Nightmare infrastructure – episodio 4: nessun limite al peggio.

29 Ottobre 2025 - 1 min. read

"Sai, penso che questa cosa del Natale non sia così complicata come sembra! Ma perché dovrebbero divertirsi solo loro? Dovrebbe essere di tutti! Anzi, non di tutti, ma mio! Perché, potrei fare un albero di Natale! E non c'è una ragione valida, non potrei avere un periodo natalizio! Scommetto che potrei anche migliorarlo! Ed è esattamente quello che farò!"

Jack Skellington

La dichiarazione di Jack Skellington esprime alla perfezione l'overconfidence di un ingegnere, la sua curiosità e la voglia di migliorare o reinventare un sistema, che porta spesso con risultati caotici o disastrosi.

Cattura lo spirito che fa prendere un processo semplice per provare a migliorarlo, per scoprire troppo tardi che la complessità originale era nascosta, appena fuori dalla nostra vista.

Molte tradizioni nascono senza intenzioni serie, senza pianificazione, quasi per esperimento. Questa serie di articoli è diventata una tradizione, che ci fa preparare in

anticipo e prendere l'anno, pensando a quando arriverà di nuovo Halloween per farvi venire i brividi.

Quindi, bando alle ciance, iniziamo il nostro viaggio negli orrori di quest'anno!

Il container di Schrödinger

"Quindi siamo noi nella scatola. Noi siamo il gatto. Siamo sia vivi che morti. Quindi ci sono due realtà separate, presumibilmente finché la cometa non passa."

Coherence, 2013

Il gatto di Schrödinger è un paradosso per spiegare il principio di indeterminazione della meccanica quantistica: osservando un sistema, lo si altera, rendendo impossibile determinare il suo stato iniziale.

Per rendere il concetto, basta immaginare che un gatto e una fiala di veleno siano messi in una scatola chiusa, isolata dall'ambiente esterno. Se si apre la scatola, si rompe la fiala, avvelenando il gatto. Allo stesso tempo, il gatto non ha cibo né acqua, quindi prima o poi sarà sicuramente morto. Il paradosso è nel fatto che, aprendo la scatola, ucciderai di sicuro il gatto, ma senza aprirla non è possibile sapere se il gatto sia vivo o morto.

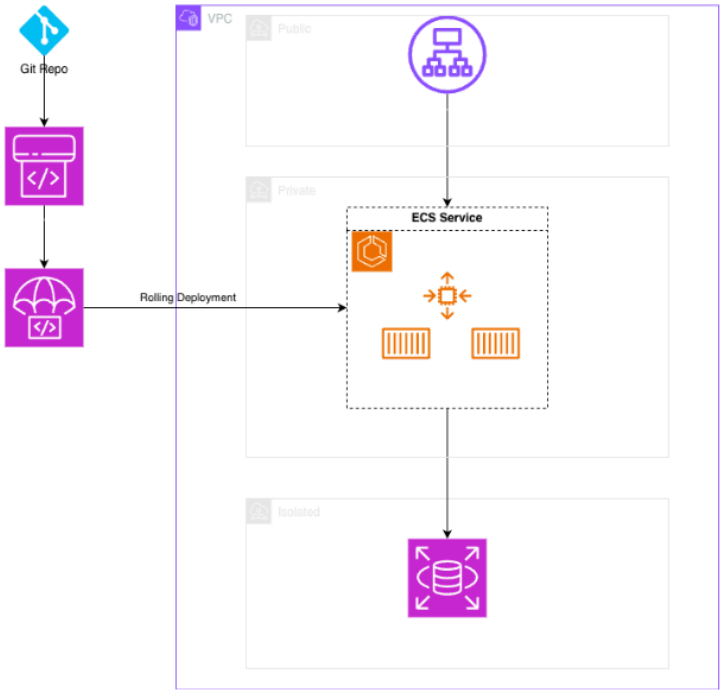
Questo può succedere anche a un *container*.

Siamo stati ingaggiati per affrontare un progetto di refactoring di un'applicazione web (il classico "breaking down the monolith") da un'istanza *EC2* a container.

Abbiamo iniziato analizzando l'applicazione facendo deploy di un ambiente di sviluppo in un account AWS Account dedicato e procedendo con una prima *containerizzazione*

dell'intera applicazione per vedere come interagiva con gli altri servizi esistenti. Dopodiché ci siamo dedicati alle *pipeline* di *CI/CD* e abbiamo impostato un approccio cloud-native.

Di seguito il diagramma "noioso" dell'infrastruttura. Visto che l'applicazione era relativamente semplice, abbiamo optato per l'utilizzo di ECS Fargate per il workload.



L'applicazione sembrava a posto, ma, dopo alcuni test iniziali, la mattina dopo il primo deployment, abbiamo visto un errore 502. Abbiamo pensato ad un problema temporaneo, visto che l'applicazione è tornata a funzionare dopo qualche minuto.

