

Spiegazioni dettagliate — Quizzone (Concetti di architettura cloud)

Domanda 1: Qual è lo scopo principale di un sistema IAM (Identity and Access Management) nel cloud?

Risposta corretta: b) Controllare chi può accedere alle risorse e quali azioni può compiere

IAM è il framework che gestisce due aspetti fondamentali: **Identity** (chi sei — autenticazione) e **Access** (cosa puoi fare — autorizzazione). Come documentato nel Lab 01 ([docker-iam-parallels.md](#)): "IAM (Identity and Access Management) è il framework che controlla: Identity — Chi sei (autenticazione), Access — Cosa puoi fare (autorizzazione)". L'intero Lab 01 è dedicato a dimostrare questo principio attraverso la gestione di utenti Linux, gruppi e permessi sul socket Docker come parallelo diretto dei sistemi IAM cloud.

- **a) Monitorare le prestazioni delle istanze di calcolo** — Errata. Il monitoraggio delle prestazioni è compito di strumenti dedicati come CloudWatch in AWS o [docker stats](#) localmente, trattati nel Lab 03 (Compute). IAM si occupa esclusivamente di identità e controllo degli accessi, non di metriche prestazionali.
- **c) Gestire la fatturazione e i costi dei servizi cloud** — Errata. La fatturazione è un servizio separato (Billing) nei cloud provider. IAM può controllare *chi* ha accesso alla console di fatturazione, ma il suo scopo primario è la gestione delle identità e dei permessi, non la contabilità dei costi.

Domanda 2: Qual è la differenza fondamentale tra autenticazione e autorizzazione?

Risposta corretta: b) L'autenticazione verifica l'identità dell'utente, l'autorizzazione determina quali azioni può compiere

Il Lab 01 definisce esplicitamente questi due concetti come i pilastri di IAM: "Identity: Chi sei (autenticazione)" e "Access: Cosa puoi fare (autorizzazione)". Sono due processi distinti e sequenziali: prima l'utente si autentica (dimostra chi è, ad esempio con password o SSH key localmente, con access key nel cloud), poi il sistema verifica cosa è autorizzato a fare (appartenenza al gruppo [docker](#) localmente, IAM Policy nel cloud).

- **a) Sono due termini diversi per indicare lo stesso processo di login** — Errata. Autenticazione e autorizzazione sono processi separati. Un utente può autenticarsi con successo (login riuscito) ma non avere autorizzazione per una determinata azione. Nel Lab 01, un utente Linux può fare login ma se non è nel gruppo [docker](#) non può eseguire container.
- **c) L'autenticazione definisce cosa puoi fare, l'autorizzazione verifica chi sei** — Errata. I concetti sono invertiti. Questa è una trappola comune: autenticazione = identità (chi sei), autorizzazione = permessi (cosa puoi fare). La tabella del Lab 01 lo mostra chiaramente con il parallelismo utente Linux/IAM User (autenticazione) e permessi file/IAM Policy (autorizzazione).

Domanda 3: Qual è la differenza tra una subnet pubblica e una subnet privata?

Risposta corretta: b) La subnet pubblica ha un percorso verso internet, la subnet privata non è raggiungibile dall'esterno

Il Lab 02 ([docker-network-vpc-parallels.md](#)) definisce chiaramente: "Subnet pubblica: Con route verso internet (Docker: senza `--internal`)" e "Subnet privata: Senza route verso internet (Docker: con `--internal`)". Nel lab, una rete Docker creata con il flag `--internal` simula una subnet privata dove i container non possono raggiungere internet e non sono accessibili dall'host. Senza il flag, la rete simula una subnet pubblica con route verso internet.

- **a) La subnet pubblica ha più indirizzi IP disponibili rispetto alla privata** — Errata. Il numero di indirizzi IP dipende esclusivamente dal CIDR block assegnato (es. /24 = 254 host, /16 = 65534 host), non dal fatto che la subnet sia pubblica o privata. Nel corso, entrambe le subnet usano /24 con lo stesso numero di indirizzi disponibili.
- **c) La subnet pubblica è gratuita, la subnet privata ha un costo aggiuntivo** — Errata. Non esiste questa distinzione di costo nei cloud provider. Entrambi i tipi di subnet sono componenti della VPC e il costo dipende dal traffico di rete e dalle risorse utilizzate, non dalla classificazione pubblica/privata della subnet.

Domanda 4: Cosa significa che un servizio è "multi-homed" nel contesto delle reti cloud?

Risposta corretta: b) Che il servizio è connesso a più reti contemporaneamente e può comunicare su ciascuna di esse

Il concetto di multi-homed è centrale nel Lab 02 e nel Lab 05. Nel [docker-compose.yml](#) del Lab 02, l'application server è descritto come "Application Server - multi-homed (pubblica + privata)" ed è connesso a entrambe le reti `vpc-public` e `vpc-private`. Lo stesso pattern è usato nel Lab 05, dove il servizio `app` ha due indirizzi: `10.0.1.10` nella rete pubblica e `10.0.2.10` nella rete privata. Il tutorial del Lab 02 verifica esplicitamente che "Container multi-homed raggiungono entrambe le reti".

