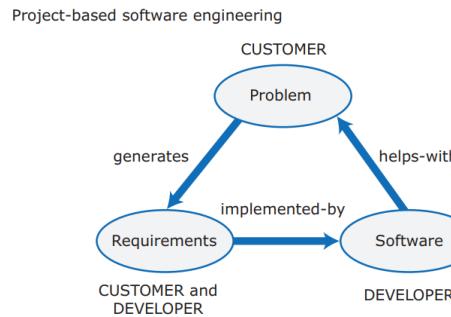


Software products

Question of Tony: What are the differences between project-based and product-based software engineering? What is software product management?

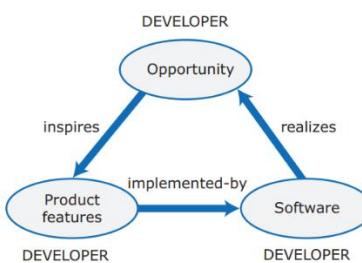
I **project based software “su misura”** sono sw in cui un cliente esterno decide le funzionalità del sistema e stipula un contratto con l'azienda di sviluppo. I requisiti software sono basati sui problemi e processi attuali del cliente. Con il cambiamento delle esigenze aziendali, il software deve essere aggiornato, e il cliente paga per le modifiche. Spesso il costo di aggiornare sistemi complessi supera quello dello sviluppo iniziale.



Man mano che sempre più aziende automatizzavano le loro attività, si capì che la maggior parte non aveva bisogno di software su misura. Potevano utilizzare **prodotti software generici** progettati per problemi aziendali **comuni**.

I **product based software** sono prodotti in cui la progettazione è guidata da un'**opportunità** che l'azienda ha identificato per creare un prodotto commerciale redditizio. L'azienda **decide i requisiti** del sw e ha il potere di decidere quali funzionalità includere nel prodotto software, progettandole in base alle esigenze comuni del mercato o per risolvere problemi specifici che il prodotto intende risolvere. L'azienda determina quando rilasciare nuovi aggiornamenti, in base alle esigenze del mercato.

Sebbene le esigenze dei clienti esterni vengano considerate, questi non possono richiedere specifiche funzionalità.



Il software basato su prodotti è più economico per ciascun cliente, poiché i costi di sviluppo sono distribuiti su un'ampia base di utenti. Tuttavia, gli acquirenti devono adattare i loro processi al software, poiché non è stato progettato per le loro esigenze. Inoltre, le aziende possono decidere quando modificare o ritirare dal mercato un prodotto, senza essere obbligati a supportarlo per tutto il suo ciclo di vita.

Una **linea di prodotti software** è un insieme di prodotti che condivide un nucleo comune, ma ogni membro include adattamenti e aggiunte specifiche per il cliente. (Android e Samsung Android).

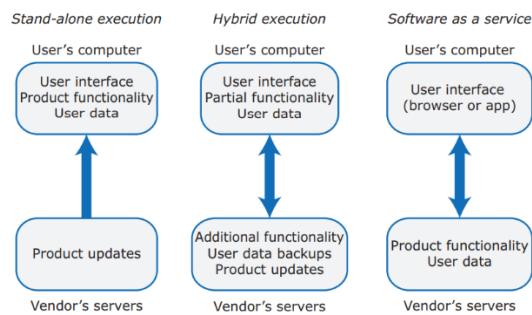
Una **piattaforma** è un prodotto che fornisce un insieme di funzionalità e strumenti utilizzabili per sviluppare nuove funzionalità. Tuttavia, è necessario operare sempre entro le restrizioni stabilite dal fornitore della piattaforma. (Android e SDK)

Software execution models

Prodotto stand alone: Prodotto che funziona in modo autonomo, senza la necessità di essere collegato con altri sistemi per funzionare. Tutti i dati e le funzionalità sono installati sulla macchina dell'utente.

Prodotti ibridi: Prodotto in cui alcune funzionalità sono implementate sul computer dell'utente, mentre altre sono eseguite sui server del fornitore del prodotto, accessibili tramite Internet.

Prodotti basati su servizi: Le applicazioni sono accessibili via Internet. Sebbene ci possa essere un'elaborazione locale con JavaScript, la maggior parte dei calcoli avviene sui server remoti.



The product vision

La **vision** di un prodotto è una **descrizione** semplice e concisa che definisce l'**essenza** del prodotto, spiegando come il prodotto si differenzia dagli altri prodotti concorrenti.

La vision è utilizzata come base per sviluppare una descrizione più dettagliata delle caratteristiche e degli attributi del prodotto.

La vision del prodotto dovrebbe rispondere a tre domande fondamentali:

1. Qual è il prodotto e cosa rende questo prodotto **diverso** dai prodotti **concorrenti**?
2. Chi sono i **clienti target** per il prodotto?
3. **Perché** i clienti dovrebbero acquistare questo prodotto?

La struttura per scrivere la vision del prodotto si basa sul seguente schema:

- **FOR** (*client target*)
- **WHO** (*esigenza dei clienti, a cosa gli serve*)
- **The** (*nome prodotto*) **is a** (*categoria prodotto*)
- **THAT** (*key benefit, ragione per comprarlo*)
- **UNLIKE** (*competitor alternativi*)
- **OUR PRODUCT** (*in cosa si differenzia il prodotto*)

Esempio: FOR a mid-sized company's marketing and sales departments **WHO** need basic CRM functionality, **THE** CRM-Innovator **IS A** Web-based service **THAT** provides sales tracking, lead generation, and sales representative support features that improve customer relationships at critical touch points. **UNLIKE** other services or package software products, **OUR PRODUCT** provides very capable services at a moderate cost.

Sorgenti di informazione per una product vision

Domain experience: I produttori di software possono lavorare nella propria area di dominio e vedere opportunità per un sistema migliorato in base alla propria esperienza.

Product experience: Gli utenti di software esistenti tramite la propria esperienza possono proporre un nuovo sistema che implementi queste soluzioni.

Customer experience: I potenziali clienti acquirenti possono suggerire caratteristiche critiche di cui hanno bisogno per il prodotto.

Prototipazione e sperimentazione: Gli sviluppatori possono sviluppare un sistema prototipo come esperimento e utilizzarlo come sorgente di informazione.

1.2 Software Product Management

Software Product Management è il processo di **pianificazione, sviluppo, rilascio, marketing e manutenzione** di un prodotto software durante tutto il suo ciclo di vita, con l'obiettivo di assicurare che il prodotto soddisfi le esigenze dell'azienda. E' composto dalle attività di **identificazione delle esigenze** dei clienti e dell'azienda, la **definizione della vision** del prodotto, la creazione di una **roadmap** per il completamento del prodotto, lo sviluppo delle **user stories** per identificare le feature, la costruzione del **product backlog** per prioritizzare le feature da sviluppare, l'esecuzione di **accepting e customer testing** e lo sviluppo della **user interface**.

I **product managers** (PMs) sono membri dell'organizzazione che si assumono la **responsabilità complessiva** del prodotto e sono coinvolti in tutto il processo di sw product management. Fungono da **interfaccia** tra il development team, l'organizzazione e i clienti del prodotto. Essi sono coinvolti in tutte le fasi della vita di un prodotto, dalla concezione iniziale, attraverso lo sviluppo della vision e l'implementazione, il marketing, fino a prendere decisioni su quando il prodotto dovrebbe essere ritirato dal mercato.

Il PM deve assicurarsi che il development team implementi funzionalità che offrano **un reale valore** ai clienti, non solo caratteristiche tecnicamente interessanti. I Product manager si devono occupare delle seguenti aree.

Business Needs: I PMs devono garantire che il software in fase di sviluppo **soddisfi gli obiettivi** dell'azienda e dei suoi clienti. Devono comunicare le preoccupazioni e le esigenze dei clienti e del development team ai manager aziendali.

Technology Constraints: I PMs devono informare gli sviluppatori delle **problematiche tecnologiche** che sono importanti per i clienti. Queste possono influenzare il programma, i costi e la funzionalità del prodotto in fase di sviluppo.

Customer Experience: I PMs dovrebbero essere in comunicazione regolare con i clienti per capire **cosa stanno cercando** in un prodotto, i tipi di utenti e i modi in cui il prodotto potrebbe essere utilizzato.

I PMs sono coinvolti in tutte le attività di sw product management. Diamo ora una spiegazione dettagliata di ogni attività.

