

CLOUD-BASED SOFTWARE

In precedenza il software era proprietario e confinato nei locali di chi lo possedeva. Il server era proprio, cioè acquistato e gestito dall'azienda oppure poteva essere affidato alla gestione di terzi. (**Hosting**)

In seguito si è passati a **software cloud based** dove le risorse, come i server, non sono più proprie, a vengono affittate, con la possibilità di cambiare le risorse quando ci sono nuove versioni di queste.

Il **cloud** può essere inteso come molti server remoti offerti in affitto dalle aziende proprietarie per eseguire il software su questi server e renderli disponibili ai tuoi clienti.

Questi server remoti, sono server **virtuali**, e questo permette di far eseguire su un unico nodo server più server che sono virtuali, dando all'utente la possibilità di personalizzare, secondo le esigenze, la propria macchina virtuale.

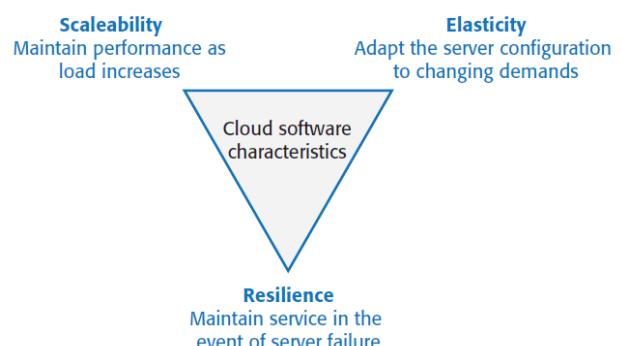
Ovviamente, per fare questo l'hardware deve essere potente, altrimenti ne risentirebbe in termini di performance.

La **scalabilità** riflette la capacità del software di gestire un numero crescente di utenti.

All'aumentare del carico sul software, il software si adatta automaticamente per mantenere le prestazioni e i tempi di risposta del sistema. I sistemi possono essere scalati aggiungendo nuovi server o migrando a un server più potente. Se si utilizza un server più potente, si parla di scalabilità verticale (scaling up). Se si aggiungono nuovi server dello stesso tipo, si parla di scalabilità orizzontale (scaling out). Se il software viene scalato orizzontalmente, copie del software vengono create ed eseguite sui server aggiuntivi.

L'**elasticità** è correlata alla scalabilità, ma consente sia la scalabilità verticale che verticale. In altre parole, è possibile monitorare la domanda sulla propria applicazione e aggiungere o rimuovere server dinamicamente al variare del numero di utenti. Ciò significa che si paga solo per i server necessari, quando servono.

Resilienza significa poter progettare l'architettura software in modo da tollerare i guasti dei server. È possibile rendere disponibili contemporaneamente più copie del software. Se una di queste si guasta, le altre continuano a fornire un servizio.



I benefici di usare questo approccio è che si evitano i costi di capitale iniziali per l'approvvigionamento dell'hardware, non si deve attendere la consegna dell'hardware prima di poter iniziare a lavorare, in quanto utilizzando il cloud, si hanno i server operativi in pochi minuti.

Se i server che stai noleggiando non sono abbastanza potenti, si può passare a sistemi più potenti, o aggiungere server per esigenze a breve termine, come i test di carico. Se hai un team di sviluppo distribuito, che lavora da sedi diverse, tutti i membri del team hanno lo stesso ambiente di sviluppo e possono condividere senza problemi tutte le informazioni.

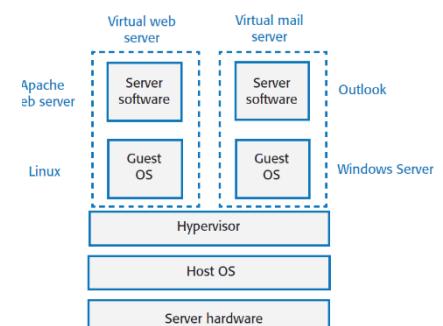
Oggi, tutti i fornitori di software offrono il proprio **software come servizio cloud**. I clienti accedono al servizio remoto tramite un browser o un'app mobile anziché installarlo sui propri computer.

Il software basato su cloud ha una architettura diversa basata su **container** per la distribuzione del software come servizio.

Virtualizzazione

Tutti i server cloud sono server virtuali. Questi sono eseguiti su un computer fisico sottostante composto da un sistema operativo e da un set di pacchetti software che forniscono le funzionalità server richieste. L'idea generale è che un server virtuale sia un sistema autonomo in grado di funzionare su qualsiasi hardware nel cloud. Questo è possibile perché il server virtuale non ha dipendenze esterne. Una dipendenza esterna significa che è necessario del software.

Le **macchine virtuali** (VM), in esecuzione su hardware server fisico, possono essere utilizzate per implementare server virtuali. L'**hypervisor** è un emulatore hardware che simula il funzionamento dell'hardware sottostante. Molti di questi emulatori hardware condividono l'hardware fisico e vengono eseguiti in parallelo. È possibile eseguire un sistema operativo e quindi installare il software server su ciascun emulatore hardware. Il vantaggio dell'utilizzo di una macchina virtuale per implementare server virtuali è che si dispone esattamente della stessa piattaforma hardware di un server fisico. È quindi possibile eseguire sistemi operativi diversi su macchine virtuali ospitate sullo stesso computer.



Il problema con l'implementazione di server virtuali su VM è che la creazione di una VM comporta il caricamento e l'avvio di un sistema operativo ampio e complesso.

Container-based virtualization

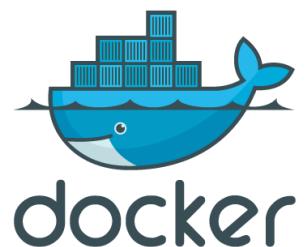
Una tecnologia di virtualizzazione più semplice e leggera chiamata **container** velocizza il processo di distribuzione di server virtuali sul cloud.

I container hanno dimensioni dell'ordine dei megabyte, mentre le VM sono dell'ordine dei gigabyte.

I container possono essere avviati e arrestati in pochi secondi anziché nei pochi minuti necessari per una VM.

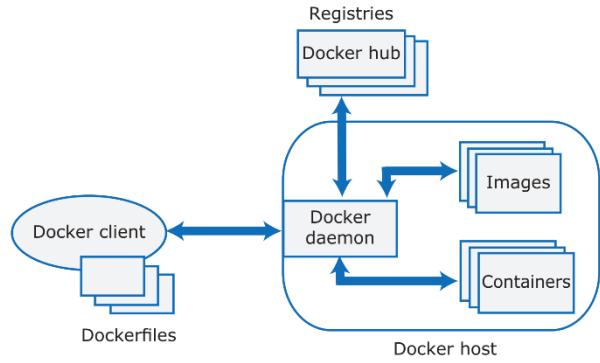
I container sono una tecnologia di virtualizzazione del sistema operativo che consente a server indipendenti di condividere un unico sistema operativo. Sono particolarmente utili per fornire servizi applicativi isolati in cui ogni utente visualizza la propria versione di un'applicazione.

Docker è un sistema di gestione dei container che consente agli utenti di definire il software da includere in un container come un'immagine Docker. Include anche un sistema di runtime in grado di creare e gestire container utilizzando queste immagini Docker.



- **Demone Docker:** processo eseguito su un server host e utilizzato per configurare, avviare, arrestare e monitorare i container, nonché per creare e gestire immagini locali.
- **Dockerfile:** i Dockerfile definiscono le applicazioni eseguibili (immagini) come una serie di comandi di configurazione che specificano il software da includere in un container. Ogni container deve essere definito da un Dockerfile associato.
- **Image:** un Dockerfile viene interpretato per creare un'immagine Docker, ovvero un insieme di directory con il software e i dati specificati installati nelle posizioni corrette. Le immagini vengono configurate per essere applicazioni Docker eseguibili.
- **Hub Docker:** registro di immagini create. Queste possono essere riutilizzate per configurare i container o come punto di partenza per la definizione di nuove immagini.
- **Container:** i container sono immagini in esecuzione. Un'immagine viene caricata in un container e l'applicazione definita dall'immagine avvia l'esecuzione. I container possono essere spostati da un server all'altro senza modifiche e replicati su più server.

Le immagini Docker sono directory che possono essere archiviate, condivise ed eseguite su diversi host Docker. L'image può fungere da file system autonomo per il server virtuale. Grazie al modo in cui Docker ha implementato il suo file system, l'image include solo i file diversi dai file standard del sistema operativo.



Docker include un meccanismo, chiamato **rete bridge**, che consente ai contenitori di comunicare tra loro. Ciò significa che è possibile creare sistemi composti da componenti comunicanti, ognuno dei quali viene eseguito nel proprio contenitore.

Vantaggi dei container

- Risolvono il problema delle dipendenze software, perché si distribuisce il prodotto (un container) che include tutto il software di supporto di cui il prodotto ha bisogno.
- Forniscono un meccanismo per la portabilità del software tra diversi cloud. I container Docker possono essere eseguiti su qualsiasi sistema o provider cloud in cui il demone Docker sia disponibile.
- Forniscono un meccanismo efficiente per l'implementazione dei servizi software e quindi supportano lo sviluppo di architetture orientate ai servizi.

Everything as a service

L'idea di un **servizio** noleggiato anziché posseduto è fondamentale per il cloud computing. Invece di possedere hardware, è possibile noleggiare l'hardware necessario da un provider cloud. Se si dispone di un prodotto software, è possibile utilizzare l'hardware noleggiato per distribuirlo ai clienti. Nel cloud computing, questo concetto si è evoluto nell'idea di "tutto come servizio".

- **Infrastructure as a Service (IaaS)** Offrono diversi tipi di servizi infrastrutturali, come servizi di elaborazione, servizi di rete e servizi di storage. Questi servizi infrastrutturali possono essere utilizzati per implementare server virtuali basati su cloud. I principali vantaggi dell'utilizzo di IaaS sono l'eliminazione dei costi di capitale per l'acquisto di hardware e la possibilità di migrare facilmente il software da un server a un server più potente. L'installazione del software sul server è responsabilità dell'utente, sebbene siano disponibili numerosi pacchetti preconfigurati per agevolare questa operazione. Utilizzando il pannello di

controllo del provider cloud, è possibile aggiungere facilmente altri server, se necessario, all'aumentare del carico del sistema.

- **Platform as a Service (PaaS)** Si utilizzano librerie e framework forniti dal provider cloud per implementare il software. L'utilizzo di PaaS semplifica lo sviluppo di software con scalabilità automatica.
- **Software as a service (SaaS)** Il prodotto software viene eseguito sul cloud ed è accessibile agli utenti tramite un browser web o un'app mobile.
- **Function as a Service (FaaS)**, fornisce una piattaforma che consente ai clienti di sviluppare, eseguire e gestire funzionalità di un'applicazione senza la complessità di dover creare e mantenere l'infrastruttura tipicamente associata allo sviluppo e al lancio di un'app. Lo sviluppo di un'applicazione secondo questo modello permette di ottenere un'architettura serverless e viene tipicamente utilizzata quando si creano applicazioni costituite da microservizi.

Una differenza tra IaaS e PaaS è l'assegnazione delle responsabilità di gestione del sistema. Se si utilizza IaaS, si ha la responsabilità dell'installazione e della gestione del database, della sicurezza del sistema e dell'applicazione. Se si utilizza PaaS, è possibile delegare la responsabilità della gestione del database e della sicurezza al provider cloud. Nel SaaS, supponendo che un fornitore di software esegua il sistema su un cloud, è il fornitore del software a gestire l'applicazione. Tutto il resto è responsabilità del provider cloud.

Software as a service

Prima i software product dovevano essere installati sui computer dei clienti che doveva configuralo per il proprio ambiente operativo e occuparsi degli aggiornamenti. Il software aggiornato non era sempre compatibile con gli altri software aziendali, quindi era comune che gli utenti utilizzassero versioni precedenti del prodotto per evitare problemi di compatibilità. Ciò significava che l'azienda produttrice del software a volte doveva gestire diverse versioni del proprio prodotto contemporaneamente.