Nel pomeriggio, abbiamo notato la stessa cosa e abbiamo quindi iniziato a investigare. Abbiamo notato che il task ECS era stato riavviato diverse volte la notte prima, anche senza traffico. Il grafico della memoria indicava chiaramente un *memory l*eak. La cosa strana era che nessun utente stava facendo traffico, trattandosi di un ambiente di sviluppo. L'health check dell'Application Load Balancer era sufficiente a scatenare il memory leak. Quindi, il solo fatto di osservare il sistema lo faceva "morire", proprio come nel paradosso di Schrödinger. Il container era contemporaneamente running e morto: il vecchio memory leak dell'applicazione aveva trovato nuova vita nel cloud.

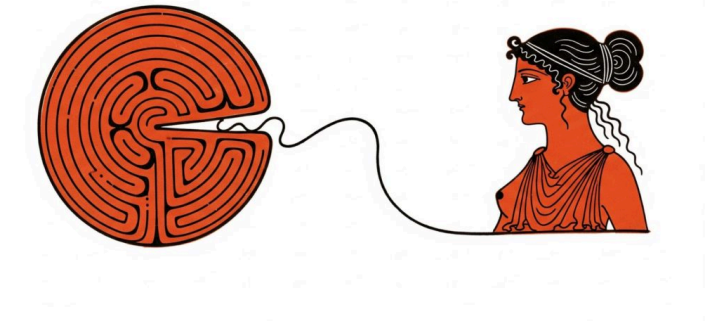
La vera tragedia non era il bug in sé, ma il fatto che la nostra architettura era diventata un sistema di delivery per memory leak. Mettere in un container un'applicazione fallata non è modernizzazione; è solo un modo per accelerarne il malfunzionamento.

Grazie, Mehmed, per l'idea e per il titolo!

Il filo di Arianna

"Arianna sussurra: “La tana del Minotauro è nel cuore stesso del labirinto. Il suono del suo respiro ti mostrerà in che direzione andare. Ecco una spada, e qui c'è un gomitolo di filo, col quale, dopo aver ucciso il mostro, potrai ritrovare la via del ritorno.”

Il Mito di Teseo e il Minotauro



Quando Teseo si trova nel labirinto del Minotauro, Arianna gli porge un filo per guidarlo e aiutarlo a ritrovare la via d'uscita.

Peccato che in questa storia noi non avessimo nessuno a darci una guida.

Questa storia inizia con un esperimento, diventato poi una grossa applicazione usata da molti grossi clienti. L'azienda è partita utilizzando un singolo account AWS per tutti gli ambienti, giusto per sperimentare. All'improvviso, l’applicazione è diventata "una cosa seria" e non c’è stato tempo per organizzarne la governance. A un certo punto, una Landing Zone è diventata necessaria in modo urgente per governare meglio le risorse AWS, isolare risorse e accessi, e rispondere alle richieste dei clienti.

Dopo aver definito la soluzione migliore per il caso d'uso, aver creato gli account AWS, aver integrato l'IdP ed aver implementato il networking centralizzato, era ora di spostare i workload dal singolo account AWS a quelli dedicati.

Non abbiamo indagato molto a fondo nell'applicazione, ma si trattava di un'applicazione distribuita in container e che seguiva un approccio a microservizi, quindi abbiamo iniziato le interviste per definire meglio le dipendenze. Visto che il tempo di esecuzione era fondamentale, abbiamo iniziato a fare deploy nel nuovo account.

Il problema è arrivato dopo il deploy dei primi database e la migrazione dei dati: ogni servizio sembrava far riferimento a qualcos'altro: un autoscaling group EC2 per ottenere i parametri di configurazione, un altro microservizio per far partire il frontend, e un batch job che interrogava il servizio preso in esame per popolare una tabella necessaria al primo servizio.

Provare a tracciare un diagramma delle dipendenze è finito per generare qualcosa che ricorda un disegno di Picasso.

L'esperimento è diventato un'applicazione, e l'applicazione è diventata una maledizione. Ci siamo resi conto in fretta che il vero labirinto non era la migrazione: era il risultato di dipendenze storiche, mancanza di documentazione e di decisioni prese "ieri". Abbiamo già visto la differenza tra container e microservice qui.

Troppo monitoring

“Dov'è la conoscenza che abbiamo perso nell'informazione?”

T.S. Eliot - Cori da "La Rocca"

Sapere come i clienti stiano fruendo un servizio e se la loro esperienza è buona è sempre una sfida; se hai un e-commerce, il più piccolo dettaglio può avere un impatto sull’esperienza utente, alterando le vendite in modo positivo o negativo.

Anche un piccolo delay durante la visita di un sito può tradursi in una perdita di soldi, quindi è cruciale trovare un modo per monitorare l'esperienza utente.

Vi presentiamo MegaCorp, che ha sviluppato un e-commerce utilizzando ogni piattaforma no-code e servizio immaginabile. Quando qualcosa andava storto, fare troubleshooting era praticamente impossibile, e capire quale componente peggiorasse le performance era un'impresa colossale.

A peggiorare le cose, i deployment erano programmati ogni 15 giorni per gli ambienti di dev e test, ed ogni mese per l'ambiente di produzione.

Per nostra fortuna, Amazon RUM semplifica l'attività, fornendo insight su ogni piccolo evento che si verifica nel browser, permettendo di monitorare i dati di performance percepiti dagli utenti, come, ad esempio, i tempi di download delle risorse e di render della pagina, e perfino monitorare le eccezioni JavaScript nella console.