Una **product roadmap** è un **piano** per lo sviluppo, il rilascio e il marketing del software product. Essa definisce **obiettivi e traguardi** importanti, come il completamento di funzionalità critiche e la conclusione della prima versione per i test degli utenti.

Le **user stories e gli scenarios** sono descrizioni in linguaggio naturale di **azioni** che gli utenti potrebbero voler compiere con il prodotto, utilizzate per affinare la product vision e identificare le funzionalità del prodotto, immedesimandosi nel cliente.

Un **product backlog** è una lista di cose da fare per completare lo sviluppo del prodotto.

L'**acceptance testing** è il processo di verifica che una release software **soddisfi gli obiettivi** stabiliti nel product roadmap e che il prodotto sia **efficiente** e affidabile.

Il **customer testing** implica la **presentazione** di una release di un prodotto **a clienti** esistenti e potenziali e la raccolta di **feedback** sulle funzionalità del prodotto, sulla sua usabilità e sulla sua compatibilità con il loro business.

L'**user interface (UI)** di un prodotto è fondamentale per l'accettazione commerciale di un software. Prodotti tecnicamente eccellenti sono improbabili di avere successo commerciale se gli utenti li trovano difficili da usare o se la loro UI è incompatibile con altri software utilizzati.

Product Prototyping

Il **product prototyping** è il processo di sviluppo di una **versione preliminare** di un prodotto per **testare** le proprie idee e convincere te stesso e i finanziatori dell'azienda che il tuo prodotto ha un **reale potenziale di mercato**.

I prototipi possono anche aiutarti a capire come organizzare e strutturare la versione finale del tuo prodotto.

Un prototipo può anche aiutare a identificare i componenti software fondamentali per il prodotto e testare la tecnologia. Potresti scoprire che la tecnologia che avevi pianificato di utilizzare è inadeguata e che devi rivedere le tue idee su come implementare il software.

Il tuo obiettivo dovrebbe essere avere una versione funzionante del tuo software che possa essere utilizzata per dimostrare le sue funzionalità principali. Dovresti puntare ad avere un sistema dimostrabile attivo in **quattro-sei settimane**. Naturalmente, dovrà risparmiare tempo su alcuni aspetti, quindi potresti decidere di ignorare questioni come l'affidabilità e le prestazioni e lavorare con un'interfaccia utente rudimentale.

Dovresti sempre considerare il tuo prototipo come un sistema "usa e getta", in quanto i compromessi e le scorticatoi inevitabili che fai per accelerare lo sviluppo portano a prototipi che diventano sempre più difficili da modificare ed evolvere per includere nuove funzionalità.

2. Agile Software Engineering

***Tony:** What is the incremental development and delivery advocated by Agile? What are the key Scrum practices?*

Portare il prodotto ai clienti **rapidamente** è fondamentale. Eccellenti prodotti spesso falliscono perché un prodotto inferiore arriva prima sul mercato e i clienti acquistano quello, trovandosi a essere riluttanti a cambiare prodotto dopo aver investito tempo e denaro nella loro scelta iniziale. Ciò significa che le tecniche ingegneristiche orientate allo sviluppo rapido del software, come i metodi agili, sono universalmente utilizzate per lo sviluppo di prodotti.

L'**Agile software engineering** si concentra sulla **consegna rapida delle funzionalità**, minimizzando l'overhead di sviluppo e rispondendo al continuo cambiamento delle specifiche con modifiche rapide.

Negli anni '90 c'era una visione chiamata **plan-driven development** secondo cui il modo migliore per creare buon software era utilizzare processi di sviluppo software controllati e rigorosi. Questi processi includevano una pianificazione dettagliata del progetto, la specifica e l'analisi dettagliata dei requisiti, l'uso di metodi di analisi e progettazione supportati da strumenti software e un controllo di qualità formale.

Il **plan-driven development** comporta un **notevole overhead** nella pianificazione, progettazione e documentazione del sistema, impossibile da sostenere prodotti di piccole o medie dimensioni. Si impiega troppo tempo a scrivere documenti che potrebbero non essere mai letti, invece di scrivere codice. Di conseguenza, è praticamente impossibile

consegnare il software rapidamente e rispondere rapidamente alle richieste di modifiche del SW.

L'insoddisfazione per il plan-driven development ha portato alla creazione dei **metodi agile** negli anni '90.

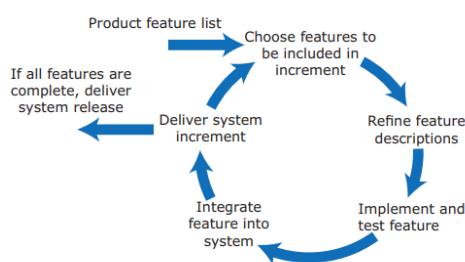
Agile è un approccio allo sviluppo software che privilegia la **consegna rapida** di soluzioni funzionanti, l'adattamento continuo ai requisiti dei clienti e la riduzione della documentazione burocrazia non necessaria, consentendo ai team di concentrarsi principalmente sul software stesso.

La filosofia alla base dei **metodi agile** è riflessa nel **Manifesto Agile**, concordato dai principali sviluppatori di questi metodi.

Messaggio chiave del manifesto agile: Stiamo scoprendo modi migliori per sviluppare software. Abbiamo imparato a dare valore a: Gli individui e le interazioni più dei processi e degli strumenti; Il software funzionante più della documentazione esaustiva; La collaborazione con il cliente più della negoziazione del contratto; Rispondere ai cambiamenti più che seguire un piano.

Tutti i **metodi agile** si basano su sviluppo e consegna incrementale.

Figure 2.1 Incremental development



In agile il prodotto è pensato come un insieme di funzionalità. Ogni funzionalità fa qualcosa per l'utente del software.

Lo sviluppo incrementale è un approccio in cui il software viene sviluppato e consegnato in **piccoli incrementi**, ognuno dei quali implementa un **piccolo numero di funzionalità** del prodotto. Il processo inizia con la **prioritizzazione** delle funzionalità in modo che quelle più importanti vengano implementate per prime. Viene **scelta** una funzionalità e viene **definito** nel **dettaglio** come deve essere implementata. Viene quindi **implementata, testata e integrata** nel sistema. Gli utenti possono provarla e fornire feedback al team di sviluppo. Successivamente si passa a definire e implementare la prossima funzionalità del sistema.

I **metodi agile** funzionano perché i prodotti sono sviluppati da team co-localizzati che possono comunicare informalmente. Il **product manager** può interagire facilmente con il team di sviluppo. Di conseguenza, non c'è bisogno di documenti formali, riunioni o comunicazione tra team diversi.

La nostra massima priorità è soddisfare il cliente attraverso la consegna anticipata e continua di software di valore.
Accogli i cambiamenti nei requisiti, anche nelle fasi avanzate dello sviluppo. I processi agile sfruttano il cambiamento per il vantaggio competitivo del cliente.
Consegnare software funzionante frequentemente, da un paio di settimane a un paio di mesi, con preferenza per intervalli di tempo più brevi.
Le persone di business e gli sviluppatori devono lavorare insieme quotidianamente durante tutto il progetto.
Costruire progetti attorno a individui motivati. Fornire loro l'ambiente e il supporto necessari e fidarsi che svolgano il lavoro.
Il metodo più efficiente ed efficace per trasmettere informazioni a e all'interno di un team di sviluppo è la conversazione faccia a faccia.
Il software funzionante è la principale misura di progresso.
I processi agile promuovono uno sviluppo sostenibile. Gli sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere un ritmo costante indefinitivamente.
L'attenzione continua all'eccellenza tecnica e a un buon design migliora l'agilità.
La semplicità, l'arte di massimizzare la quantità di lavoro non svolto, è essenziale.
Le migliori architetture, requisiti e design emergono da team autorganizzati.
A intervalli regolari, il team riflette su come diventare più efficace, quindi adatta il proprio comportamento di conseguenza.

2.2 Extreme Programming

Extreme Programming (XP) è una metodologia di sviluppo software che porta le pratiche Agile a **livelli estremi**, concentrandosi su sviluppo rapido, incrementale e adattabile ai cambiamenti, con un'enfasi su consegne frequenti e coinvolgimento continuo del cliente.

