzione dei microservizi, semplificandoci la vita..

Compute

Si tratta della parte più ovvia. Possiamo utilizzare EC2, Fargate (per ECS o EKS), Lambda. EC2 sembra controintuitivo, ma ricordiamo che i microservizi riguardano il design, non la tecnologia.

Lambda e Fargate offrono piattaforme di calcolo serverless, quindi la spesa si basa esclusivamente sull'uso. Sono la soluzione perfetta per un approccio a microservizi: se un microservizio basato su Lambda viene raramente invocato, il suo costo sarà marginale.

L'uso di Fargate e Lambda aiuta anche ad adattarsi ai modelli di utilizzo: la scalabilità orizzontale delle risorse orizzontali utilizzando piccole unità di calcolo ottimizza la scalabilità.

Orchestrazione

L’orchestrazione è un componente critico di un’architettura a microservizi perché aiuta ad automatizzare i rilasci, aggiunge scalabilità, consente il coordinamento e introduce la tolleranza ai guasti dei microservizi, gestendo anche il ciclo di vita del servizio e i flussi di esecuzione.

ECS e EKS (su EC2 e Fargate) offrono libertà di scelta per il mondo dei container e minimizzano l’effort di gestione; API Gateway e Step Functions aiutano con l’esecuzione e l’integrazione di Lambda e altri servizi AWS.

AWS AppMesh può anche essere utilizzato per gestire il routing e l’autorizzazione tra i servizi, aggiungendo un ulteriore strato di osservabilità.

Discovery Service

In un ambiente dinamico, utilizzare le risorse senza fare affidamento a nomi o indirizzi IP fissi è obbligatorio (questo requisito in realtà si applica a ogni ambiente cloud).

I microservizi dovrebbero anche essere in grado di notificare automaticamente la loro presenza a un registro centrale, che tiene traccia di cosa si trova dove e facilita la comunicazione.

Il discovery service mantiene un registro dei servizi disponibili e fornisce un meccanismo per gli utilizzatori per l'interrogazione e la scoperta, gestendo i cambiamenti dovuti alla scalabilità, ai guasti o agli aggiornamenti.

EKS utilizza il DNS interno integrato di Kubernetes, mentre ECS può fare affidamento su Amazon ECS Service Connect (questi due servizi possono essere utilizzati insieme).

Altri aspetti da tenere in considerazione

Per distribuire i microservizi in modo indipendente, avremo bisogno di pipelin. Inoltre sarà necessario impostare un sistema di logging efficiente per monitorare i container. Non dimentichiamoci poi dei componenti di comunicazione (come code, bilanciatori di carico e trigger di eventi) per disaccoppiare l'applicazione.

Questi componenti aumenteranno sicuramente la complessità dell'architettura, quindi è fondamentale sceglierli con attenzione: a volte insistiamo nell'uso di una tecnologia specifica, piegando le esigenze aziendali e il design della soluzione alla nostra fantasia o curiosità. Scegliere il design in base alla tecnologia è pericoloso e può portare un progetto

a lfallimento. design sbagliato influisce sulla disponibilità, sulla gestibilità e, quindi, sul costo della soluzione.

Per Concludere

Come sempre, non esiste una soluzione perfetta: ci sono casi d'uso che si adattano perfettamente ai microservizi, mentre, a volte, suddividere ogni componente in servizi diversi può solo aumentare la complessità del progetto: c'è davvero bisogno di un servizio di registrazione utente per l'e-commerce di un negozio di animali di paese?

Iniziare con un monolite per accelerare il tempo di sviluppo e rilasciare il prodotto più velocemente potrebbe essere un approccio migliore rispetto al perdere il momento giusto per mettere sul mercato la soluzione.

In quale situazione e caso d'uso siete? State mantenendo un monolite legacy, cercando di scomporlo, o progettando un nuovo sistema completamente da zero? Fateci sapere nei commenti!

Nel frattempo, se state pensando di rifattorizzare un'architettura monolitica in microservizi, questo articolo di Alessandro e Simone può darvi più informazioni.

Buona fortuna con i vostri microservizi!

About Proud2beCloud

Proud2beCloud è il blog di beSharp, APN Premier Consulting Partner italiano esperto nella progettazione, implementazione e gestione di infrastrutture Cloud complesse e servizi AWS avanzati. Prima di essere scrittori, siamo Solutions Architect che, dal 2007, lavorano quotidianamente con i servizi AWS. Siamo innovatori alla costante ricerca della soluzione più all'avanguardia per noi e per i nostri clienti. Su Proud2beCloud condividiamo regolarmente i nostri migliori spunti con chi come noi, per lavoro o per passione, lavora con il Cloud di AWS. Partecipa alla discussione!

Damiano Giorgi

Ex sistemista on-prem, pigro e incline all'automazione di task noiosi. Alla ricerca costante di novità tecnologiche e quindi passato al cloud per trovare nuovi stimoli.L'unico hardware a cui mi

dedico ora è quello del mio basso; se non mi trovate in ufficio o in sala prove provate al pub o in qualche aeroporto!