- **a) Che il servizio è replicato in più data center geograficamente distanti** — Errata. Questa descrizione corrisponde alla multi-region replication o geo-redundancy, un concetto completamente diverso. Multi-homed si riferisce alle connessioni di rete di una singola istanza, non alla sua distribuzione geografica.
- **c) Che il servizio ha più nomi DNS associati allo stesso indirizzo IP** — Errata. Questa descrizione corrisponde ai DNS aliases o al virtual hosting. Multi-homed significa avere più interfacce di rete su reti diverse, con IP diversi su ciascuna rete — come dimostrato dall'app server del Lab 05 con due IP distinti.

Domanda 5: Qual è la differenza tra scaling verticale e scaling orizzontale?

Risposta corretta: a) Lo scaling verticale aumenta le risorse di una singola istanza, quello orizzontale aggiunge più istanze

Il Lab 03 ([ec2-instance-mapping.md](#)) mostra diverse configurazioni di risorse che simulano instance types EC2: da `t2.nano` (0.5 vCPU, 512 MB) fino a `m5.4xlarge` (16 vCPU, 64 GB). Passare da un `t2.micro` a un

m5.large è un esempio di scaling verticale (scale up): la stessa istanza riceve più CPU e memoria. Lo scaling orizzontale (scale out) aggiunge invece repliche del servizio, distribuendo il carico su più istanze.

- **b) Lo scaling verticale aggiunge più istanze, quello orizzontale aumenta le risorse di una singola istanza** — Errata. I concetti sono invertiti. Verticale = "scale up" (più risorse per una singola istanza), orizzontale = "scale out" (più istanze). Un modo per ricordarlo: verticale = crescere in altezza (stessa macchina, più potente), orizzontale = crescere in larghezza (più macchine).
- **c) Lo scaling verticale riguarda la CPU, quello orizzontale riguarda la memoria** — Errata. Entrambi i tipi di scaling possono riguardare sia CPU che memoria. La distinzione non è nel *tipo* di risorsa ma nella *strategia*: aumentare le risorse di una singola istanza oppure aggiungere più istanze. Le famiglie di instance types del Lab 03 mostrano che CPU e memoria crescono insieme.

Domanda 6: Perché è importante definire le dipendenze tra servizi con una condizione di health check?

Risposta corretta: a) Per garantire che un servizio si avvii solo quando i servizi da cui dipende sono effettivamente pronti

Nel Lab 05, il servizio `app` dichiara `depends_on: db: condition: service_healthy`. Questo significa che il sistema non avvierà `app` finché il health check di `db` (che usa `pg_isready` per verificare che PostgreSQL accetti connessioni) non restituisce "healthy". Senza questa condizione, `app` potrebbe avviarsi prima che il database sia pronto, causando errori di connessione. Lo stesso pattern è usato nel Lab 03 tra `web` e `app`.

- **b) Per ridurre il consumo di risorse durante la fase di avvio dei servizi** — Errata. Le dipendenze con health check non riducono il consumo di risorse — anzi, aggiungono un overhead minimo per i controlli periodici. Il loro scopo è garantire l'ordine corretto di avvio e la readiness dei servizi prerequisito, non l'ottimizzazione delle risorse.
- **c) Per impedire che due servizi vengano eseguiti sulla stessa istanza di calcolo** — Errata. La co-locazione dei servizi è gestita da placement constraints e scheduling, non dalle dipendenze di avvio. Nel corso, tutti i servizi di ogni lab girano sullo stesso host Docker — il punto è l'ordine temporale di avvio, non la distribuzione spaziale.

Domanda 7: Qual è la differenza principale tra block storage e object storage?

Risposta corretta: a) Il block storage si monta come un disco e gestisce file system, l'object storage gestisce oggetti accessibili via API

Il Lab 04 (`storage-s3-parallels.md`) mostra entrambi i pattern. I volumi Docker (`db-data`) simulano EBS (block storage): vengono montati come percorsi nel file system del container (`/var/lib/postgresql/data`), esattamente come un volume EBS si attacca a `/dev/sdX` di un'istanza EC2. MinIO simula S3 (object storage): gli oggetti sono accessibili tramite API HTTP S3-compatible su `http://localhost:9000`, non tramite mount di file system.

- **b) Il block storage è più economico, l'object storage è più veloce** — Errata. In realtà è generalmente il contrario: l'object storage (S3) è più economico per grandi quantità di dati, mentre il

block storage (EBS) offre latenza inferiore per accesso random. Ma la distinzione fondamentale è nel *modello di accesso* (file system vs API), non nel costo o nella velocità.

- **c) Il block storage è utilizzabile solo per i database, l'object storage solo per le immagini** — Errata. Entrambi hanno molteplici casi d'uso. Il block storage può ospitare qualsiasi file system (applicazioni, log, dati generici), e l'object storage può contenere qualsiasi tipo di oggetto (documenti, backup, archivi, media). La tabella del Lab 04 mostra diversi use case per entrambi.