Pratica	Descrizione
Programmazione in coppia	Due sviluppatori lavorano insieme sullo stesso codice: uno scrive e l'altro rivede, alternandosi regolarmente. Migliora la qualità del codice e favorisce la condivisione della conoscenza.
Proprietà collettiva del codice	Il codice è considerato di proprietà del team, non di un singolo sviluppatore. Qualsiasi membro può modificarlo per migliorarne qualità e funzionalità.
Ritmo di lavoro sostenibile	Promuove un equilibrio tra lavoro e vita privata, evitando straordinari prolungati, per mantenere alta la produttività e prevenire il burnout.
Focus su funzionalità richieste	Si sviluppano solo le funzionalità strettamente necessarie, riducendo il rischio di "over-engineering". Tuttavia, può trascurare aspetti come sicurezza e affidabilità.
Feedback continuo del cliente	Il cliente fornisce feedback su ogni rilascio incrementale, garantendo che il prodotto soddisfi le sue esigenze. Può risultare impegnativo per il cliente mantenere un coinvolgimento costante.

Pratica	Descrizione
Incremental Planning/User Stories	L'intero sistema non viene pianificato in anticipo, ma i requisiti per ogni incremento vengono definiti attraverso discussioni con il cliente. I requisiti sono scritti come user stories e selezionati per i rilasci in base a priorità e tempo disponibile.
Small Releases	Ogni rilascio implementa un set minimo di funzionalità che forniscono valore commerciale. I rilasci sono frequenti e incrementali, aggiungendo valore rispetto al rilascio precedente.
Test-Driven Development (TDD)	Prima di scrivere il codice, si scrivono i test per definire cosa il codice dovrebbe fare, garantendo un sistema sempre testato e funzionante.
Continuous Integration	Ogni nuova feature viene integrata immediatamente nell'intero sistema, generando una nuova versione. I test automatizzati verificano che tutte le modifiche funzionino correttamente prima di accettare la nuova versione.
Refactoring	Si migliora costantemente la struttura, la leggibilità, l'efficienza e la sicurezza del codice, mantenendo il sistema semplice e manutenibile.

XP ha diversi svantaggi. È difficile per i clienti o i product manager trovare il tempo per essere completamente integrati nel team. La proprietà collettiva del codice può risultare impraticabile, specialmente in contesti che richiedono specialisti. Non ci sono prove chiare che la programmazione in coppia sia più produttiva; può essere percepita come inefficiente dai manager. Convincere i manager a mantenere un ritmo di lavoro sostenibile può essere complicato, specialmente con scadenze ravvicinate. La mancanza di un project manager dedicato può portare a inefficienze nella comunicazione e nella pianificazione delle attività.

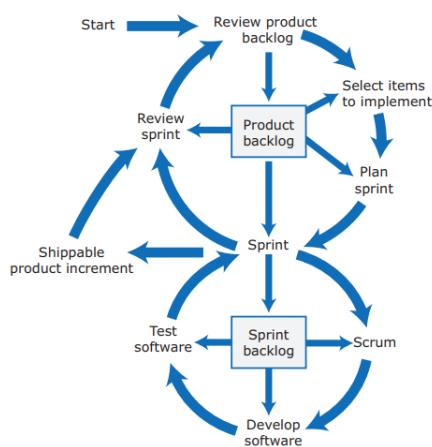
2.3 Scrum

Scrum è un **framework Agile** per l'organizzazione di progetti complessi, basato su iterazioni a tempo fisso (sprint), che enfatizza la collaborazione, la flessibilità e la consegna incrementale di valore attraverso ruoli, eventi e artefatti ben definiti.

Scrum designa due membri, lo **ScrumMaster** e il **Product Owner**, che fungono da interfaccia tra il team di sviluppo e l'organizzazione.

Scrum term	Spiegazione (Italiano)
Product	Il prodotto software che viene sviluppato dal team Scrum.
Product Owner	Un membro del team responsabile dell'identificazione delle caratteristiche e degli attributi del prodotto. Il Product Owner verifica il lavoro svolto e aiuta nei test.
Product backlog	Un elenco di cose da fare, come bug, funzionalità e miglioramenti del prodotto, che il team Scrum non ha ancora completato.
Development team	Un piccolo team auto-organizzato di cinque-otto persone responsabile dello sviluppo del prodotto.
Sprint	Un breve periodo, tipicamente di due-quattro settimane, durante il quale viene sviluppato un incremento del prodotto.
Scrum	Una riunione quotidiana del team in cui si esamina il progresso e si discute e concorda il lavoro da svolgere nella giornata.
ScrumMaster	Un coach del team che guida l'uso efficace di Scrum.
Potentially shippable product increment	Il risultato di uno sprint che ha una qualità sufficiente per essere distribuito ai clienti.
Velocity	Una stima di quanto lavoro un team può completare in un singolo sprint.

Scrum cycle



L'idea fondamentale alla base del processo Scrum è che il software debba essere sviluppato in una serie di "**sprint**".

Uno **sprint** è un'attività di **durata fissa** (timeboxed), in cui il team produce un incremento di prodotto. Ogni sprint dura normalmente da **due a quattro settimane**, in cui si tengono riunioni quotidiane "**Scrum**" per esaminare il lavoro già svolto e concordare le attività da svolgere durante il giorno.

La pianificazione dello sprint si basa sul **product backlog**, che è un elenco di tutte le attività che devono essere completate per terminare il prodotto in sviluppo. Lo **sprint backlog** è un elenco del lavoro da svolgere durante lo sprint.

Prima che inizi un nuovo sprint, viene rivisto il product backlog. Gli elementi con la priorità più alta vengono selezionati dai membri del team per essere aggiunti nello sprint backlog, in modo da essere implementati nel prossimo sprint.

Durante l'implementazione, il team implementa quanti più elementi dello sprint backlog possibile nel periodo di tempo fisso concesso per lo sprint. Gli elementi incompleti vengono restituiti al product backlog. Gli sprint non vengono mai estesi per completare un elemento incompleto.

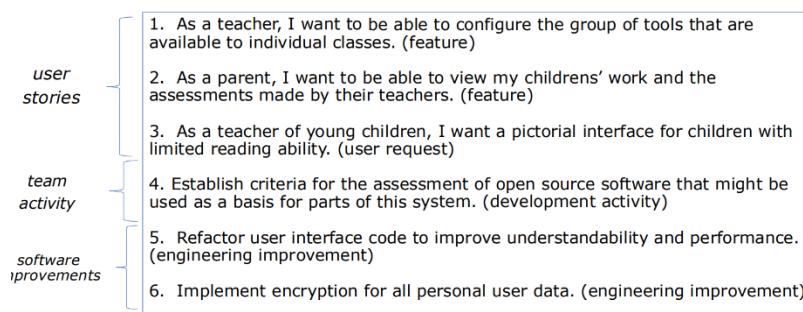
Uno sprint produce o un incremento di prodotto pronto per essere distribuito ai clienti, o un deliverable interno. I deliverable interni, come un prototipo di prodotto o un design architettonico, forniscono informazioni per gli sprint futuri. Se l'output dello sprint è parte del prodotto finale, deve essere codice di qualità, completamente testato, documentato e, se necessario, revisionato, pronto per essere mostrato ai clienti.

A meno che il team non debba cambiare la funzionalità del software, non dovrebbe dover fare ulteriore lavoro su quell'incremento di software negli sprint successivi.

2.3.1 Product backlog

Il **product backlog** è un elenco delle cose che devono essere fatte per completare lo sviluppo del prodotto. Gli elementi in questo elenco sono chiamati **product backlog items** (PBIs).

Il product backlog può includere funzionalità del prodotto da implementare, richieste degli utenti e attività di sviluppo essenziali.



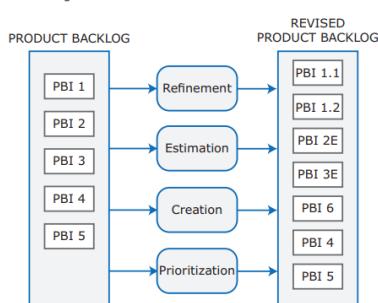
I **PBIs** possono essere specificati ad alto livello di astrazione e il team decide come implementarli.

Gli elementi nel **product backlog** sono considerati in uno di tre stati possibili stati.