Questo approccio esiste ancora anche se oggi, si preferisce offrire il **software come servizio**. In questo caso, il software viene eseguito sui propri server, che possono essere noleggiati da un provider cloud. I clienti non installano più il software, ma accedono al sistema remoto tramite un browser web o un'app mobile dedicata.

Vantaggi per i developer

- **Gestione degli aggiornamenti:** si ha il controllo degli aggiornamenti del prodotto e tutti i clienti li ricevono contemporaneamente evitand il problema di dover utilizzare e gestire contemporaneamente diverse

versioni. Questo riduce i costi e semplifica il mantenimento di una base di codice software coerente.

- **Distribuzione continua:** si possono distribuire nuove versioni del software non appena vengono apportate e testate le modifiche. Significa che si possono correggere rapidamente i bug, in modo che l'affidabilità del tuo software possa migliorare costantemente.

Vantaggi per i clienti

- non devono sostenere i costi di capitale per l'acquisto di server o del software stesso. Il flusso di cassa dei clienti migliora, poiché il software rappresenta un costo operativo mensile anziché una spesa in conto capitale significativa. Per mantenere l'accesso a un prodotto software basato su servizi, tuttavia, i clienti devono continuare a pagare, anche se lo utilizzano raramente. Questo a differenza del software che può essere acquistato con un pagamento una tantum. Una volta acquistato, è possibile utilizzarlo per tutto il tempo desiderato senza ulteriori pagamenti.
- accesso al software da qualunque dispositivo. L'offerta SaaS consente ai clienti di accedere al software da qualsiasi piattaforma in qualsiasi momento. Gli utenti possono utilizzare il software da più dispositivi senza doverlo installare in anticipo. Tuttavia, questo può comportare che gli sviluppatori di software debbano sviluppare app mobili per diverse piattaforme al fine di fidelizzare la propria base clienti.
- non devono assumere personale per installare e aggiornare il sistema.

Svantaggi per i clienti

- nessun controllo su quando installare gli aggiornamenti software. Se un aggiornamento introduce incompatibilità con le modalità di lavoro del cliente, questi deve apportare modifiche immediate per continuare a utilizzare il software.

Alcune aziende, in particolare le multinazionali, preferiscono non usare software cloud per motivi legati alla protezione dei dati, specialmente se gestiscono informazioni personali o finanziarie. In questi casi, può essere necessario fornire soluzioni on-premise. Il modello SaaS è ideale quando non ci sono vincoli normativi rilevanti. È fondamentale che i provider cloud rispettino la localizzazione dei dati imposta dalle normative.

Un sistema software può distribuire l'elaborazione tra client (locale) e server (remoto): quella locale migliora la reattività ma consuma più energia, un aspetto critico per dispositivi mobili. La scelta della distribuzione dipende dall'applicazione e dal suo uso.

Gli utenti devono autenticarsi su sistemi condivisi. È possibile usare un proprio sistema o un'autenticazione federata legata all'organizzazione dell'utente, per semplificare la gestione delle credenziali.

Infine, i software cloud devono essere progettati con attenzione per evitare fughe di dati tra utenti di organizzazioni diverse, un rischio di sicurezza importante.

Esistono due approcci principali per fornire un sistema software a più utenti o organizzazioni:

- **Sistema multi-tenant:** un'unica istanza del software e del database serve più clienti, mantenendo separati i loro dati a livello logico.
- **Sistema multi-istanza:** ogni cliente ha una propria istanza del software e del database, garantendo maggiore isolamento ma con costi e complessità maggiori.

Multi-tenant System

In un database multi-tenant, un singolo schema di database, definito dal provider SaaS, è condiviso da tutti gli utenti del sistema. Gli elementi nel database sono contrassegnati con un identificativo tenant, che rappresenta l'azienda che ha archiviato i dati nel sistema. Il software di accesso al database utilizza questo identificativo tenant per fornire un "isolamento logico", il che significa che gli utenti sembrano lavorare con il proprio database.

Le aziende che acquistano un software come servizio raramente desiderano utilizzare un software multi-tenant generico. Desiderano una versione del software adattata alle proprie esigenze e che offra al personale una versione personalizzata del software.

Vantaggi

Utilizzo efficiente delle risorse: Il provider SaaS ha il controllo di tutte le risorse utilizzate dal software e può ottimizzare il software per un utilizzo efficace di queste risorse.

Gestione degli aggiornamenti semplificata: È più facile aggiornare un'unica istanza di software piuttosto che più istanze. Gli aggiornamenti vengono distribuiti a tutti i clienti contemporaneamente, quindi tutti utilizzano la versione più recente del software.

Costi contenuti: I sistemi multi-tenant sono solitamente più convenienti dei sistemi multiistanza perché non richiedono risorse dedicate per ciascun tenant.

Svantaggi

Inesistenza di flessibilità: Tutti i clienti devono utilizzare lo stesso schema di database con possibilità limitate di adattare questo schema alle esigenze individuali.

Problemi di sicurezza: Poiché i dati di tutti i clienti sono archiviati nello stesso database, c'è una possibilità che i dati vengano trafugati da un cliente all'altro. Inoltre se c'è un security breach nel database, la vulnerabilità coinvolgerà tutti gli utenti.

Complessità: I sistemi multi-tenant sono generalmente più complessi dei sistemi multiistanzaa causa della necessità di gestire molti utenti.

Tutti gli utenti condividono una singola copia del sistema, fornire queste funzionalità significa che l'interfaccia utente del software e il sistema di controllo degli accessi devono essere configurabili e deve essere possibile creare **database virtuali** per ciascun cliente aziendale.

Quando un prodotto SaaS rileva che l'utente appartiene a una particolare organizzazione, ne cerca il profilo utente. Il software utilizza le informazioni del profilo per creare una versione personalizzata dell'interfaccia da presentare agli utenti. Per rilevare un utente, è possibile chiedergli di selezionare la propria organizzazione o di fornire il proprio indirizzo email aziendale.

L'interfaccia utente è progettata utilizzando elementi generici, come il nome e il logo dell'azienda. In fase di esecuzione, le pagine web vengono generate sostituendo questi elementi generici con il nome e il logo dell'azienda tratti dal profilo associato a ciascun utente. Anche i menu possono essere adattati, disabilitando alcune funzionalità se non necessarie all'attività dell'utente.

I singoli utenti sono generalmente disposti ad accettare uno schema fisso condiviso in un database multitenant e ad adattare il proprio lavoro per adattarlo a tale schema. Gli utenti aziendali potrebbero voler estendere o adattare lo schema per soddisfare le proprie specifiche esigenze aziendali. Se si utilizza un sistema di database relazionale, ci sono due modi per farlo:

1. Aggiungere diversi campi aggiuntivi a ciascuna tabella e consentire ai clienti di utilizzarli come desiderano.
2. Aggiungere un campo a ciascuna tabella che identifichi una "tabella di estensione" separata e consentire ai clienti di creare queste tabelle di estensione in base alle proprie esigenze.

È facile estendere il database fornendo campi aggiuntivi a ciascuna tabella del database e si definisce un profilo client che mappa i nomi delle colonne desiderati dal cliente a queste colonne aggiuntive.

Questo approccio presenta due problemi:

1. È difficile sapere quante colonne aggiuntive includere
2. Clienti diversi necessitino di tipi diversi di colonne

Un approccio alternativo consiste nell'aggiungere un numero qualsiasi di campi aggiuntivi e definirne i nomi, i tipi e i valori. I nomi e i tipi di questi valori sono contenuti in una tabella separata, a cui si accede tramite l'identificativo del tenant.

Un problema è la sicurezza visto che le informazioni di tutti i clienti sono archiviate nello stesso database, un bug del software o un attacco potrebbe esporre i dati di alcuni o di tutti i clienti ad altri.

Il **controllo degli accessi multilivello** implica che l'accesso ai dati deve essere controllato sia a livello organizzativo che individuale. La prima fase consiste nell'eseguire l'operazione sul database, selezionando gli elementi contrassegnati con l'identificativo dell'organizzazione. Anche i singoli utenti che accedono ai dati devono disporre delle proprie autorizzazioni di accesso quindi è necessario effettuare un'ulteriore selezione dal database per presentare solo gli elementi di dati a cui un utente identificato è autorizzato ad accedere.

La **crittografia** dei dati in un database multi-tenant assicura agli utenti aziendali che i loro dati non possono essere visualizzati da persone di altre aziende in caso di errore di sistema. Sono operazioni computazionalmente intensive che rallentano il funzionamento del database, quindi si crittografano solo i dati sensibili.

Multi-instance systems

I sistemi multi-istanza sono sistemi SaaS in cui ogni cliente dispone di un proprio sistema adattato alle proprie esigenze, inclusi database e controlli di sicurezza. I sistemi multi-istanza basati su cloud sono concettualmente più semplici dei sistemi multi-tenant ed evitano problemi di sicurezza come la fuga di dati da un'organizzazione all'altra.

Esistono due tipi di sistemi multi-istanza:

1. **Sistemi multi-istanza basati su VM** L'istanza software e il database di ciascun cliente vengono eseguiti sulla propria macchina virtuale. Tutti gli utenti dello stesso cliente possono accedere al database di sistema condiviso.
2. **Sistemi multi-istanza basati su container** Ogni utente dispone di una versione isolata del software e del database in esecuzione in un set di container. Generalmente, il software utilizza un'architettura a microservizi, con ogni servizio eseguito in un container e che gestisce il proprio database.

Vantaggi:

Flessibilità: ogni istanza del software può essere personalizzata e adattata alle esigenze del cliente. I clienti possono utilizzare schemi di database completamente diversi ed è semplice trasferire dati da un database del cliente al database del prodotto.

Sicurezza: ogni cliente ha il proprio database, quindi non vi è alcuna possibilità di perdita di dati da un cliente all'altro.

Scalabilità: le istanze del sistema possono essere scalate in base alle esigenze dei singoli clienti.

Resilienza: in caso di errore del software, probabilmente questo influenzerà solo un singolo cliente. Gli altri clienti possono continuare a lavorare normalmente.

Svantaggi:

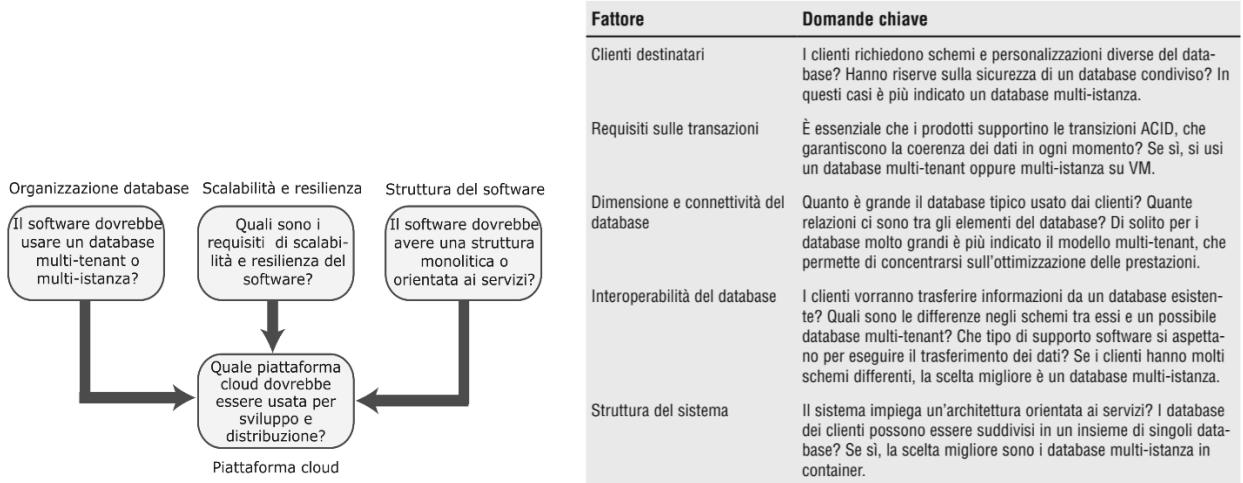
Costo: è più costoso utilizzare sistemi multi-istanza a causa dei costi del noleggio di molte VM nel cloud e dei costi di gestione di sistemi multipli. Poiché il tempo di avvio è lento, le VM potrebbero dover essere noleggiate e mantenute in esecuzione continuamente, anche se la domanda del servizio è molto bassa.