Amazon RUM inserisce un piccolo JavaScript che riporta gli eventi utente a un endpoint. È buona norma testare potenziali interazioni indesiderate con gli script JavaScript esistenti in un ambiente di test. Inoltre, visto che il billing usa un modello pay-as-you-go basato sugli eventi generati, occorre trovare il giusto equilibrio tra monitorare ogni singolo utente e abilitare il sampling, selezionando il tipo di eventi da registrare.

Abbiamo iniziato quindi con un semplice ambiente di test per trovare il punto giusto e tenere tutto sotto controllo.

I nostri test non hanno riportato problemi con l'integrazione. Tuttavia, abbiamo dovuto escludere gli eventi per il caricamento delle immagini, visto che era in uso una CDN veloce e il numero di eventi sarebbe stato troppo grande.

Abbiamo notato alcuni errori JavaScript in console, ma ci hanno detto che erano dovuti al fatto che gli sviluppatori erano in procinto di rilasciare una nuova feature.

E allora, produzione sia! Dopo aver atteso 20 giorni per il deployment, RUM è andato live. E funzionava come previsto, ma...

Dopo mezza giornata di raccolta dati, abbiamo ricevuto un budget alert: la nostra soglia dell'80% del budget mensile (6000 $) era già stata raggiunta.

Abbiamo controllato i numeri: visitatori e page view erano allineati con le nostre previsioni, ma i conti non tornavano per quanto riguardava il numero di eventi. Abbiamo scoperto che ogni visita generava almeno 5 o 6 eccezioni JavaScript nella console del browser, che RUM ovviamente registrava. Gli errori erano dovuti a un bug che era stato sì sistemato, ma era stato rilasciato in produzione per colpa del ciclo di release. Non potevamo disattivare RUM perché la finestra di deploy era passata.

Per fortuna, RUM usa un Cognito Identity Pool per creare credenziali temporanee per inviare i dati all'endpoint. Disattivare Cognito ha fermato il billing.

Alla fine, un bug perfettamente monitorato è solo un bug ancora più costoso, e l'unica cosa più importante dell'avere una soluzione di monitoring è avere un processo per controllare i bug e i deployment in produzione! Quando il ciclo di deployment non riesce a stare dietro al debito tecnico, gli strumenti nati per fornire chiarezza possono trasformarsi molto velocemente in una spesa enorme senza aggiungere valore.

I soliti sospetti

"In un mondo dove niente è ciò che sembra devi guardare oltre..."

Roger Kint (Kevin Spacey) - I Soliti Sospetti

"La nostra infrastruttura è cloud-native".

Quando abbiamo sentito questa frase, eravamo felici ma anche un po' sospettosi.

Dopo una scissione tra aziende, siamo stati incaricati di occuparci della separazione di un account AWS dall'organization originale e dell'implementazione di una Landing Zone nuova fiammante. Questo significava un nuovo set di account, la creazione di un

network centralizzato, deleghe dei servizi e una nuova federazione IAM Identity Center.

Dopo l'assessment iniziale e le interviste, ci è stato chiesto di fare troubleshooting su alcuni problemi di latenza e irraggiungibilità sull'applicazione web utilizzata dagli utenti esterni ed in esecuzione in un cluster Amazon EKS.

Durante le ore di picco, la latenza sembrava aumentare fino a che l'applicazione diventava irraggiungibile per diversi minuti. Poi, senza nessun intervento, il problema sembrava risolversi da solo.

Anche se era un'applicazione con architettura a microservizi, non era stato implementato nessun meccanismo di tracing. Dopo aver implementato una soluzione di tracing, il quadro è diventato chiaro: l'applicazione, in fin dei conti, non era poi così cloud-native: solo la parte a cui accedevano gli utenti girava nativamente su Kubernetes. Tutti gli altri componenti erano ospitati in un datacenter on-premise, con una connessione VPN configurata con un singolo tunnel attivo e senza BGP. Durante le ore di picco, il traffico superava la quota massima di 1,25 Gbit del tunnel, aggiungendo latenza, ritardi e perdita di pacchetti con un effetto valanga.

L'applicazione girava pure su EKS, ma il suo spirito guida era in realtà una connessione dial-up a 56k.

La luce in fondo al tunnel

E così, è giunta l'ora di richiudere il coperchio della bara su un'altra collezione di incubi infrastrutturali, con un ultimo promemoria: adottare semplicemente l'ultima tecnologia non rende immuni dai peccati del passato.

Ma non temete! Gli orrori torneranno la prossima stagione di Halloween. Fino ad allora, che le vostre pipeline siano verdi e i vostri grafici della memoria piatti.

Abbiamo raccontato le nostre storie, ora vogliamo sentire le vostre! Quali orrori indicibili si nascondono nei vostri ambienti di produzione?

Lasciateci un commento qui sotto, scambiamoci racconti del terrore prima che le luci si spengano per sempre.

Buon Halloween, e che la vostra infrastruttura possa riposare in pace... per ora.

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!