Heading	Descrizione
Ready for consideration	Idee e descrizioni generali delle funzionalità che potrebbero essere incluse nel prodotto. Sono provvisorie e soggette a cambiamenti o esclusioni nel prodotto finale.
Ready for refinement	Elementi considerati importanti per lo sviluppo attuale, con una definizione chiara ma non ancora completa. È necessario ulteriore lavoro per comprendere e perfezionare l'elemento.
Ready for implementation	Elementi con dettagli sufficienti per consentire al team di stimare lo sforzo richiesto e implementarli. Le dipendenze con altri elementi sono state identificate.

Il primo elemento del processo di sprint planning è la revisione del **product backlog**. In questa revisione, a cui partecipa tutto il team, il product backlog viene analizzato e i **backlog items** vengono prioritizzati e affinati.

Product backlog activities

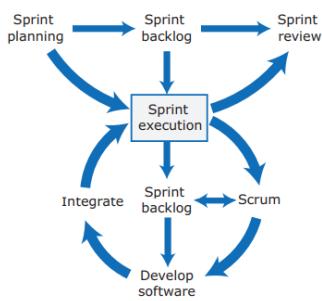


I PBI esistenti vengono analizzati e **raffinati** per creare PBI più dettagliati. Questo può anche portare alla creazione di nuovi elementi di backlog. Il team stima la quantità di lavoro necessaria per implementare un PBI e aggiunge questa valutazione a ciascun PBI analizzato. (Lo sforzo richiesto può essere espresso in person-hours cioè il numero di ore che a una persona serve per implementare quel PBI.). Nuovi elementi vengono aggiunti al backlog. Durante la revisione, gli elementi del backlog possono cambiare stato. Infine, i PBI vengono riordinati per tenere conto di nuove informazioni e circostanze modificate.

2.3.2 Timeboxed Sprint

Durante uno sprint, il team lavora sugli elementi del backlog del prodotto. I prodotti vengono sviluppati in una serie di sprint, ciascuno dei quali fornisce un incremento del prodotto. Gli sprint sono attività di breve durata (1-4 settimane) che si svolgono **tra una data di inizio e una di fine definite**. Gli sprint sono time-boxed, il che significa che lo sviluppo si ferma alla fine di uno sprint, indipendentemente dal fatto che il lavoro sia stato completato o meno.

Sprint activities



Ogni sprint comporta tre attività fondamentali:

Sprint planning: Gli elementi da completare durante lo sprint vengono selezionati e affinati per creare uno **sprint backlog**. Questo non dovrebbe durare più di un giorno all'inizio dello sprint. In questa fase le attività da eseguire sono :

- Stabilire un **sprint goal** concordato: Gli obiettivi dello sprint possono essere incentrati sulla funzionalità del software, sul supporto o sulle prestazioni e l'affidabilità.
- Decidere l'elenco dei PBIs che devono essere implementati. Il criterio è lo sforzo stimato e la velocità del team.
- Creare uno sprint backlog: Creare la versione più dettagliata del product backlog che registra il lavoro da svolgere durante lo sprint.

Sprint Execution: Il team lavora per implementare gli elementi del sprint backlog che sono stati scelti per quello sprint. Se non è possibile completare tutti gli elementi del sprint backlog, il tempo per lo sprint non viene esteso. Piuttosto, gli elementi non completati vengono restituiti al product backlog e messi in attesa per un futuro sprint.

Scrum: Viene effettuata una **breve riunione giornaliera** all'inizio della giornata in cui tutti i membri del team condividono informazioni, descrivono i progressi fatti dal precedente scrum, i problemi che sono emersi e i piani per la giornata successiva.

Le riunioni scrum dovrebbero essere brevi (in piedi) e concentrate. Lo sprint backlog viene esaminato. Gli elementi completati vengono rimossi. Nuovi elementi possono essere aggiunti al backlog man mano che emergono nuove informazioni. Il team decide quindi chi dovrebbe lavorare sugli elementi del backlog di sprint in quella giornata.

Sprint review: Il lavoro svolto durante lo sprint viene esaminato dal team e (possibilmente) da parte degli stakeholder esterni. Il team riflette su ciò che è andato bene e su ciò che è andato storto durante lo sprint, con l'obiettivo di migliorare il processo lavorativo.

2.3.3 Self-organizing teams

I team auto-organizzati non hanno un project manager che assegna compiti e prende decisioni per il team, ma lavorano discutendo le questioni e prendendo decisioni per consenso.

La **dimensione** ideale di un team Scrum è tra **5-8 persone**: abbastanza grande da essere diversificato, ma abbastanza piccolo per comunicare in modo informale ed efficace e per concordare le priorità del team. Poiché i team devono affrontare compiti

diversi, è importante avere una gamma di diverse competenze in un team Scrum, come networking, user experience, design di database e così via.

Una buona comunicazione all'interno del team significa che i membri inevitabilmente imparano qualcosa sulle aree di competenza degli altri. Possono quindi compensare, fino a un certo punto, quando le persone lasciano il team.

Gli sviluppatori di Scrum presumevano che i membri del team fossero co-localizzati. Lavorano nello stesso ufficio e possono comunicare in modo informale. Se un membro del team ha bisogno di sapere qualcosa su ciò che un altro ha fatto, parla semplicemente con l'altro per scoprirla. Non è necessario che le persone documentino il loro lavoro affinché altri lo leggano.

Tramite i daily scrum membri del team spiegano il proprio lavoro e sono a conoscenza dei progressi del team e dei possibili rischi che potrebbero influenzarlo. Tuttavia, ci sono motivi pratici per cui la comunicazione verbale informale potrebbe non funzionare sempre: Scrum presume che il team sia composto da lavoratori a tempo pieno che condividono uno spazio di lavoro. In realtà, i membri del team potrebbero lavorare part-time e in luoghi diversi.

Se il lavoro co-localizzato con riunioni giornaliere è impraticabile, il team deve trovare altri modi per comunicare. I sistemi di messaggistica, possono essere efficaci per le comunicazioni informali, e hanno il vantaggio che tutti i messaggi vengono registrati, in modo che le persone possano aggiornarsi su conversazioni che hanno perso.

Le riunioni giornaliere potrebbero essere a volte impossibili, ma i team agili devono davvero programmare riunioni di aggiornamento regolarmente. I membri che non possono partecipare dovrebbero quindi inviare un breve riassunto dei propri progressi.

Tutti i team di sviluppo hanno alcune interazioni esterne, utili a capire cosa richiedono i clienti dal software o a cosa richiede l'azienda. Solo lo ScrumMaster e il Product Owner dovrebbero essere responsabili congiuntamente della gestione delle interazioni con le persone al di fuori del team.

I Product Owner sono responsabili delle interazioni con i clienti, nonché con il personale di vendita dell'azienda. Il loro compito è comprendere **cosa cercano i clienti** in un prodotto software.

La Scrum Guide afferma che lo ScrumMaster dovrebbe anche lavorare con persone al di fuori del team per "rimuovere ostacoli esterni", in modo che il team possa lavorare allo sviluppo software senza interferenze o distrazioni esterne.

La guida di Scrum non indica ruoli da project manager per lo scrum master. Tuttavia, in tutti i progetti commerciali, qualcuno deve assumersi le responsabilità fondamentali della gestione del progetto.

Un team auto-organizzato deve nominare un membro del team per assumere compiti di gestione. A causa della necessità di mantenere la continuità della comunicazione con le persone al di fuori del gruppo, condividere i compiti di gestione tra i membri del team non è un approccio praticabile.

A mio avviso, è irrealistico escludere le responsabilità di gestione del progetto dal ruolo dello ScrumMaster. Gli ScrumMaster conoscono il lavoro in corso e sono di gran lunga nella migliore posizione per fornire informazioni accurate e piani di progetto e progressi.

4. Features, Personas, Scenarios e User Stories

Tony: *What are personas, scenarios, user stories, and features?*

In genere i prodotti basati unicamente sull'ispirazione degli sviluppatori sono fallimenti commerciali. I **fattori** che **guidano** la progettazione dei prodotti software sono le **necessità** dei consumatori che non sono ancora soddisfatte dai prodotti attuali, l'**insoddisfazione** per i prodotti esistenti e i cambiamenti nella **tecnologia** che rendono possibili tipi completamente nuovi di prodotti.

Nella fase iniziale dello sviluppo del prodotto si cerca di capire quali funzionalità del prodotto saranno utili agli utenti e cosa gli utenti apprezzano e non apprezzano dei prodotti che usano.