Gestione degli aggiornamenti: molte istanze devono essere aggiornate, quindi gli aggiornamenti sono più complessi, soprattutto se le istanze sono state adattate alle esigenze specifiche del cliente.

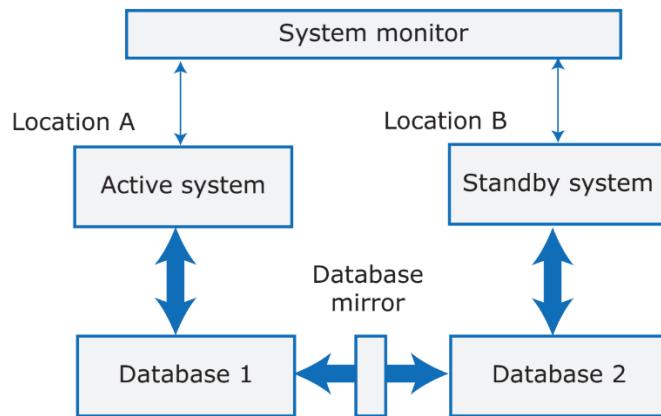
CLOUD ARCHITECTURE

Esistono tre possibili modi per fornire un database clienti in un sistema basato su cloud:

1. Come sistema multi-tenant, condiviso da tutti i clienti del tuo prodotto. Questo può essere ospitato nel cloud utilizzando server di grandi dimensioni e potenti.
2. Come sistema multi-istanza, con ogni database cliente in esecuzione sulla propria macchina virtuale.
3. Come sistema multi-istanza, con ogni database in esecuzione nel proprio contenitore. Il database cliente può essere distribuito su più contenitori.



La **scalabilità** di un sistema è la sua capacità di adattarsi automaticamente alle variazioni di carico, mentre la **resilienza** indica la capacità del sistema di continuare a fornire i servizi essenziali in caso di guasto o di utilizzo fraudolento. La scalabilità di un sistema si ottiene rendendo possibile l'aggiunta di nuovi server virtuali (**scaling out**, scalabilità orizzontale) o l'aumento della potenza di un server del sistema (**scaling up**, scalabilità verticale) in reazione alla crescita del carico. Nei sistemi su cloud, l'approccio più consueto è scalare in orizzontale, quindi il software deve essere progettato in modo da poter replicare i singoli componenti software ed eseguirli in parallelo. Le richieste saranno smistate tra le diverse istanze dei **componenti da bilanciatori di carico** hardware o software.



Varianti dell'organizzazione del sistema

1. Repliche del software e dei dati sono conservate in luoghi diversi.
2. Gli aggiornamenti del database sono replicati attraverso il mirroring, in modo che il database in standby sia una copia precisa del database attivo.
3. Un monitor di sistema verifica continuamente lo stato, e può passare automaticamente al sistema in standby se quello attivo si guasta.

Per proteggersi da guasti hardware o del software di gestione del cloud, il sistema principale e quello di backup devono essere dislocati in luoghi fisici diversi. Devono essere usati server virtuali che non si trovino fisicamente sullo stesso computer; l'ideale è che si trovino in due data center diversi. Se un server fisico si guasta, o se si verifica un guasto più ampio nel data center, l'operatività può essere trasferita subito alle copie del software conservate altrove.

Se le copie del software sono eseguite in parallelo, la commutazione può essere completamente trasparente senza alcun effetto sugli utenti. Un sistema in **hot standby**, i dati di luoghi diversi sono sincronizzati, quindi il ritardo nel passaggio al nuovo sistema è minimo. Nel sistema **cool standby** i dati vengono ripristinati da un backup, e le transazioni vengono rieseguite per aggiornare il backup e portarlo allo stato del sistema immediatamente prima del guasto.

Microservices Architecture

Scomporre il software in maniera corretta è molto importante perché le varie componenti possono essere sviluppate in parallelo da team differenti e possono essere riusati e sostituiti se le tecnologie cambiano. In questo modo sfruttiamo anche tutti i vantaggi dell'architettura cloud.

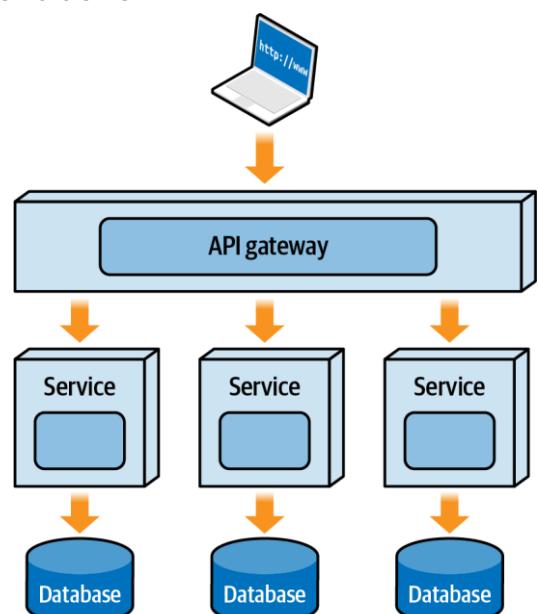
L'**architettura a microservizi** è un ecosistema composto da **servizi monouso**, distribuiti separatamente, a cui si accede in genere tramite un gateway API.

Un **servizio software** è un componente software accessibile via Internet da computer remoti che dato un input, produce un output senza effetti collaterali. Si accede al servizio attraverso la sua interfaccia resa pubblica, e tutti i dettagli relativi alla sua implementazione sono nascosti. Il gestore di un servizio è detto **provider** del servizio, l'utente è detto **requestor**.

I servizi non mantengono alcuno stato interno. Le informazioni sullo stato sono memorizzate in un database o mantenute dal richiedente. Poiché non esiste alcuno stato locale, i servizi possono essere riallocati dinamicamente da un server virtuale all'altro e possono essere replicati per rispondere all'aumento del numero di richieste del servizio, rendendo l'applicazione scalabile.

Ogni servizio è di solito associato a un database separato. Ogni servizio possiede la propria raccolta di tabelle, solitamente sotto forma di schema che può essere ospitata in un singolo database altamente disponibile o in un singolo database dedicato a un dominio specifico. Solo il servizio è proprietario delle tabelle e può accedere e aggiornare i dati. Se altri servizi necessitano di accedere a tali dati, devono richiedere tali informazioni al microservizio proprietario anziché accedere direttamente alle tabelle.

L'**API gateway** nei microservizi serve a nascondere la posizione e l'implementazione dei servizi corrispondenti che corrispondono agli endpoint dell'API gateway. Tuttavia, l'API gateway può anche svolgere funzioni trasversali relative all'infrastruttura, come sicurezza, raccolta di metriche, generazione di ID di richiesta e così via. L'API gateway nei microservizi **NON** contiene logica di business, né esegue alcuna orchestrazione o mediazione. Questo è fondamentale all'interno dei microservizi per preservare quello che è noto come **boundend context**



cioè che ogni microservizio gestisce una specifica parte del dominio aziendale, con le proprie regole e logiche. Tale separazione consente:

- **Autonomia:** Ogni microservizio può essere sviluppato, distribuito e scalato indipendentemente.
 - **Chiarezza:** I confini tra i diversi ambiti del sistema sono ben definiti, riducendo le interdipendenze.
 - **Manutenibilità:** Le modifiche in un microservizio hanno un impatto limitato, facilitando l'evoluzione del sistema.
-

SOA

Questa idea non è nata di recente, ma esisteva già negli anni '90 con l'architettura **SOA Service-oriented architecture** in cui ogni servizio **indipendente** dagli altri ed esponeva le proprie funzionalità tramite **interfacce standard**. Il vantaggio è che tutto ciò rendeva il software **interoperabile**, può funzionare su piattaforme diverse.

Ciò è reso possibile grazie all'utilizzo di protocolli standard come HTTP, SOAP o REST e formati di scambio dati come XML o JSON, che garantiscono una comunicazione uniforme e piattaforma-indipendente.

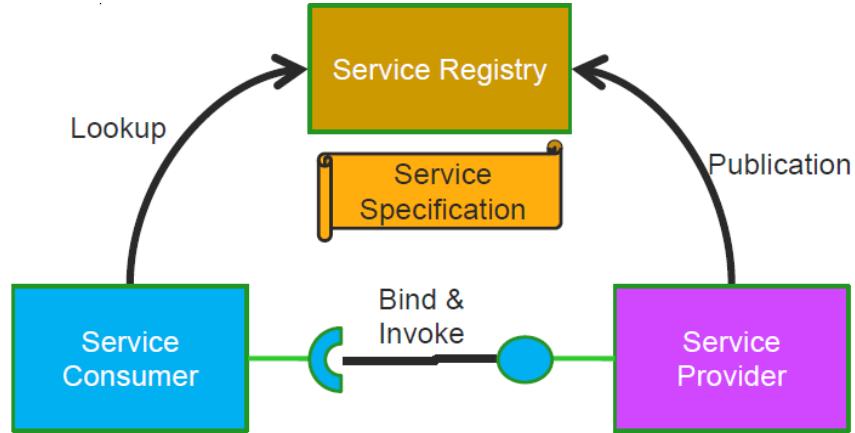
Un elemento fondamentale che permette il funzionamento efficace e standardizzato di questa architettura è il **WSDL Web Services Description Language** che permetteva ai vari servizi di poter comunicare tra loro, indipendentemente da chi li ha sviluppati o da quale tecnologia utilizzano. Descrive cosa fa un servizio, quali operazioni espone, quali dati si aspetta in input e cosa restituisce in output, descrivendo le interfacce in modo indipendenti dalla implementazione effettiva. Poi in una seconda parte del WSDL c'erano le varie implementazioni concrete dei vari protocolli di comunicazione.

Il WSDL è composto da diverse sezioni che, nel loro insieme, forniscono una descrizione completa del servizio. Tra queste troviamo la definizione dei **tipi di dati utilizzati** (grazie a XML Schema), dei **messaggi** che rappresentano le richieste e le risposte, delle **operazioni** disponibili (cioè le "funzioni" del servizio), e infine delle modalità tecniche di comunicazione, come il protocollo da usare (spesso SOAP) e l'indirizzo di rete dove il servizio è raggiungibile.

Il funzionamento di SOA si basa su tre passaggi fondamentali: **pubblicazione, scoperta e invocazione**.

1. Ogni servizio viene prima pubblicato in un **registro** che funge da catalogo, dove è descritto cosa fa e come può essere utilizzato.
2. Le applicazioni che hanno bisogno di quel servizio possono cercarlo in questo registro, scoprirne l'interfaccia e

3. poi invocarlo per utilizzarne le funzionalità. Questo processo avviene in modo dinamico e trasparente, e può essere gestito da un componente centrale chiamato **Enterprise Service Bus (ESB)**, che si occupa di mediare, trasformare e instradare i messaggi tra i vari servizi.



Un **microservizio** è un'unità software monofunzionale, distribuita separatamente, che svolge una sola funzione. I microservizi possono essere distribuiti come servizi containerizzati o come funzioni serverless. I microservizi comunicano scambiandosi messaggi. Un messaggio inviato tra servizi include alcune informazioni utili alla gestione del messaggio, una richiesta di servizio e i dati necessari a fornire il servizio richiesto. I servizi restituiscono una risposta ai messaggi di richiesta di un servizio, e questa risposta contiene a sua volta i dati che rappresentano la risposta alla richiesta del servizio.

L'obiettivo nel progettare un microservizio dovrebbe essere la creazione di un servizio che abbia **coesione forte** e **accoppiamento debole**.

L'accoppiamento è una misura del numero di relazioni che un componente ha con altri componenti del sistema.

Un accoppiamento debole significa che i componenti non hanno molte relazioni con altri componenti.