Domanda 8: Perché in un'architettura cloud il database viene tipicamente collocato in una rete privata?

Risposta corretta: b) Per impedire l'accesso diretto dall'esterno e ridurre la superficie di attacco

Il Lab 05 dimostra questo principio in modo esplicito. Nel `docker-compose.yml`, il database PostgreSQL è collocato esclusivamente nella rete `vpc-private` con il commento: "NESSUNA PORTA ESPOSTA - completamente privato (INF-02). RDS in VPC privata non è accessibile dall'host". Solo il servizio `app`, che è multi-homed su entrambe le reti, può raggiungere il database attraverso la rete privata. Questo è il pattern standard di sicurezza: il database non ha ragione di essere raggiungibile da internet.

- **a) Perché i database funzionano più velocemente se non hanno accesso a internet** — Errata. L'assenza di accesso internet non migliora le prestazioni del database. La velocità dipende da CPU, memoria, storage e ottimizzazione delle query — non dalla raggiungibilità internet. La motivazione è esclusivamente di sicurezza.
- **c) Perché le reti private hanno maggiore larghezza di banda rispetto alle pubbliche** — Errata. La larghezza di banda non dipende dal tipo di subnet (pubblica/privata) ma dall'instance type e dalla configurazione di rete. Sia il Lab 02 che il Lab 05 usano le stesse configurazioni di rete; la distinzione è nell'isolamento, non nelle prestazioni.

Domanda 9: Qual è il vantaggio di usare un approccio dichiarativo per definire l'infrastruttura (Infrastructure as Code)?

Risposta corretta: b) Descrive lo stato desiderato dell'infrastruttura in modo riproducibile e versionabile

Tutti i 5 laboratori del corso usano file `docker-compose.yml` come definizione dichiarativa dell'infrastruttura. Lo studente dichiara cosa vuole (servizi, reti, volumi, limiti di risorse, health check) e il sistema si occupa di *come* realizzarlo. Questi file sono versionati in git, quindi ogni modifica è tracciata, e l'infrastruttura è riproducibile da chiunque eseguendo semplicemente `docker compose up -d`. L'idempotenza è un altro vantaggio: eseguire il comando più volte produce lo stesso risultato.

- **a) Permette di scrivere meno righe di codice rispetto a un linguaggio di programmazione tradizionale** — Errata. La concisione non è il vantaggio principale e non è nemmeno sempre vera. Un file compose dichiarativo può richiedere anche più righe di uno script imperativo equivalente. Il valore reale è nella riproducibilità, nell'idempotenza e nella possibilità di versionare l'infrastruttura come codice.
- **c) Elimina completamente la necessità di monitorare i servizi dopo il deployment** — Errata. L'IaC definisce e provisiona l'infrastruttura ma non elimina il bisogno di monitoraggio operativo. I servizi

possono comunque fallire, degradarsi o avere problemi dopo il deploy — come dimostrano i health check implementati in tutti i lab del corso, che monitorano lo stato dei servizi continuamente.

Domanda 10: Quali sono i quattro principi di sicurezza infrastrutturale fondamentali per un servizio cloud?

Risposta corretta: b) Processi non privilegiati, porte esposte solo su localhost, limiti di risorse CPU/memoria, volumi con nome per la persistenza

Questi sono i quattro requisiti infrastrutturali del corso, verificati dallo script `02-security-compliance-test.sh` su tutti i lab:

- **INF-01:** Nessun container gira come root — processi non privilegiati (dimostrato nel Lab 01 con utente `labuser` e nel Lab 05 con utente `postgres` UID 70)
- **INF-02:** Porte esposte solo su `127.0.0.1` o reti private senza porte sull'host (tutti i compose usano `127.0.0.1:porta:porta`)
- **INF-03:** Tutti i container hanno limiti CPU e memoria (`deploy.resources.limits` in ogni compose)
- **INF-04:** Dati persistenti in volumi nominativi (`db-data`, `minio-data`, `test-data` — mai bind mount anonimi)

Questi principi sono applicati coerentemente in tutti e 5 i laboratori e verificati automaticamente dai test di integrazione.

- **a) Crittografia dei dati, autenticazione a due fattori, backup giornalieri, monitoraggio continuo** — Errata. Sebbene siano pratiche di sicurezza valide nel cloud in generale, non corrispondono ai quattro principi infrastrutturali insegnati nel corso (INF-01/02/03/04). Il corso si concentra sulla sicurezza a livello di container e infrastruttura locale, non su crittografia o MFA.
- **c) Firewall perimetrale, antivirus aggiornato, password complesse, log centralizzati** — Errata. Questi sono concetti di sicurezza IT tradizionale (perimetrale), non i principi infrastrutturali del corso. L'approccio del corso è basato su isolamento per design, minimo privilegio e limiti di risorse a livello di container — un paradigma di sicurezza cloud-native.