Una **feature** è un **frammento di funzionalità** che **implementa un qualche bisogno** dell'utente o del sistema. Il punto di partenza per la progettazione è creare un elenco di feature del prodotto.

Ha senso spendere tempo per cercare di comprendere gli utenti potenziali del prodotto, usando tecniche come le interviste agli utenti, sondaggi, analisi informali e consultazioni.

Per i prodotti aziendali, l'azienda è l'acquirente del prodotto, ma gli utenti sono i dipendenti. Le esigenze dei manager spesso non rispecchiano i desideri degli utenti del prodotto.

Personas

È necessario avere una certa comprensione degli utenti target del prodotto per progettare funzionalità che troveranno utili. I membri del team di sviluppo possono avere idee distorte sugli utenti del prodotto e sulle loro capacità.

Le **personas** sono **rappresentazioni fittizie** ma realistiche di **utenti tipo** che potrebbero adottare il prodotto.

Ad esempio, se il tuo prodotto è destinato alla gestione degli appuntamenti per i dentisti, potresti creare una persona per il dentista, una per il receptionist e una per il paziente.

Le *personas* di diversi tipi di utenti aiutano a immaginare cosa questi utenti potrebbero voler fare con il tuo software, come potrebbero usarlo e le difficoltà che potrebbero incontrare, in modo da valutare se una funzionalità software sarà utile e comprensibile per gli utenti tipici del prodotto.

Una *persona* dovrebbe **descrivere** il background degli utenti e perché potrebbero voler utilizzare il tuo prodotto. Nella descrizione della persona si deve dare loro un **nome**, descrivere le loro **circostanze di vita**, il loro **lavoro**, il loro background **educativo** e il loro livello di **competenze tecniche**.

Jack, insegnante di scuola primaria

Jack, 32 anni, è un insegnante di scuola primaria (elementary school) a Ullapool, un grande villaggio costiero nelle Highlands scozzesi. Insegna a bambini di età compresa tra 9 e 12 anni. È nato in una comunità di pescatori a nord di Ullapool, dove suo padre gestisce un'attività di fornitura di carburanti marini e sua madre è un'infermiera comunitaria. Ha una laurea in inglese presso l'Università di Glasgow e si è formato come insegnante dopo diversi anni di lavoro come autore di contenuti web per un grande gruppo di svago.

L'esperienza di Jack come sviluppatore web lo rende sicuro in tutti gli aspetti della tecnologia digitale. Credere fermamente che l'uso efficace delle tecnologie digitali, combinato con l'insegnamento faccia a faccia, possa migliorare l'esperienza di apprendimento per i bambini. È particolarmente interessato a utilizzare il sistema *iLearn* per l'insegnamento basato su progetti, dove gli studenti lavorano insieme su argomenti interdisciplinari e impegnativi.

Se il tuo prodotto è destinato a un gruppo specifico di utenti, potresti avere bisogno di **1/2 - 4/5 personas** per rappresentare i potenziali utenti del sistema.

Se si lavora su un prodotto nuovo, può essere più difficile accedere ai potenziali utenti, e in questi casi bisogna fare ricerche preliminari informali.

Le **proto-personas** sono versioni preliminari delle *personas*, sviluppate sulla base di informazioni limitate su **ipotesi e conoscenze di base** disponibili al team di sviluppo. Anche se meno precise, le *proto-personas* aiutano comunque il team a orientarsi nelle fasi iniziali del design del prodotto.

Scenarios

Per aiutarti a selezionare e progettare le funzionalità, si consiglia di inventare scenari per immaginare come gli utenti potrebbero interagire con il prodotto che stai progettando.

Uno **scenario** è una narrazione che descrive una situazione in cui un utente **utilizza le funzionalità** del tuo prodotto per fare qualcosa che desidera fare.

Lo scenario dovrebbe spiegare brevemente il problema dell'utente e presentare un modo immaginario in cui il problema potrebbe essere risolto.

Pesca a Ullapool

Jack è un insegnante di scuola primaria a Ullapool e insegna agli alunni di P6. Ha deciso che un progetto di classe sarà incentrato sull'industria della pesca nella zona, esaminando la storia, lo sviluppo e l'impatto economico della pesca.

Come parte del progetto, agli studenti viene chiesto di raccogliere e condividere i ricordi dei parenti, utilizzare gli archivi dei giornali e raccogliere vecchie fotografie relative alla pesca e alle comunità di pescatori della zona. Gli alunni utilizzano un *wiki* di iLearn per raccogliere storie sulla pesca e SCAN (un sito di archivi storici) per accedere agli archivi dei giornali e alle fotografie. Tuttavia, Jack ha anche bisogno di un sito per la condivisione di foto, poiché vuole che gli studenti scattino e commentino le foto degli altri e carichino scansioni di vecchie fotografie che potrebbero avere nelle loro famiglie. Deve essere in grado di moderare i post con le foto prima che vengano condivisi, poiché i bambini preadolescenti non comprendono le questioni di copyright e privacy.

Jack invia un'email a un gruppo di insegnanti di scuola primaria per vedere se qualcuno può raccomandare un sistema appropriato. Due insegnanti rispondono e suggeriscono entrambi di utilizzare KidsTakePics, un sito di condivisione di foto che consente agli insegnanti di verificare e moderare i contenuti. Poiché KidsTakePics non è integrato con il servizio di autenticazione di iLearn, Jack crea un account per insegnante e un account per la classe su KidsTakePics.

Utilizza il servizio di configurazione di iLearn per aggiungere KidsTakePics ai servizi visibili dagli studenti nella sua classe, in modo che, quando accedono, possano utilizzare immediatamente il sistema per caricare foto dai loro telefoni e da' computer della classe.

Gli elementi più importanti che uno scenario deve includere sono una breve dichiarazione **dell'obiettivo generale, riferimenti alla persona coinvolta** ottenere informazioni sulle capacità e motivazioni dell'utente, **informazioni su cosa è coinvolto nell'attività** (includendo attori, strumenti, processi, ambiente, interazioni e obiettivi), **una spiegazione dei problemi** che non possono essere affrontati facilmente con il sistema esistente e una descrizione di come **il problema** identificato potrebbe essere **afrontato**.

Gli *scenarios* sono storie di alto livello sull'uso del sistema, che descrivono una sequenza di interazioni con il sistema, ma non devono includere i dettagli di queste interazioni. Mancano di dettagli, possono essere incompleti e potrebbero non rappresentare tutti i tipi di interazione dell'utente. Sono principalmente un mezzo per facilitare la comunicazione e stimolare la creatività progettuale, piuttosto che fornire una descrizione completa del sistema.

Il punto di partenza per la scrittura degli scenari dovrebbe essere rappresentato dalle *personas* che hai creato. Dovresti cercare di immaginare **3-4 scenari** per **ciascuna persona**.

Ogni membro del team deve creare un piccolo numero di scenari. Il team poi discute insieme gli scenari proposti, in modo da raffinarli.

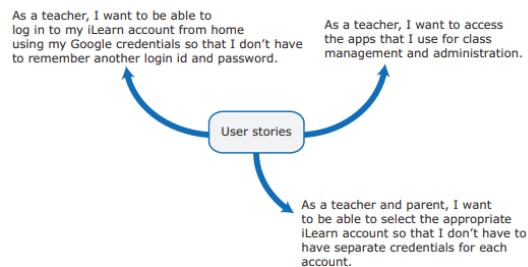
User stories

Le **user stories** sono narrazioni dettagliate e strutturate che descrivono in modo specifico una singola funzionalità che un utente vuole da un prodotto software.

Il formato standard di una user story è: **AS A <ruolo> I WANT / NEED TO <fare qualcosa> SO THAT <ragione>**

Esempio. AS A teacher, I NEED TO be able to report who is attending a class trip SO THAT the school maintains the required health and safety records.

Figure 3.6 User stories from Emma's scenario

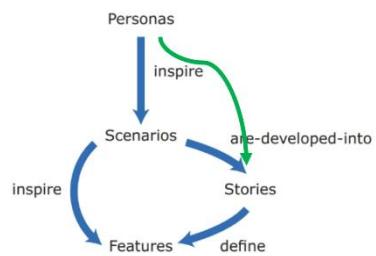


Molti utilizzatori del metodo *Scrum* rappresentano il *product backlog* come un insieme di *user stories*. Se la *story* riguarda una funzionalità complessa che potrebbe richiedere diversi *sprints* per essere implementata, viene chiamata **epic**. Per essere implementata una epic deve essere suddivisa in *stories* più semplici.