La **coesione** è una misura del numero delle relazioni che le singole parti del componente hanno tra loro.

Una coesione forte significa che tutte le parti necessarie a fornire la funzionalità del componente sono incluse nel componente stesso

L'accoppiamento debole è importante nei microservizi perché conduce a servizi indipendenti. Fintanto che se ne mantiene l'interfaccia, un servizio può essere aggiornato senza bisogno di modificare altri servizi del sistema. La coesione forte è importante perché significa che il servizio non deve chiamare molti altri

servizi durante l'esecuzione. La chiamata di altri servizi implica un overhead di comunicazione, che può rallentare il sistema.

PRINCIPIO SINGOLA RESPONSABILITÀ

Ciascun elemento di un sistema dovrebbe fare una sola cosa, e farla bene

Ma questo non vuol dire che un microservizio deve svolgere una sola funzionalità. Infatti è difficile definire cosa si intenda con “cosa sola”.

Quando i dati utilizzati in un microservizio sono usati anche da altri servizi, è necessaria la **gestione della consistenza dei dati**, deve esistere un modo per comunicare gli aggiornamenti dei dati tra i servizi, e garantire che le modifiche apportate in un servizio siano recepite da tutti i servizi che utilizzano i dati.

L'architettura a microservizi punta a risolvere due problemi di quella monolitica:

1. dopo qualsiasi modifica è necessario effettuare nuovamente la build dell'intero sistema, il suo testing e il suo deployment. Questo processo può essere lento, dal momento che le modifiche a una parte del sistema possono influire negativamente su altri componenti. Risulta quindi impossibile aggiornare frequentemente l'applicazione.
2. Se le richieste al sistema aumentano, è necessario scalare l'intero sistema anche se le richieste sono limitate a un piccolo numero di componenti del sistema che implementano le funzioni più usate.. A seconda di come viene gestita la virtualizzazione, l'avvio di un server grande può richiedere diversi minuti; il servizio del sistema è degradato fino a quando il nuovo server non è attivo e funzionante.

I microservizi sono autocontenuti ed eseguiti in processi separati. Nei sistemi su cloud, ciascun microservizio può essere rilasciato in un container proprio. Ciò significa che un microservizio può essere arrestato e riavviato senza influire su altre parti del sistema. Se la domanda di un servizio aumenta, è possibile creare e dislocare rapidamente delle repliche del servizio; esse non richiedono un server più potente, quindi scalare in orizzontale di solito costa meno del farlo in verticale

DECISIONI DI PROGETTAZIONE



Come scomporre il sistema

Troppi microservizi significano troppe comunicazioni tra servizi, e il tempo necessario per elaborarle rallenta il sistema. Se i microservizi sono troppo pochi, ogni servizio dovrà avere più funzionalità; i servizi saranno più grandi, con più dipendenze, e risulterà più difficile modificarli.

1. Bilanciare funzionalità a grana fine e prestazioni del sistema

Nei servizi a singola funzione, le modifiche sono normalmente limitate a pochi servizi. Se tutti i servizi offrono solo un unico servizio molto specifico, però, è inevitabile la necessità di un gran numero di comunicazioni tra servizi per implementare la funzionalità per l'utente. L'eccesso di messaggi rallenta il sistema, perché ciascun servizio deve impacchettare e spacchettare i messaggi inviati da altri servizi.

2. Principio della chiusura comune

Significa che gli elementi di un sistema che potrebbero essere modificati nello stesso momento dovrebbero trovarsi nello stesso servizio. La maggior parte dei requisiti nuovi e di quelli modificati dovrebbe così interessare un solo servizio.

3. Associare servizi a competenze di business

Una competenza di business indica un insieme discreto di funzionalità di business la cui responsabilità è di un singolo individuo o di un gruppo. Ad esempio, il fornitore di un sistema di stampa fotografica avrà un gruppo responsabile di inviare le foto agli utenti (competenza relativa all'invio), un insieme di stampanti (competenza relativa alla stampa), un responsabile della contabilità (servizio di pagamento), e così via. È bene

identificare i servizi che sono necessari per supportare ciascuna competenza di business.

4. **Progettare servizi che accedono solo ai dati di cui hanno bisogno**

Nelle situazioni in cui c'è una sovrapposizione tra dati usati da servizi diversi, è necessario un meccanismo che garantisca che le variazioni nei

Comunicazioni tra servizi

I servizi comunicano scambiandosi messaggi che contengono informazioni sul mittente e i dati che costituiscono l'input della richiesta o l'output della risposta. I messaggi scambiati sono strutturati seguendo un protocollo che definisce che cosa deve essere incluso in ciascun messaggio e come identificare ogni componente del messaggio.

Nel progettare un'architettura a microservizi è necessario stabilire uno standard di comunicazione che deve essere seguito da tutti i microservizi.

1. L'interazione dei servizi dovrebbe essere sincrona o asincrona?
2. I servizi dovrebbero comunicare direttamente o attraverso un middleware che funga da broker dei messaggi?
3. Quali protocolli dovrebbero essere usati per lo scambio di messaggi tra i servizi?

In un'interazione sincrona, il servizio A emette una richiesta per il servizio B; il servizio A quindi sospende l'elaborazione mentre il servizio B elabora la richiesta, e attende che il servizio B restituisca l'informazione richiesta prima di proseguire l'esecuzione dati in un servizio siano propagate agli altri servizi che usano gli stessi dati

In un'interazione asincrona, il servizio A emette la richiesta, che viene messa nella coda di elaborazione del servizio B. Il servizio A quindi continua a elaborare senza aspettare che il servizio B termini la sua elaborazione. Qualche tempo dopo, il servizio B completa la richiesta precedente del servizio A e accoda il risultato che il servizio A dovrà prelevare. Il servizio A, quindi, deve controllare periodicamente la sua coda per vedere se il risultato è disponibile.

In un'interazione asincrona, il servizio A emette la richiesta, che viene messa nella coda di elaborazione del servizio B. Il servizio A quindi continua a elaborare senza aspettare che il servizio B termini la sua elaborazione. Qualche tempo dopo, il servizio B completa la richiesta precedente del servizio A e accoda il risultato che il servizio A dovrà prelevare. Il servizio A, quindi, deve controllare periodicamente la sua coda per vedere se il risultato è disponibile. Il broker è poi responsabile di trovare il servizio che può esaudire la richiesta di servizio.

Per la comunicazione indiretta è necessario del software aggiuntivo (un broker di messaggi), ma i servizi vengono richiesti per nome anziché per URI. Il broker di messaggi trova l'indirizzo del servizio richiesto e gli inoltra la richiesta. Ciò è particolarmente utile quando esistono diverse versioni di un servizio.

Il servizio richiedente non ha bisogno di conoscere la specifica versione utilizzata; per default, il broker di messaggi può inviare le richieste alla versione più recente.

I broker di messaggi, instradano una richiesta di servizio al servizio giusto, e possono anche gestire la traduzione dei messaggi da un formato a un altro. Un servizio che accede a un altro servizio attraverso un broker di messaggi non ha la necessità di conoscere dove si trova quel, né il formato che usa per i messaggi.

I broker di messaggi possono supportare interazioni sincrone e asincrone; un servizio richiedente invia semplicemente la richiesta di servizio al broker di messaggi, dopodiché può fermarsi ad aspettare una risposta o continuare a elaborare. Quando la richiesta di servizio è completa, il broker di messaggi si occupa di garantire che la risposta sia nel formato giusto, e informa il servizio richiedente che è disponibile

Un protocollo di messaggistica è un accordo tra servizi che stabilisce come debbano essere strutturati i messaggi scambiati. La definizione del protocollo stabilisce quali dati devono essere inclusi in un messaggio e come devono essere organizzati.

Forma di dati e come vengono scambiati

I microservizi dovrebbero gestire dati propri che, in una situazione ideale, dovrebbero completamente indipendenti, e se un servizio modifica dei dati non sarebbe necessario propagare tale modifica ad altri servizi. Ovviamente nella realtà, questo non è vero.

Il problema principale è garantire la **consistenza**.

1. Occorre isolare i dati all'interno di ciascun servizio del sistema, con il minor numero possibile di dati condivisi.
2. Se la condivisione dei dati è inevitabile, i microservizi dovrebbero essere progettati in modo che la maggior parte della condivisione sia in sola lettura, e che l'aggiornamento dei dati sia responsabilità di un numero limitato di servizi.
3. Se nel sistema ci sono servizi replicati, è necessario un meccanismo che mantenga consistenti le copie del database utilizzate dalle repliche

Essendo che usiamo i database e questi sono gestiti dai DBMS, questo garantisce la consistenza tramite transazione ACID, n modo che per ogni

operazione di modifica venga svolta in modo completo per mantenere la consistenza dei dati.

Quando si utilizza un'architettura a microservizi è difficile implementare in modo efficiente questo tipo di transazioni, a meno che si possano confinare i dati coinvolti entro un singolo microservizio. Questo però significa quasi certamente infrangere la regola secondo la quale un microservizio dovrebbe avere un'unica responsabilità. I sistemi che impiegano microservizi devono essere progettati in modo da tollerare un qualche grado di inconsistenza dei dati. È necessario gestire due tipi di incoerenza:

1. **Inconsistenza di dati dipendenti** Le azioni o i guasti di un servizio possono causare l'incoerenza dei dati gestiti da un altro servizio.
2. **Inconsistenza delle repliche** È possibile che siano in esecuzione concorrente più repliche dello stesso servizio. Tutte le repliche hanno una propria copia del database e ogni replica aggiorna la propria copia dei dati. È necessario un modo per rendere “consistenti alla fine” tutti questi database, così che tutte le repliche operino sugli stessi dati

Per risolvere il problema delle inconsistenze di dati dipendenti si può usare una **transazione compensativa**, cioè una transazione che annulla l'operazione precedente. Quando viene rilevato il fallimento del servizio può essere creata una transazione compensativa che riporta il sistema allo stato consistente prima dell'azione. Le transazioni compensative, però, non garantiscono l'assenza di problemi.

Mentre per il problema delle repliche è necessario che ogni servizio possa aggiornare il database. Si può impiegare un approccio di **consistenza finale** cioè che il sistema garantisce che “alla fine” i database torneranno consistenti. È possibile implementarla mantenendo un registro (log) delle transazioni. Quando un servizio modifica il database, la modifica viene registrata in un elenco chiamato “**aggiornamenti in sospeso**”. Gli altri servizi controllano questo elenco, aggiornano i propri database e segnalano di averlo fatto. Una volta che tutti hanno aggiornato, la modifica viene rimossa dal registro. Quando un servizio riceve una richiesta, prima controlla se ci sono aggiornamenti da applicare, li esegue se necessario e poi inizia l'elaborazione. In alternativa, gli aggiornamenti possono essere applicati nei momenti di **basso carico** del sistema.

Coordinamento

Ogni microservizio è autonomo e si occupa di una parte specifica del sistema, è necessario un meccanismo che consenta loro di lavorare insieme in modo

coerente per svolgere una certa iterazione chiesta dell'utente, si parla di **workflow**.

Nell'**orchestrazione**, un componente centrale, chiamato **orchestratore** – che controlla l'intero flusso del processo e decide chi fa cosa, in quale ordine, e quando. I singoli servizi non conoscono il processo nel suo insieme: eseguono semplicemente le operazioni che vengono loro richieste.

Questo approccio è utile quando il flusso del processo è complesso, ben definito e deve essere monitorato da un punto centrale. Tuttavia, ha anche alcuni svantaggi: il componente centrale può diventare un collo di bottiglia o un single point of failure, e il sistema risulta meno flessibile ai cambiamenti.

Nella **coreografia**, ogni microservizio reagisce agli eventi che accadono nel sistema e compie la propria parte in base a regole prestabilite. Il **controllo è distribuito**, e i servizi sono più autonomi e loosely coupled. Questo rende il sistema più flessibile e scalabile, ma richiede una progettazione più attenta per evitare comportamenti inattesi, soprattutto quando i processi diventano complessi. Inoltre, monitorare e tracciare il flusso di un processo distribuito tra molti eventi può essere più difficile rispetto all'orchestrazione.