Una buona *story* dovrebbe richiedere 1-2 giorni di lavoro. Se una storia è troppo lunga, è consigliabile suddividerla in storie più brevi.

Le storie vengono associate a **priorità** e stimate in base allo sforzo necessario per implementarle. Le storie vengono poi ordinate in base alla loro priorità, utilizzando i colori ROSSO, GIALLO E VERDE, come all'ospedale (triage).

Sebbene sia possibile descrivere una funzionalità tramite user stories, scrivere scenari non è irrilevante perché sono più naturali da leggere e riflettono ciò che un utente effettivamente fa con il sistema. Gli utenti reali non esprimono le loro esigenze nel formato strutturato delle user stories, e gli scenari offrono un contesto più ampio. Tuttavia, alcuni metodi agili e **Brogi consigliano di saltare** la scrittura degli scenari, preferendo l'uso diretto delle user stories.



Identificazione delle features

L'obiettivo in questa fase della progettazione del prodotto è creare un elenco di feature che definiscano il tuo prodotto software.

Una **feature** di un prodotto è una caratteristica che consente agli utenti di svolgere **determinate azioni** o ottenere determinati benefici dall'uso del prodotto. Le feature definiscono ciò che il prodotto **può fare e come** può essere utilizzato. La **lista delle feature** del prodotto definisce la funzionalità complessiva del sistema.

Una feature non dovrebbe **dipendere** da come sono implementate altre feature del sistema e non dovrebbe essere influenzata dall'ordine di attivazione di altre feature. Le feature non dovrebbero fare **più di una cosa** e non dovrebbero mai avere effetti collaterali. Le feature del sistema dovrebbero **riflettere** il modo in cui gli utenti **eseguono normalmente** un compito. Non dovrebbero offrire funzionalità non richieste.

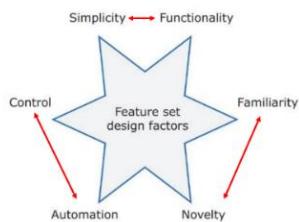
Le quattro principali fonti di conoscenza nella progettazione delle feature sono:

User knowledge: Puoi utilizzare le *user stories* per informare il team su cosa vogliono gli utenti e come potrebbero utilizzare le funzionalità del software.

Product knowledge: Potresti avere esperienza di prodotti esistenti e replicare feature esistenti, poiché forniscono funzionalità fondamentali sempre richieste.

Domain knowledge: Comprendere il dominio del prodotto consente di pensare a modi innovativi per aiutare gli utenti a fare ciò che desiderano.

Technology knowledge: Se comprendi le ultime tecnologie, puoi progettare feature che ne facciano uso.



Ottimizzare un fattore in una feature de ottimizza il suo duale, quindi devono essere fatti dei compromessi. Le scelte da fare sono:

Semplicità e funzionalità: Sviluppare un software il più semplice possibile da usare o offrire funzionalità attrattive.

Familiarità e novità: Offrire feature familiari per gli utenti o aggiungerne novità dando una motivazione per cambiare.

Automazione e controllo: Sviluppare un prodotto che faccia automaticamente cose che altri prodotti non fanno o lasciare il controllo delle azioni all'utente.

Gli sviluppatori dovrebbero cercare di evitare il “*feature creep*.” Il ***feature creep*** si verifica quando **il numero di feature in un prodotto aumenta troppo** man mano che si immaginano nuovi potenziali usi del prodotto.

Il *feature creep* aggiunge complessità a un prodotto, il che significa che è più probabile introdurre bug.

Il *feature creep* si verifica per **tre ragioni**: i *product manager* discutono la funzionalità necessaria con una gamma di utenti diversi, che possono fare **la stessa cosa in modi leggermente diversi**. Prodotti concorrenti vengono introdotti **con funzionalità leggermente diverse** rispetto al tuo prodotto, portando una pressione di marketing per includere funzionalità simili. Il prodotto cerca di supportare sia utenti esperti che inesperti, aggiungendo modi **semplici per eseguire azioni già esistenti**.

Per evitare il *feature creep*, i *product manager* e il team di sviluppo dovrebbero rivedere tutte le proposte di feature e confrontare le nuove proposte con le feature già accettate per l'implementazione.

Software Product

Tony: *What is the role of non-functional quality attributes in a software architecture? What is a distribution architecture? What are the technology choices that affect a software architecture?*

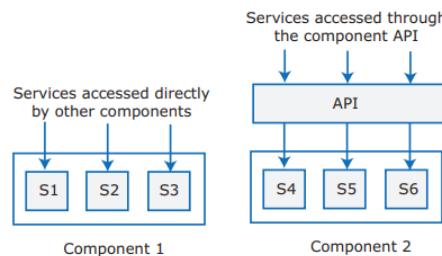
Def. L'**architettura software** è la **struttura** di un sistema composta dai suoi componenti principali, dalle loro interazioni e dai principi di progettazione ed

evoluzione. Essa organizza le parti del software per soddisfare requisiti funzionali e non funzionali, garantendo qualità quali manutenibilità, scalabilità, sicurezza e prestazioni.

L'architettura funge da blueprint per lo sviluppo, l'implementazione e la manutenzione, assicurando un funzionamento armonioso nell'ambiente previsto.

Un **component** è un'unità software con un nome che offre esternamente uno o più **services**. Quando utilizzato da altri componenti, questi servizi vengono accessi tramite un'**API**.

Figure 4.1 Access to services provided by software components



Un **service** è un'unità coerente di **funzionalità**. Può variare da un servizio di grande scala, come un servizio di database, a un microservizio che svolge un'attività molto specifica.

Un **modulo** è un insieme di componenti con un nome. I componenti in un modulo dovrebbero avere qualcosa in comune.

Un team di sviluppo dovrebbe progettare e discutere l'architettura del prodotto software prima di iniziare l'implementazione finale del prodotto. Dovrebbero creare una descrizione dell'architettura del prodotto che ne stabilisca la struttura fondamentale e serva come riferimento per l'implementazione.

Nonostante agile dice che la pianificazione deve essere ridotta al minimo (si sono ricreduti), la struttura complessiva del sistema può essere il risultato di uno scrum.

4.1 Perché l'architettura è importante?

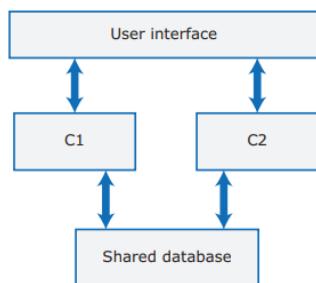
Le caratteristiche "**non funzionali**" sono molto importanti per il prodotto finale. Sono queste caratteristiche, piuttosto che le **feature** del prodotto, a influenzare i **giudizi degli utenti** sulla qualità del software.

Attribute	Key Issue
Security	La sicurezza è cruciale per proteggere i dati e prevenire attacchi.
Usability	L'usabilità determina quanto sia facile per gli utenti interagire con il sistema.
Performance	Le prestazioni indicano la velocità con cui il sistema risponde alle richieste.
Maintainability	La manutenibilità riflette quanto sia semplice apportare modifiche o aggiornamenti al sistema.
Scalability	La scalabilità riguarda la capacità del sistema di gestire un numero crescente di utenti o dati senza compromettere le prestazioni.
Reliability	L'affidabilità misura quanto spesso il sistema funziona correttamente senza interruzioni.
Flexibility	La flessibilità definisce quanto facilmente il sistema può adattarsi a nuovi requisiti o cambiamenti.
Cost	Il costo è una considerazione importante, influenzata dal tempo e dalle risorse necessarie per sviluppare e mantenere il sistema.
Time to market	Il tempo di commercializzazione si riferisce alla velocità con cui il prodotto può essere sviluppato e lanciato sul mercato.
Compatibility	La compatibilità riguarda la capacità del sistema di funzionare con altri software o piattaforme esistenti.

L'architettura ha un'influenza fondamentale su queste proprietà non funzionali. È impossibile ottimizzare tutte le caratteristiche non funzionali nello stesso sistema.