Gestione dei fallimenti

Prima o poi qualcosa del sistema va a farsi fottere.

1. **Fallimento del servizio interno** Una condizione rilevata dal servizio che può essere segnalata al richiedente in un messaggio d'errore
2. **Fallimento del servizio esterno** Un fallimento con una causa esterna che influisce sulla disponibilità di un servizio. Il fallimento può far sì che il servizio smetta di rispondere ed è necessario intraprendere delle azioni per riavviarlo.
3. **Fallimento prestazionale del servizio** Le prestazioni del servizio degradano fino a un livello inaccettabile. La causa può essere un carico eccessivo o un problema interno del servizio. Per rilevare fallimenti legati alle prestazioni e servizi che non rispondono può essere utilizzato un monitoraggio esterno dei servizi.

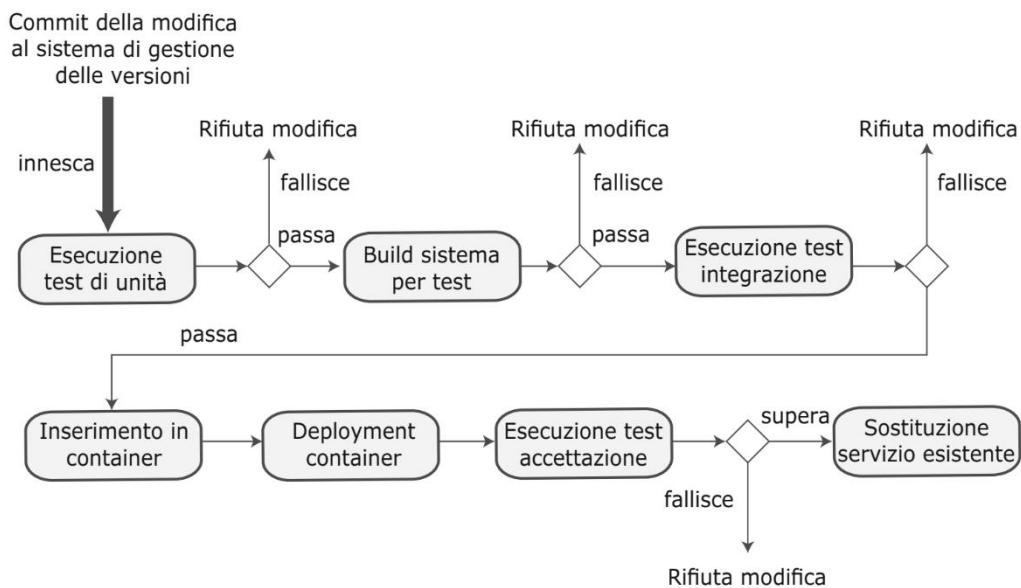
DEPLOYMENT

Dopo essere stato sviluppato e consegnato, un sistema deve essere distribuito (deployed) sui server, monitorato per intercettare eventuali problemi e aggiornato man mano che si rendono disponibili nuove versioni.

In un'architettura a microservizi, la distribuzione è più complessa rispetto ai sistemi monolitici perché ogni servizio può usare tecnologie diverse e cambiare

frequentemente. Per evitare colli di bottiglia nel deployment, è ormai comune che i team di sviluppo gestiscano anche la distribuzione e il funzionamento dei propri servizi, seguendo l'approccio **DevOps**, che unisce sviluppo e operazioni. I team di sviluppo ha la totale responsabilità del suo servizio, inclusa quella di decidere quando distribuirne le nuove versioni. È buona pratica adottare una **politica di continuous deployment** cioè appena una modifica a un servizio è stata apportata e convalidata, viene effettuato il deployment del servizio modificato.

Il deployment continuo richiede l'automazione: appena una modifica è definitiva, viene avviata una serie di attività automatizzate per il test del software. Se il software supera i test, entra in un'altra pipeline automatizzata che lo pacchettizza e lo distribuisce.



TESTING

Il testing in tempi passati veniva fatto a valle dello sviluppo del progetto software per verificare il corretto funzionamento del sistema. Oggi invece non è più così e il testing svolge un'altra funzione.

I test si possono progettare anche prima di iniziare a sviluppare il sistema basandosi sul quanto definito in precedenza.

In passato si poteva dire che se un test, dati certi input, restituiva degli output che era uguali a quelli previsti (**oracoli**), allora il sistema funzionava.

SFATIAMO MITI

La correttezza del codice non prova che il sistema funzioni correttamente
SHOCK

Quello che fa il testing, non è altro che eseguire il sistema in condizioni controllate che vengono ben definite per mostrare le possibili situazioni in cui il sistema si potrebbe trovare.

In situazioni diverse che non sono state pensate, il sistema non si sas come funziona.

Quindi l'obiettivo del testing non è provare il funzionamento del sistema, ma farlo funzionare in certe condizione per individuare gli errori.

Se il test individua degli errori allora il test ha avuto successo.

Effettuando i testi siamo i grado di individuare i **bugs/defects** che possono portare ad uno scorretto funzionamento. correggere. Le cause dei bug nei programmi possono essere due:

1. **Errori di programmazione** Il programmatore ha inavvertitamente inserito dei difetti nel codice.
2. **Errori di comprensione** Il programmatore ha frainteso o non è stato informato su alcuni dettagli relativi a ciò che il sistema dovrebbe fare.

Alcuni di questi bugs possono portare a delle **failure** del sistema, incapacità del sistema di non funzionare correttamente e che portano ad una violazione dei requisiti del funzionamento del sistema.

Tutto ciò origina degli **errori** commessi in qualche artefatto del progetto e questi non si individuano con il testing, ma effettuando una revisione di questi.

Il testing non si concentra principalmente sugli errori di programmazione, ma sui **problemi di carattere logico o di comprensione**. L'obiettivo è anche quello di testare il sistema nel suo complesso, perché scomponendo il componenti è

possibile che la componente presa singolarmente funzioni, ma insieme alle altre no. 😞.

Per affrontare i problemi nel software adottiamo tre strategie principali:

1. **Fault detection**: scoprire quanti più errori possibile
2. **Fault avoidance**: usare metodologie **per evitare l'insorgere di errori**
3. **Fault tolerance**: programmare sapendo che ci sono errori, ma minimizzando i bug

SFATIAMO MITI PT.2

1. **È impossibile testare tutto** Questo è un problema sia pratico (il numero di prove sarebbe infinito) sia teorico.
2. Il testing può mostrare la presenza di errori, non la loro assenza (Dijkstra)
3. Fare testing cost tempo e denaro, vanno progettati

Per progettare il testing è necessario stabilire delle **strategia di testing** sulla base dei mondo dei dati e dagli obiettivi. Le strategie usate sono

- **black box**, sono testi di tipo comportamentale. Questi possono essere progettati anche se avere l'implementazione effettiva del sistema, ci basa sapere cosa fa per trasformare l'input e produrre un risultato;
- **white box**, sono test strutturali che dipendono da come si è implementato il codice

TIPI DI TESTING

Funzionale Verificare la funzionalità dell'intero sistema. L'obiettivo del testing funzionale è scoprire il maggior numero possibile di bug nell'implementazione del sistema, e fornire prove convincenti che il sistema sia adatto allo scopo previsto.

Utente Verificare che il prodotto software sia utile e utilizzabile per l'utente finale. Va dimostrato che le feature del sistema aiutino l'utente a fare ciò che desidera con il software. Va anche dimostrato che gli utenti capiscano come accedere alle feature del software, e che riescano a usarle in modo efficace.

Prestazioni e carico Verificare che il software funzioni velocemente e riesca a gestire il carico degli utenti previsto sul sistema. Va dimostrato che i tempi di risposta ed elaborazione del sistema siano accettabili per l'utente finale. Va anche dimostrato che il sistema possa gestire carichi diversi e che scali senza problemi all'aumentare del carico.

Sicurezza Verificare che il software mantenga la sua integrità e possa proteggere le informazioni degli utenti da furti e danni.

Il testing utente può essere organizzato in due fasi:

1. **Alpha testing** Gli utenti lavorano insieme agli sviluppatori per testare il sistema. L'obiettivo è scoprire se gli utenti desiderano davvero le feature pianificate per il prodotto. Idealmente gli utenti dovrebbero essere coinvolti nello sviluppo già nelle fasi iniziali, in modo da ottenere un feedback sull'utilità delle feature del prodotto.
2. **Beta testing** Vengono distribuite agli utenti le prime versioni del software per raccogliere commenti. Il beta testing risponde anche agli interrogativi sull'utilità delle feature, ma di solito è più focalizzato sull'usabilità del prodotto e sull'efficacia del suo funzionamento nell'ambiente in cui opera l'utente

TEST DI FUNZIONALITÀ

Il testing del software è un'attività in più fasi, che prevede inizialmente il testing di singole unità di codice. Le unità di codice vengono poi integrate con altre unità, allo scopo di creare unità più grandi da testare a loro volta. Il procedimento continua fino a quando si crea un sistema completo pronto per il rilascio.

- **Test di unità** Servono a verificare il funzionamento corretto di **singole unità di codice**, come funzioni o metodi, in isolamento. Servono a controllare che ogni componente elementare del software si comporti come previsto. Sono scritti di solito dagli sviluppatori e sono automatizzati.
- **Test delle feature** Verificano che **una funzionalità completa del sistema** funzioni correttamente, coinvolgendo più moduli o componenti. Spesso vengono chiamati anche **test di integrazione**, soprattutto quando testano l'interazione tra moduli diversi.
- **Test del sistema** Valutano il comportamento dell'intero software come un **sistema integrato**, verificando che tutte le funzionalità lavorino insieme secondo i requisiti specifici. Si eseguono in un ambiente il più possibile simile a quello di produzione.
- **Test di release** Sono eseguiti **prima della distribuzione del software** agli utenti finali per assicurarsi che la versione da rilasciare sia stabile, funzionante e conforme ai requisiti.

TEST DI UNITÀ-WHITEBOX TESTING

Il white box si basa sul concetto di **coverage**, ovvero la **copertura del codice** che indica quanta parte del codice è stata effettivamente eseguita dai test automatici.

Esistono diversi tipi di copertura.

- **statement coverage**, che verifica se ogni istruzione è stata eseguita almeno una volta.
- **branch coverage** analizza invece le diramazioni logiche, assicurandosi che ogni possibile ramo sia stato percorso.
- **path coverage**, mira a testare ogni possibile percorso di esecuzione all'interno del codice.

Non è detto che con una copertura del 100% abbiamo la correttezza assoluta del codice. Per individuare i casi di test basta guardare il codice. Per comprendere meglio il tipo di copertura e minimizzare il numero di test da fare, si usano i grafici del flusso di controllo.

TEST DI UNITÀ-BLACK BOX TESTING

Essendo che nel black-box, non si ha accesso al codice, l'attenzione è rivolta al comportamento del sistema rispetto agli input e agli output. In questo contesto, si parla più spesso di **copertura dei requisiti** o **copertura funzionale**, poiché ciò che conta è verificare che tutte le funzionalità previste siano correttamente implementate e accessibili all'utente. Essendo che è impossibile testare tutti i possibili casi, si utilizza un approccio detto a **classi di equivalenza**, cioè troviamo delle partizioni di equivalenza che rappresentano un insieme di input che sono trattati allo stesso modo.

Le partizioni identificate non dovrebbero includere solo quelle contenenti input che producono risultati corretti; è opportuno identificare anche **partizioni di non correttezza** nelle quali gli input siano deliberatamente sbagliati. Serviranno a testare che il programma rilevi e gestisca nel modo atteso gli input non corretti

Una volta identificate le partizioni di equivalenza, è bene chiedersi quali siano gli input di ciascuna partizione che diano maggiori probabilità di scoprire dei bug.

Linee Guida

Casi limite di test Se la partizione ha limiti superiori e inferiori (ad esempio, lunghezza di stringhe, numeri, ecc.), scegli gli input ai limiti dell'intervallo.

Errori di forzatura Scegli input di test che costringano il sistema a generare tutti i messaggi di errore. Scegli input di test che dovrebbero generare output non validi.

Buffer di riempimento Scegli input di test che causino l'overflow di tutti i buffer di input.

Ripetizione Ripeti lo stesso input di test o serie di input più volte.

Overflow e underflow Se il programma esegue calcoli numerici, scegli input di test che gli facciano calcolare numeri molto grandi o molto piccoli.

Non dimenticare null e zero Se il programma utilizza puntatori o stringhe, esegui sempre il test con puntatori e stringhe null. Se utilizzi sequenze, esegui il test con una sequenza vuota. Per gli input numerici, esegui sempre il test con zero.

Mantieni il conteggio Quando gestisci liste e trasformazioni di liste, tieni il conteggio del numero di elementi in ciascuna lista e verifica che siano coerenti dopo ogni trasformazione.

Uno è diverso Se il tuo programma gestisce sequenze, esegui sempre il test con sequenze che hanno un singolo valore.

Il **boundary test** si concentra sui **valori ai bordi** di un intervallo valido, anziché solo all'interno. In altre parole, invece di testare solo "valori normali", si testano:

- Il valore minimo e massimo validi
- I valori immediatamente fuori dai limiti (uno sotto e uno sopra)

TEST DI INTEGRAZIONE

In generale, una feature fa più cose ed è implementata da più unità di programma che interagiscono che potrebbero essere implementate da sviluppatori diversi. Tutti dovrebbero essere coinvolti nel processo di test della feature. Il processo dovrebbe includere due tipi di test:

1. **Test di interazione** Verificano le interazioni tra le unità che implementano la feature. L'integrazione può anche rivelare bug nelle unità che non sono stati rilevati dai test di unità.
2. **Test di utilità** Verificano che la feature implementi ciò che gli utenti potrebbero desiderare.

Un buon modo di organizzare il testing delle feature è basarlo su uno scenario o su un insieme di user story. Il testing delle feature è una parte integrante dello sviluppo guidato dai comportamenti (**behavior-driven development, BDD**) di un prodotto, specificato da un linguaggio caratteristico del dominio, e i test delle feature sono derivati automaticamente da questa specifica. Esistono strumenti speciali per automatizzare questi test.

Spesso i vari componenti (o moduli) non vengono sviluppati tutti contemporaneamente e può capitare di dover testare un modulo anche se le sue altre parti di cui ha bisogno non sono ancora disponibili. Si costruiscono due categorie di moduli

Gli **stub** sono dei "finti moduli" che **sostituiscono un componente che viene chiamato** dal modulo che stai testando. In altre parole, se il tuo modulo ha bisogno di ricevere informazioni o servizi da un altro modulo non ancora pronto, lo puoi sostituire con uno stub. Questo stub non fa davvero tutto ciò che farebbe

il modulo originale, ma risponde in modo controllato, restituendo magari dati statici o simulati, giusto per permetterti di testare il flusso di integrazione.

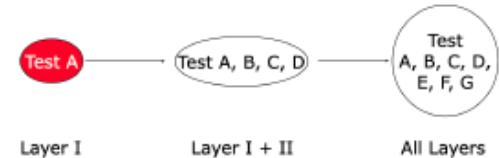
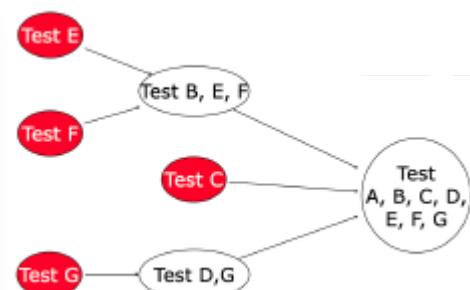
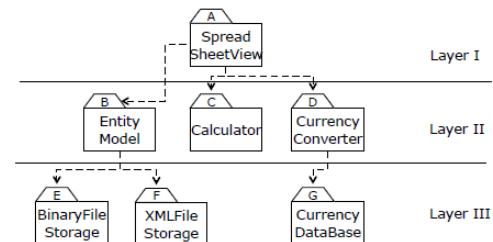
I **driver**, invece, simulano **il modulo che chiama** quello che vuoi testare. Li usi quando il modulo da testare è pronto, ma chi dovrebbe usarlo non lo è ancora. Il driver serve quindi per **attivare** il modulo testato, inviargli dei dati, chiamare le sue funzioni, ecc.

Strategia Bottom-Up

Questa strategia parte "dal basso", cioè dai moduli più di basso livello che magari gestiscono operazioni tecniche come il salvataggio dei dati, l'accesso al database, o il calcolo di valori.

Una volta che questi moduli funzionano, si procede verso l'alto, integrando moduli sempre più complessi, fino ad arrivare a quelli di alto livello, come la logica di business o l'interfaccia utente.

Il problema è che, all'inizio, i moduli superiori non ci sono ancora. Allora si usano dei **driver**.

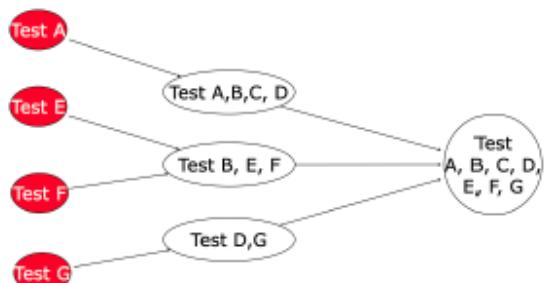


Strategia Top-Down

In questo caso si parte invece "dall'alto", cioè dai moduli principali, come l'interfaccia grafica o il sistema di controllo generale. Questi moduli, però, per funzionare, chiamano altri moduli che magari non sono ancora pronti. Si usano gli **stub**: versioni semplificate di quei moduli mancanti, che restituiscono risposte simulate.

Strategia Sandwich (o mista)

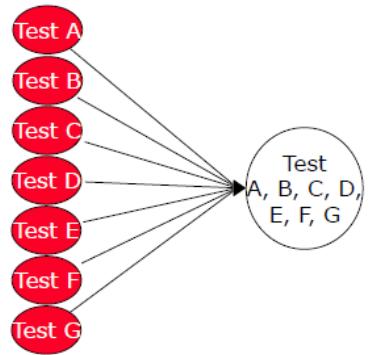
Si lavora contemporaneamente sia con i moduli di alto livello sia con quelli di basso livello. Si integrano prima questi due estremi e poi ci si sposta verso i moduli centrali. Questo approccio si chiama **strategia Sandwich**. In questo caso usi sia **stub** sia **driver**, a seconda di quali moduli ti mancano. È un buon compromesso se vuoi far procedere il test in parallelo su più parti del sistema.



Strategia Big Bang

Non si integra niente finché tutto il sistema non è completo. Quando ogni singolo modulo è pronto, li metti tutti insieme e testi tutto in una volta sola.

Sembra comodo, ma in realtà è rischioso: se qualcosa va storto, non sai subito dove cercare l'errore, e potresti passare molto tempo a capire quale modulo ha causato il problema. In questa strategia **non servono né stub né driver**, perché non fai alcun test parziale.



TEST DI SISTEMA

Il testing di sistema dovrebbe concentrarsi su quattro aspetti:

1. Scoprire se esistono interazioni inattese e indesiderate tra le feature del sistema. Possono verificarsi perché i progettisti delle feature possono avere idee diverse rispetto al loro funzionamento
2. Scoprire se le feature del sistema cooperano in modo efficace a supporto di ciò che gli utenti vogliono realmente fare con il sistema.
3. Accertarsi che il sistema operi nel modo atteso nei diversi ambienti in cui sarà utilizzato.
4. Testare reattività, throughput, sicurezza e altri attributi di qualità del sistema.

Tipi di test di sistema

1. **Test funzionale**, che serve a controllare che il sistema **faccia quello che deve fare**. Si prendono i requisiti funzionali e si verifica che queste funzioni siano effettivamente presenti e funzionanti.
Accanto ai test funzionali troviamo quelli **non funzionali** o di **performance**, che non guardano a cosa fa il sistema, ma come lo fa. Si analizzano quindi aspetti come la **velocità**, l'**usabilità**, la **sicurezza**, la **scalabilità** o la **stabilità**.
2. **Test di accettazione**, orientati all'utente finale e servono a dimostrare che il sistema **soddisfa le aspettative del cliente** o degli utenti per cui è stato costruito. A volte, questi test vengono svolti direttamente da chi ha commissionato il software, oppure da utenti reali in una fase di beta testing.
3. Il **test di regressione**, è svolto ogni volta che si apporta una modifica al sistema per assicurarsi che il resto del software non ne risenta e quindi a verificare che le vecchie funzionalità continuino a comportarsi come prima, senza che siano stati introdotti errori inattesi

4. I **mutation testing** servono a per valutare **l'efficacia dei test** automatici, cioè per capire **quanto siano davvero buoni i test che abbiamo scritto**. Si **modifica intenzionalmente il codice**, in modo piccolo e controllato, per introdurre errori artificiali. Poi si eseguono i test: se i test falliscono (cioè si accorgono che qualcosa è cambiato), allora sono considerati buoni; se invece i test continuano a passare nonostante il codice sia stato "corrotto", vuol dire che **non sono abbastanza sensibili** e potrebbero non accorgersi nemmeno di un vero bug.
5. Gli **scenario test based** è un modalità di testing in cui si presenta uno scenario di funzionamento del sistema in modo da testare feature, sistema ecc. Trasformiamo le user stories in casi di test

TEST DI REALISE

Il testing di rilascio è un tipo di testing del sistema che deve essere rilasciato ai clienti. Le differenze fondamentali tra testing di sistema e di rilascio sono due:

1. Il testing di rilascio verifica il sistema nel suo ambiente operativo reale, anziché in un ambiente di test
2. L'obiettivo del testing di rilascio è decidere se il sistema è nella condizione di essere rilasciato, non di rilevare bug, quindi si possono ignorare alcuni test che falliscono se hanno conseguenze minime per la maggior parte degli utenti.

Legge di conservazione dei bug → N bugs corretti => N bugs nel sistema nuovi

AUTOMAZIONE DEI TEST

Una delle innovazioni più significative nell'ingegneria del software agile è il testing automatico, basato sul concetto che i test debbano essere eseguibili. Un test eseguibile include i dati di input per l'unità da testare, il risultato atteso e una verifica che l'unità restituisca il risultato atteso. Il test viene eseguito, e ha successo se l'unità restituisce il risultato atteso. Di norma, per un prodotto software si dovrebbero sviluppare centinaia o migliaia di test eseguibili.

Graphs for Testing

Per poter strutturare i test si usano i grafi e più in particolare quelli che rappresentano il **flusso di controllo del programma** che mette meglio in evidenza la struttura del codice e come procede l'esecuzione sulla base di iterazioni, selezioni ecc... Lo scopo è quello di **coprire** il grafo e per copertura si intende passare per i vari path che si possono generare sulla base degli input che vengo dati.

I **test path** sono path all'interno del grafo di test che parte da un nodo iniziale e arriva ad un nodo finale che appartengono ad insieme di nodi diversi. Quindi possiamo dire che rappresenta il programma in esecuzione.

I **SESE (Single Entry-Single End) graphs** sono grafi in cui i test path partono da un singolo nodo e finiscono in un singolo nodo.

Visita: Un test path si dice che visita un nodo o un arco se quel nodo o quell'arco fa parte del percorso eseguito dal test.

Tour: Un test path "completa un tour" di un sottopercorso (sottopath) se l'intero sottopercorso è contenuto nel test path.

path(t): Questo indica il percorso eseguito da un singolo test t, ovvero l'insieme di nodi e archi attraversati da quel test durante la sua esecuzione.

path(T): Qui si parla dell'intero insieme di test path eseguiti da un insieme di test T. In pratica, è l'unione dei percorsi di tutti i test appartenenti a T.