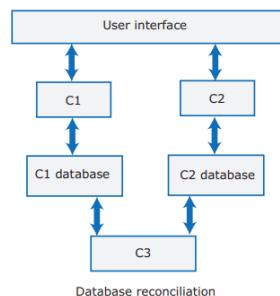
Ottimizzare una caratteristica, come la sicurezza, influisce inevitabilmente su altre caratteristiche, come l'usabilità e l'efficienza del sistema.

Shared database architecture



Se due componenti condividono un DB comune, e uno dei due componenti funziona lentamente a causa del DB, e deve cambiare lo schema del DB, il secondo componente deve essere modificato, il che potrebbe influenzare il suo tempo di risposta.

Figure 4.3 Multiple database architecture



Nel caso in cui ogni componente ha una propria copia delle parti del database di cui ha bisogno, se un componente deve modificare lo schema del DB, ciò non influenza sull'altro componente.

L'architettura a database distribuito potrebbe funzionare più lentamente e potrebbe costare di più da implementare e modificare. Deve esserci un meccanismo (C3) per garantire che i dati condivisi dai due componenti rimangano coerenti quando vengono apportate modifiche.

L'architettura software influisce sulla complessità del tuo prodotto. Più complesso è un sistema, più è difficile capirlo e modificarlo, portando durante sviluppo all'introduzione di bug.

Architectural design

Le caratteristiche non funzionali del prodotto, come la sicurezza e le prestazioni, influenzano tutti gli utenti. Se sbagli queste caratteristiche, è improbabile che il prodotto abbia successo commerciale.

Argomento	Definizione ed Esplicazione
Product lifetime	Periodo durante il quale il software rimane utile per i suoi utenti. Se si prevede un lungo product lifetime, è fondamentale pianificare revisioni regolari. Questo richiede un'architettura capace di adattarsi per integrare nuove funzionalità e tecnologie.
Software reuse	Riutilizzo di componenti già esistenti, per ridurre tempi e costi di sviluppo. Tuttavia, questa scelta impone vincoli architettonici, poiché è necessario adattare il design al software riutilizzato.
Numero di utenti	Quantità di persone che utilizzano il sistema. L'architettura deve essere progettata per scalare efficacemente con l'aumento del carico.
Compatibilità del software	Capacità di un prodotto di funzionare con altri sistemi, assicurando l'integrazione e il riutilizzo dei dati tra piattaforme diverse. Per alcuni prodotti, è fondamentale consentire agli utenti di utilizzare dati generati con sistemi differenti. Questa esigenza può limitare le scelte disponibili nelle decisioni architettoniche.
Manutenibilità	Quanto sia costoso apportare modifiche al sistema una volta rilasciato ai clienti. La manutenibilità viene migliorata costruendo un sistema composto da piccole parti auto-contenute, ognuna delle quali può essere sostituita o migliorata se necessario. Dove possibile, dovresti evitare strutture dati condivise.

System decomposition

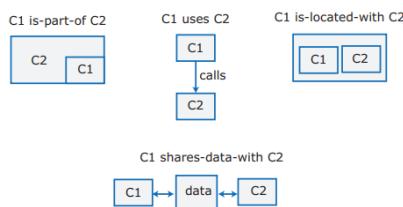
Per **astrazione** si intende concentrarsi sugli **elementi essenziali** di un **sw component** senza preoccuparsi dei dettagli tecnici **irrilevanti** al momento della progettazione.

La **decomposizione** consiste nello scomporre un componente di grandi dimensioni in parti più piccole e gestibili, ciascuna con una propria funzionalità specifica.

La **complessità** in un'architettura di sistema deriva dal numero e dalla natura delle **relazioni** tra i componenti del sistema.

I componenti hanno diversi tipi di relazioni con altri componenti. A causa di queste relazioni, quando modifichi un componente, spesso è necessario apportare modifiche a diversi altri componenti.

Figure 4.7 Examples of component relationships



Relazione Part-of: Un componente fa parte di un altro componente.

Relazione Uses: Un componente utilizza la funzionalità fornita da un altro componente.

Relazione Is-located-with: Un componente è definito nello stesso modulo o oggetto di un altro componente.

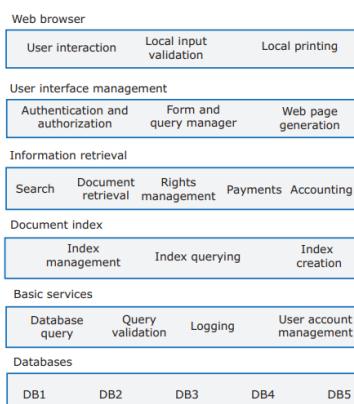
Relazione Shares-data-with: Un componente condivide dati con un altro componente.

Man mano che il numero di componenti aumenta, il numero di relazioni tende ad aumentare a un ritmo più rapido. È impossibile evitare che la complessità aumenti con le dimensioni del software.

Le seguenti linee guida aiutano a controllare la complessità. La linea guida della **separazione delle preoccupazioni** suggerisce di raggruppare le funzionalità correlate. Ogni componente dovrebbe fare solo una cosa. La linea guida "**implementa una sola volta**" suggerisce che non dovesti duplicare funzionalità. La linea guida delle **interfacce stabili** suggerisce che i componenti che utilizzano un'interfaccia non debbano essere modificati ogni volta che l'implementazione sottostante cambia.

I componenti non devono mai essere a conoscenza dell'implementazione di altri componenti e affidarsi ad essa. I dettagli dell'implementazione dovrebbero essere nascosti dietro un'API.

An architectural model of a document retrieval system



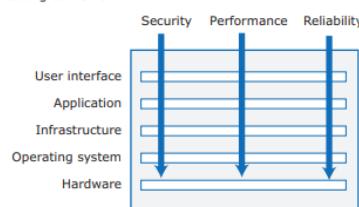
Le architetture a strati sono architetture sw che organizzano **ogni strato** come **un'area di interesse** considerato separatamente dagli altri strati. Gli strati non sono componenti o moduli, ma semplicemente raggruppamenti logici di componenti. Gli strati sono rilevanti in fase di progettazione, ma non si può identificare questi strati nell'implementazione.

All'interno di ogni strato, i componenti sono indipendenti e non si sovrappongono nelle loro funzionalità. I livelli inferiori includono componenti che forniscono funzionalità generali, quindi non è necessario replicarle nei componenti di livello superiore.

Idealmente, i componenti del livello X dovrebbero interagire solo con le API dei componenti del livello X-1. In un modello a strati, i componenti nei livelli inferiori non dovrebbero mai dipendere dai componenti di livello superiore. Se modifichi un componente al livello X nella pila, non doverresti dover apportare modifiche ai componenti dei livelli inferiori nella pila.

Le preoccupazioni trasversali importanti per i prodotti software sono **sicurezza, prestazioni e affidabilità**. Ogni strato deve tenerle in considerazione, e inevitabilmente ci sono interazioni tra i livelli a causa di queste preoccupazioni. Queste preoccupazioni trasversali rendono difficile migliorare la sicurezza del sistema dopo che è stato progettato.

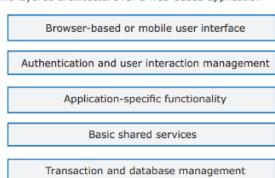
Figure 4.9 Cross-cutting concerns



La sicurezza non può essere concentrata in un singolo livello, poiché ogni livello utilizza tecnologie diverse che potrebbero presentare vulnerabilità sfruttabili dagli attaccanti. È essenziale proteggere ogni livello sia dagli attacchi diretti sia da quelli che potrebbero compromettere i livelli inferiori. Affidarsi a un unico componente di sicurezza rappresenta una vulnerabilità critica: se compromesso o malfunzionante, l'intero sistema perde affidabilità. Distribuendo la sicurezza su più livelli, il sistema diventa più resiliente agli attacchi e ai guasti, riducendo il rischio di fallimenti dovuti a punti deboli isolati.

Molti prodotti software distribuiti via web hanno un template di struttura a strati comune che puoi usare come punto di partenza per il tuo design.