Un nodo o arco può essere **raggiunto** da un altro nodo o arco, se esiste una sequenza che permette di raggiungere il nodo o arco interessato. Nei test graph si può raggiungere uno di questi elementi in due modo:

1. **Sintatticamente:** che sulla base della struttura del grafo può essere raggiunto
2. **Semantica:** se esiste un'esecuzione reale del programma che può effettivamente passare per quel nodo o arco, tenendo conto delle condizioni logiche e del valore delle variabili. Anche se un nodo è collegato ad un altro, potrebbe non essere semanticamente raggiungibile se le condizioni per arrivarci non si verificano mai.

Per modellare i software come dei grafi definiamo:

1. **Test Requirements** Sono delle **condizioni o proprietà** che i percorsi di test (test path) devono soddisfare. In pratica, definiscono cosa deve essere coperto o verificato nei test (ad esempio: "tutti i nodi devono essere visitati almeno una volta").
2. **Test Criterion** È l'**insieme di regole** che definisce quali devono essere i test requirements. Ogni criterio impone un certo tipo di copertura (ad esempio: coprire tutti i nodi, tutti gli archi, tutti i possibili usi di una variabile, ecc.).
3. **Satisfaction** Dato un insieme di test requirements **TR** relativo a un criterio C, un insieme di test T soddisfa quel criterio C se per ogni requisito in TR

esiste almeno un percorso di test in T che lo copre. Il criterio è soddisfatto se tutti i requisiti sono stati effettivamente testati almeno una volta.

4. **Structural Coverage Criteria** Sono criteri basati **solo sulla struttura del grafo** che rappresenta il programma o il flusso di controllo. Guardano solo a **nodi e archi** del grafo
5. **Data Flow Coverage Criteria** Questi criteri sono più avanzati e richiedono che il grafo sia **annotato con informazioni sulle variabili** (come dove vengono definite e usate). L'obiettivo è verificare che i test coprano i diversi modi in cui i dati (cioè le variabili) si muovono attraverso il programma.

Una volta che abbiamo deciso cosa rappresentare e come modellarlo possiamo decidere le strategie di copertura dei grafi scelte sulla base degli obiettivi che ci siamo prefissati.

CRITERI DI COPERTURA STRUTTURALE

Node Coverage Ogni **nodo** del grafo (che rappresenta un punto di esecuzione del programma) deve essere visitato da almeno un test. Serve a garantire che **ogni punto del programma venga eseguito almeno una volta**.

Un insieme di test T soddisfa la copertura dei nodi su un grafo G se e solo se, per ogni nodo sintatticamente raggiungibile n appartenente all'insieme dei nodi N , esiste almeno un percorso p nell'insieme dei percorsi $\text{path}(T)$ tale che p visita il nodo n .

Edge Coverage Ogni **arco** (cioè ogni transizione tra due nodi) deve essere percorso da almeno un test.

La **Node Coverage (NC)** è un criterio in cui l'insieme dei **Test Requirements (TR)** contiene **ogni nodo raggiungibile** nel grafo G .

La edge coverage è più rigorosa della copertura dei nodi, perché impone anche la verifica delle transizioni tra punti del programma.

Edge-Pair Coverage Ogni **coppia di archi adiacenti** (cioè sequenze di due transizioni consecutive) deve essere coperta. Assicura che le **sequenze di decisioni** vengano testate.

Complete Path Coverage L'obiettivo è eseguire **tutti i possibili percorsi** nel grafo. Quando ci sono strutture di iterazione, soddisfare questa copertura è

impossibile perché si hanno **infiniti path** visto che ogni volta che percorre un ciclo in un numero diverso di volte può esser considerato un path.

Alcune soluzioni per gestire i loop:

- eseguire il ciclo una sola volta
- eseguire ogni ciclo esattamente una sola volta
- eseguirlo, zero, una o più volte

Oggi la soluzione migliore è quella dei **prime paths**.

CRITERI DI COPERTURA PER FLUSSO DI DATI

L'obiettivo è verificare come le variabili vengono definite, utilizzatee modificate nel programma.

- **Definizione (def)**: un punto nel codice in cui una variabile riceve un valore
- **Uso (use)**: un punto in cui quella variabile viene utilizzata
 - **use computazionale (c-use)**: quando la variabile è usata in un'espressione o calcolo
 - **use predicativa (p-use)**: quando la variabile è usata in una condizione
- **DU pair** Una coppia di posizioni (l_i, l_j) tale che una variabile v è definita in l_i (cioè le viene assegnato un valore) e usata in l_j . una coppia composta da una definizione e da un uso della stessa variabile, collegati da un cammino nel grafo senza una nuova definizione nel mezzo.
- **Def-clear** Un percorso da l_i a l_j è def-clear rispetto alla variabile v se v non viene ridefinita in nessun nodo o arco lungo quel percorso.
- **Reach** Se esiste un percorso def-clear da l_i a l_j rispetto alla variabile v , allora si dice che la definizione di v in l_i raggiunge l'uso in l_j . Il valore assegnato in l_i può effettivamente influenzare quello che succede in l_j .
- **du-path** È un sottopercorso semplice (cioè senza ripetizioni di nodi) che è def-clear rispetto a v , e che va dalla definizione di v a un suo uso. Serve per analizzare i percorsi reali in cui un valore può propagarsi nel programma.
- **du(n_i, n_j, v)** È l'insieme di tutti i du-paths (cioè percorsi def-clear) che vanno dal nodo n_i al nodo n_j per la variabile v
- **du(n_i, v)** È l'**insieme di tutti i du-paths** che **iniziano nel nodo n_i** , dove c'è una definizione della variabile v .

Principali criteri di copertura per flussi di dati

All-Defs Coverage Se per set di du-paths $S=du(n, v)$, TR contiene un path d in S.

👉 tra i requisiti di test (TR), c'è almeno un percorso che parte da quella definizione di **v** nel nodo **n** e arriva fino a un suo uso, senza che v venga ridefinita nel mezzo.

All-Uses Coverage Per ogni set di du-paths $S=du(n_i, n_j, v)$, TR contiene almeno un path d in S.

👉 Ogni volta che assegna un valore a una variabile e poi la usi da qualche parte, devi avere almeno un test che verifica quel collegamento diretto, assicurandoti che quel valore venga davvero usato senza essere cambiato prima.

All-du-Paths Coverage Per ogni du (n_i, n_j, v) , TR contiene ogni path d in S.

👉 Quindi, testare tutti i percorsi semplici (cioè senza cicli interni) dalla definizione all'uso, senza ridefinizioni intermedie.

APPLICAZIONI Code coverage

Un'applicazione comune dei criteri dei grafi è il codice rappresentato tramite **grafo del flusso di controllo (CFG)**.

- Copertura dei nodi: Esegui ogni istruzione
- Copertura dei bordi: Esegui ogni ramo
- Loops : Strutture di loop come per i loop, mentre i loop, ecc
- Copertura del flusso di dati: vanno meglio a definire il CFG
 - – **defs** sono istruzioni che assegnano valori alle variabili
 - – gli **usi** sono istruzioni che utilizzano variabili

Un CFG modella tutte le esecuzioni di un metodo descrivendo le strutture di controllo

- **Nodi** : Istruzioni o sequenze di affermazioni (base blocchi)
- **Edge** : Trasferimenti di controllo
- **Blocco di base** : Una sequenza di affermazioni tale che se il primo viene eseguita, tutte le istruzioni saranno (senza rami)

I CFG sono talvolta annotati con informazioni aggiuntive come predici di ram, defs e usi

APPLICAZIONI Logic Coverage

La **Logic Coverage** si occupa di verificare come le condizioni logiche vengono valutate ed eseguite all'interno del codice. Il suo obiettivo è esplorare le varie possibili valutazioni di una condizione logica, non solo se è vera o falsa, ma anche come lo diventa.

Usiamo i predici nei test come segue:

– Sviluppo di un modello del software come uno o più predici

- Richiedere test per soddisfare una combinazione di clausole

Abbreviazioni:

- P è l'insieme dei predicati
- p è un singolo predicato in P
- C è l'insieme delle proposizioni in P
- Cp è l'insieme delle proposizioni nel predicato p
- c è una singola proposizione in C

Principali tipi di Logic Coverage

Predicate Coverage (PC) Per ogni p in P, TR contiene due requirements=> p valuta sia la condizione vera e falsa, senza interessarci di come ci siamo arrivati

Clause Coverage Per ogni c in C, TR contiene due requisiti: C restituisce vero e c restituisce falso, ma in questo caso siamo interessati alla clausola, quindi di come arriviamo ad avere un risultato vero e uno falso.

PC non esercita pienamente tutte le clausole, soprattutto nel presenza di valutazione del corto circuito

- CC non sempre garantisce il PC
- Cioè, possiamo soddisfare CC senza causare che il predicato sia sia vero che falso
- Questo non è assolutamente quello che vogliamo!
- La soluzione più semplice è quella di Testare tutte le combinazioni...

Il criterio Predicate Coverage richiede che ogni intera espressione logica venga valutata almeno una volta come vera e una volta come falsa.

Il problema è i linguaggi di programmazione moderni usano la **short-circuit evaluation** per evitare valutazioni inutili. Quindi non si garantisce che ogni singola condizione all'interno del predicato venga realmente valutata
if ($x > 0 \text{ || } y > 10$)

Se $x > 0$ è vero, allora l'intero predicato è già vero, e $y > 10$ non viene nemmeno controllato. Anche se il predicato è stato valutato, non tutte le sue clausole sono state esercitate

Il criterio **Clause Coverage** richiede che ogni singola clausola (cioè ogni condizione dentro il predicato) venga valutata almeno una volta come vera e una volta come falsa, senza considerare il risultato finale del predicato. È possibile soddisfare la Clause Coverage **senza mai cambiare il risultato finale del predicato** da vero a falso (o viceversa). Cioè la clausola cambia, ma non ha

effetto sull'esito della decisione e vuole dire che stiamo testando una condizione che non ha alcuna influenza reale sul comportamento del programma.

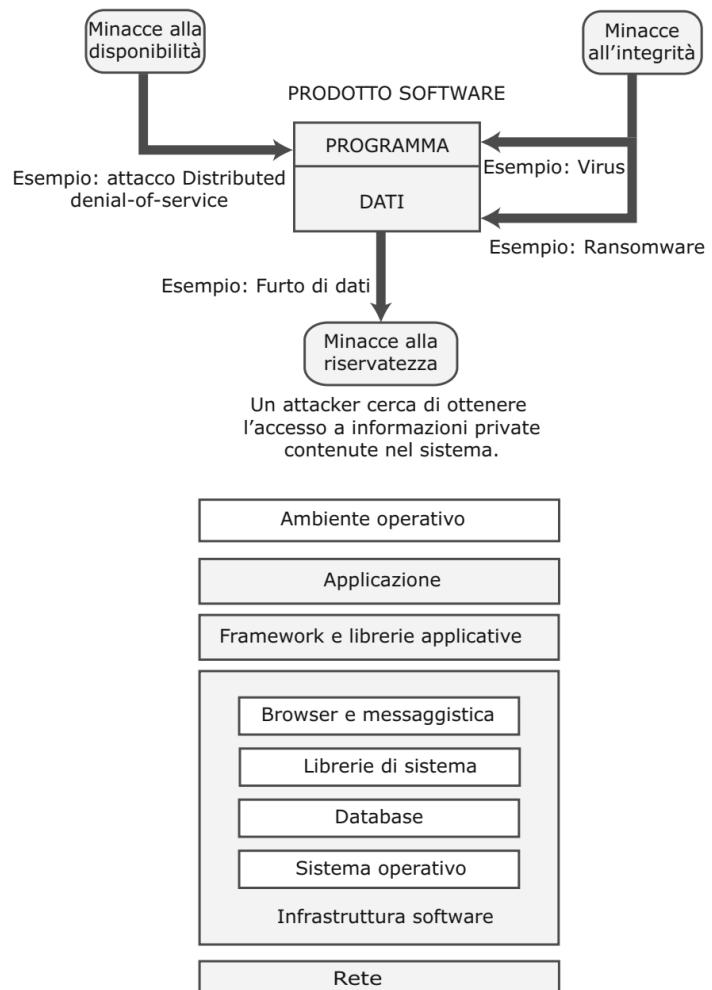
La soluzione è usare la **Combinatorial Clause Coverage**, cioè testare **tutte le possibili combinazioni di vero/falso** delle clausole di un predicato in modo che ogni clausola viene valutata in ogni possibile stato; si verifica se e come ciascuna influisce davvero sull'esito finale.

Il numero di test può **esplodere**: per n clausole, servono 2^n combinazioni.

SICUREZZA DEL CODICE

La sicurezza è un problema che riguarda tutto il sistema. Il software applicativo dipende da una piattaforma di esecuzione che include un sistema operativo, un server web, un sistema di run-time e un database. Esiste anche una dipendenza da framework e strumenti di generazione del codice quando si riusa il software sviluppato da altri.

L'obiettivo di un attacco può essere il furto di dati, oppure appropriarsi di un computer per qualche scopo criminale. Mantenere la sicurezza dell'infrastruttura software è una questione di gestione del sistema più che di sviluppo del software. Sono necessarie procedure e politiche di gestione per ridurre al minimo il rischio che un attacco abbia successo e riesca a compromettere il sistema applicativo



Procedura	Spiegazione
Autenticazione e autorizzazione	È necessario disporre di standard e procedure di autenticazione e autorizzazione che garantiscono a tutti gli utenti un'autenticazione forte e permessi di accesso correttamente configurati. Ciò riduce al minimo il rischio che utenti non autorizzati accedano alle risorse del sistema.
Gestione dell'infrastruttura di sistema	Il software dell'infrastruttura deve essere configurato correttamente, e le patch di sicurezza che correggono le vulnerabilità devono essere applicate appena sono disponibili.
Monitoraggio degli attacchi	Il sistema dovrebbe essere controllato regolarmente per verificare che possibili accessi non autorizzati siano identificati. Se viene rilevato un attacco, può essere possibile mettere in atto strategie di difesa che ne riducano l'effetto.
Backup	Vanno implementate politiche di backup che garantiscono l'esistenza di copie non danneggiate dei file dei programmi e dei dati. Sarà possibile ripristinarli dopo l'attacco.

AUTENTICAZIONE

L'**autenticazione** è il processo che garantisce che un utente del sistema sia effettivamente chi dichiara di essere.

L'autenticazione software si basa su vari approcci, anche combinati.

Autenticazione basata sulla conoscenza prevede che l'utente fornisca informazioni personali e segrete all'atto della registrazione sul sistema. Ogni volta che un utente accede, il sistema chiede alcune di queste informazioni.

Punto debole	Spiegazione
Password non sicure	Gli utenti scelgono password facili da ricordare, ma ciò facilita il compito anche agli attacker che possono indovinarle o generarle con attacchi di tipo dictionary o brute force.
Phishing	Gli utenti cliccano su un link in un'email che punta a un sito fraudolento, che cercherà di carpire i dettagli di login e password.
Riutilizzo della password	Gli utenti usano la stessa password per più siti; se la sicurezza di uno dei siti viene violata, gli attacker hanno delle password che possono provare anche su altri siti.
Password dimenticate	Gli utenti dimenticano regolarmente le password, quindi è necessario fornire un meccanismo di recupero che consenta di reimpostarle. Ciò può essere una vulnerabilità se le credenziali degli utenti sono state rubate, e l'attacker può usare il meccanismo per reimpostare le loro password.

È possibile ridurre i rischi dell'autenticazione con password obbligando gli utenti a usare password forti, ma questo aumenta la probabilità che le dimentichino. L'autenticazione con password può essere rafforzata con ulteriori elementi di conoscenza, chiedendo agli utenti di rispondere a delle domande, oltre che di inserire la password.

Autenticazione basata sul possesso si basa sul fatto che l'utente dispone di un dispositivo fisico che può essere collegato al sistema di autenticazione. Il dispositivo può generare o visualizzare informazioni note al sistema di autenticazione; l'utente le inserisce per confermare di essere in possesso del dispositivo. La versione più usata di questo tipo di autenticazione prevede che l'utente fornisca il numero del telefono mobile quando si registra per creare un account. Il sistema di autenticazione invia un codice al numero di telefono dell'utente, poi l'utente deve inserire il codice per completare l'autenticazione.

Autenticazione basata sulle caratteristiche si riferisce a una caratteristica biometrica unica dell'utente, come un'impronta digitale, registrata sul sistema.

Autenticazione a due fasi, dopo aver inserito la password, l'utente deve inserire un codice inviato al telefono mobile. L'uso del telefono dà un ulteriore livello di sicurezza, poiché il telefono deve a sua volta essere sbloccato con un codice, un'impronta o in qualche altro modo.

HTTP Basic L'autenticazione HTTP Basic è uno dei meccanismi più semplici per proteggere l'accesso alle risorse su un server. Si basa su uno schema che prevede la collaborazione del client, cioè del software che invia le richieste al server.

Quando una risorsa sul server è protetta, il server si aspetta che ogni richiesta contenga nell'header un campo

Authorization: Basic <stringa codificata>

nella quale siano presenti le credenziali dell'utente, cioè **username e password**. Queste credenziali vengono concatenate in una stringa del tipo username:password e poi codificate in Base64.

Sul server, è necessario mantenere una **lista di utenti autorizzati**, cioè un insieme di credenziali valide che possono accedere a determinate risorse. Ogni volta che arriva una richiesta, il server controlla se nell'header sono presenti username e password e verifica se corrispondono a un utente autorizzato.

Se la richiesta non contiene l'header Authorization, oppure se le credenziali non sono corrette, il server risponde con un codice di errore 401 Unauthorized. In risposta, invia anche un'intestazione WWW-Authenticate, che serve a indicare al client che l'accesso richiede un'autenticazione. Nei browser, questa risposta attiva automaticamente la comparsa di un **pop-up di login**, in cui l'utente può inserire le proprie credenziali.

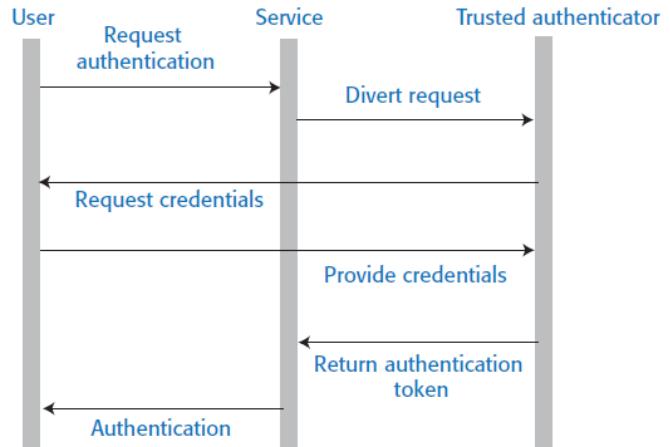
Una volta effettuato l'accesso, il browser memorizza temporaneamente username e password, e le invia automaticamente in tutte le richieste successive allo stesso server. Questo comportamento è coerente con la natura **stateless** del protocollo HTTP, che non mantiene informazioni di stato tra una richiesta e l'altra. Tuttavia, questo ha uno svantaggio importante: **non esiste un vero e proprio meccanismo di logout**.

Identità federata impiega un servizio esterno per l'autenticazione, come Accedi con Google o Accedi con Facebook ecc

Il vantaggio dell'identità federata per gli utenti è che necessitano di un unico gruppo di credenziali memorizzato da un servizio di identificazione fidato.

Anziché accedere direttamente a un servizio, le credenziali vengono passate a un servizio noto che conferma l'identità al servizio di autenticazione. Non è necessario ricordare ID e password diverse, e poiché le credenziali sono memorizzate in meno luoghi si riducono le possibilità di una violazione della sicurezza che possa rivelarle.

“Accedi con Google”->l’utente che la clicca viene dirottato al servizio di identificazione di Google, che convalida l’identità dell’utente usando le credenziali del suo account Google, poi restituisce al sito da cui proviene la richiesta un token che conferma che l’utente è un utente registrato presso Google. Se l’utente ha già effettuato l’accesso a un servizio Google, l’identità è già stata registrata e l’utente non deve inserire altre informazioni.



I vantaggi dell’autenticazione mediante identità federata sono due:

1. Non è necessario mantenere un database di password e altre informazioni segrete. Implementare e mantenere un sistema di autenticazione può risultare molto oneroso per una piccola azienda di sviluppo
2. Il provider di identità può fornire ulteriori informazioni sull’utente, utilizzabili per personalizzare il servizio o per inviare all’utente pubblicità mirata. Naturalmente, quando si imposta un sistema di identità federata con un grande provider è necessario chiedere agli utenti se desiderano condividere con l’azienda fornitrice del prodotto le informazioni in possesso del provider, e non è garantito che accettino.

Per implementarla si usa il protocollo **OAuth**, pregettato per supportare l’autenticazione distribuita e la restituzione di token di autenticazione al sistema chiamante. I **token OAuth** non includono informazioni sull’utente autenticato, indicano soltanto che l’accesso deve essere consentito. Ciò significa che non è possibile usare i token di autenticazione OAuth per prendere decisioni sui privilegi dell’utente, ad esempio a quali risorse del sistema deve poter accedere. Per aggirare il problema è stato sviluppato un protocollo di autenticazione denominato **OpenID Connect**, che fornisce informazioni sull’utente prelevandole dal sistema di identificazione

Autenticazione dei dispositivi mobili Un’alternativa alla coppia login/password usata nei dispositivi mobili prevede l’installazione di un token di autenticazione sul dispositivo. Quando l’app si avvia, il token viene inviato al fornitore del servizio per identificare l’utente del dispositivo.

Gli utenti si registrano dal sito web del fornitore della app, e creano un account nel quale definiscono le loro credenziali di autenticazione. Quando installano l’app, gli utenti si autenticano usando le stesse credenziali, che sono inviate su una connessione sicura al server di autenticazione. Il server emette quindi un

token che viene installato sul dispositivo mobile dell’utente. In seguito, quando l’app viene avviata invia il token al server di autenticazione per confermare l’identità dell’utente. Per maggior sicurezza, il token di autenticazione può scadere dopo un certo periodo di tempo, quindi l’utente dovrà periodicamente ri-autenticarsi presso il sistema.

Un potenziale punto debole di questo approccio è che se un dispositivo viene rubato o perduto, qualcuno che non è il suo proprietario potrebbe riuscire ad accedere al prodotto software. È possibile evitarlo verificando che il proprietario del dispositivo abbia impostato un codice o un’identificazione biometrica che dovrebbe proteggere il dispositivo da accessi non autorizzati. In alternativa, all’utente sarà richiesto di re-identificarsi a ogni avvio dell’app.

AUTORIZZAZIONE

L’autenticazione prevede che un utente provi la sua identità a un sistema software. L’autorizzazione è un processo complementare, nel quale l’identità è utilizzata per controllare l’accesso alle risorse di un sistema software. La definizione dell’accesso alle risorse concesso agli utenti, si basa su una **politica di controllo degli accessi** costituita da un insieme di regole che definiscono **quali informazioni** sono controllate, **chi** può accedervi e il **tipo di accesso** consentito.

Access Control Matrix

Le liste di controllo degli accessi (access control list, ACL) sono impiegate nella maggior parte dei file system e nei database per implementare le politiche di controllo degli accessi. Le ACL sono tabelle che collegano utenti e risorse, e specificano che cosa è permesso agli utenti. Ad esempio, per un libro sarebbe opportuno impostare un’ACL per il file del libro che consenta ai revisori di leggere il file e di annotarlo con commenti, ma non di modificare il testo né di eliminare il file.

Se le ACL sono relative ai permessi dei singoli possono ben presto diventare molto grandi; è però possibile ridurne drasticamente le dimensioni suddividendo gli utenti in gruppi, e poi associando i permessi ai gruppi (Figura 7.8). Usando una gerarchia di gruppi sarà possibile aggiungere permessi o limitazioni a sottogruppi e singole persone.