A generic layered architecture for a web-based application



Layer	Descrizione
Interfaccia utente basata su browser o mobile	Gestisce l'interfaccia utente, utilizzando un browser web con moduli HTML per raccogliere input e componenti Javascript per elaborazioni locali, come la validazione. Può includere anche un'interfaccia mobile come app.
Gestione dell'autenticazione e dell'interfaccia utente	Si occupa dell'autenticazione degli utenti e della generazione delle pagine web necessarie per l'interazione con il sistema.
Funzionalità specifiche dell'applicazione	Fornisce le funzionalità principali dell'applicazione, eventualmente suddivise in sottolivelli per una maggiore modularità.
Servizi condivisi di base	Contiene componenti che offrono servizi generici utilizzati dai livelli applicativi, come logging, notifiche o gestione della configurazione.
Gestione del database e delle transazioni	Gestisce le operazioni sul database, incluse transazioni e funzioni di recupero. Questo livello può essere omesso se l'applicazione non utilizza un database.

Il primo passo è pensare se questo template sia quello giusto o se hai bisogno di più o meno strati, adattandolo alle esigenze del sistema che devi progettare. Una volta determinato il numero di livelli da includere nel tuo sistema, puoi iniziare a popolarli. Insieme a tutto il team si sperimentano varie decomposizioni per comprendere i loro vantaggi e svantaggi. La decomposizione del sistema deve essere eseguita insieme alla scelta delle tecnologie per il sistema, perché la scelta della tecnologia utilizzata in un particolare livello influenza i componenti nei livelli superiori.

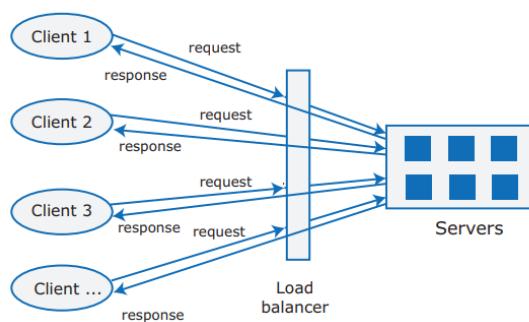
Architettura di distribuzione

L'**architettura client-server** è un modello di distribuzione in cui la funzionalità del sistema è suddivisa tra due entità principali: **il client e il server**. Il **client**, generalmente sul dispositivo dell'utente, gestisce l'interfaccia utente e invia richieste al server. Il **server**, invece, elabora queste richieste eseguendo operazioni di logica di business, accedendo a database condivisi o fornendo altre risorse.

Questa architettura è adatta per applicazioni in cui i client accedono a un database condiviso e a operazioni di logica di business su tali dati.

Durante il processo di progettazione architettonica, devi decidere l'**architettura di distribuzione** del sistema, che definisce i server nel sistema e l'allocazione dei componenti a questi server.

Client-server architecture



Le applicazioni client-server includono diversi server, come server web e server di database. L'accesso all'insieme dei server è solitamente mediato da un **bilanciatore di carico**, che distribuisce le richieste ai server per garantire che il carico di calcolo sia equamente condiviso.

Il client è responsabile dell'interazione con l'utente, basata sui dati inviati e ricevuti dal server. Oggi le macchine client sono computer o dispositivi mobili con molta potenza di elaborazione, quindi la maggior parte delle applicazioni include una significativa elaborazione lato client.

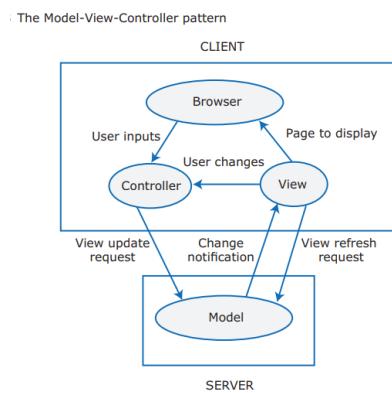
L'interazione client-server è solitamente organizzata utilizzando il **MVC**. Il **Model-View-Controller (MVC)** è un pattern architettonico utilizzato per organizzare l'interazione client-server in modo da separare i dati, la logica di business e la loro presentazione.

Model: Rappresenta i dati e la logica di business del sistema. È mantenuto sul server e agisce come unica fonte di verità. Quando i dati nel model cambiano, tutte le viste registrate vengono notificate per aggiornarsi.

View: Si occupa della presentazione e dell'interfaccia utente. Ogni client ha una propria vista che genera le pagine HTML e gestisce l'interazione con l'utente tramite moduli e interfacce grafiche.

Controller: È responsabile di gestire gli input degli utenti ricevuti dalla vista. Interpreta queste azioni, le traduce in richieste per il model e aggiorna la vista di conseguenza.

Il modello MVC garantisce che il **model (i dati e la logica)** sia disaccoppiato dalla **view (la presentazione)**, consentendo al sistema di aggiornare i dati indipendentemente dalla modalità in cui vengono presentati.



La comunicazione client-server avviene normalmente utilizzando il protocollo **HTTP**. Le richieste strutturate inviate dai client possono essere rappresentate utilizzando **JSON** o **XML**.

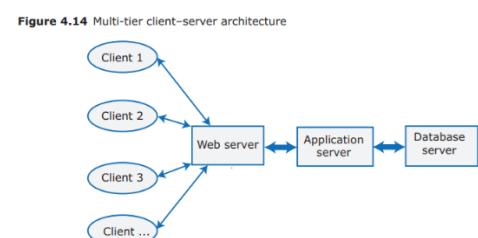
L'**architettura multi-tier** è un modello in cui un'applicazione è suddivisa in livelli distinti, dove ogni server ha responsabilità specifiche. I principali livelli in un'architettura multi-tier sono:

Web server: Comunica con i client tramite il protocollo HTTP, consegna le pagine web e gestisce le richieste HTTP.

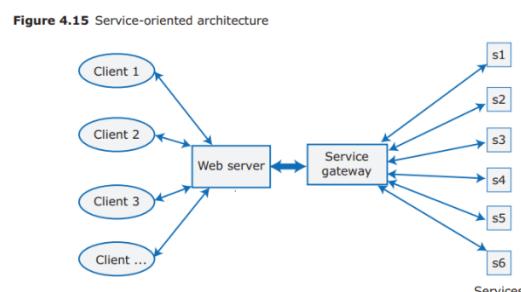
Application server: Esegue la logica di business dell'applicazione, elaborando le richieste ricevute dal web server.

Database server: Gestisce l'archiviazione e il recupero dei dati, rispondendo alle richieste dell'application server.

Questa struttura consente di distribuire i carichi di lavoro tra diversi server, migliorando l'efficienza e permettendo la scalabilità del sistema in base alle necessità.



L'**architettura orientata ai servizi** è un modello in cui le funzionalità di un sistema sono fornite attraverso una collezione di servizi indipendenti e riutilizzabili. I servizi, essendo senza stato, possono essere replicati e migrati tra macchine diverse, rendendo il sistema scalabile e resiliente ai guasti.



Le questioni principali da considerare nella scelta di un'architettura di distribuzione sono le seguenti. Se utilizzi principalmente dati strutturati, è preferibile avere un database condiviso per gestire blocchi e transazioni. Se prevedi che i componenti del

sistema vengano regolarmente modificati, isolare questi componenti come servizi separati semplifica i cambiamenti. Se prevedi di eseguire il sistema nel cloud con accesso via Internet, un'architettura orientata ai servizi è generalmente la scelta migliore per la scalabilità. Tuttavia, per sistemi aziendali che funzionano su server locali, un'architettura multi-tier potrebbe essere più adatta.

Questioni tecnologiche

Le tecnologie scelte per un sistema influenzano profondamente la sua architettura e limitano le opzioni disponibili durante lo sviluppo. Poiché cambiarle in corso d'opera è complesso e costoso, è fondamentale valutarle con attenzione sin dall'inizio. Queste scelte riguardano componenti chiave come database, piattaforme di distribuzione, server (interni o cloud), strumenti di sviluppo e l'eventuale adozione di tool open source.

Database

Oggi esistono due principali categorie di database:

Database relazionali (RDBMS): Organizzano i dati in tabelle strutturate e utilizzano linguaggi come SQL per interrogarli. Sono ideali per applicazioni in cui è essenziale la gestione delle transazioni e le strutture dati sono ben definite e prevedibili. Supportano transazioni **ACID** (Atomicità, Coerenza, Isolamento, Durabilità), garantendo che il database rimanga coerente anche in caso di errori o guasti. Esempio: **MySQL**. Sono preferibili per applicazioni con dati strutturati, necessità di coerenza rigorosa e requisiti transazionali.

Database NoSQL: Forniscono un'organizzazione dei dati più flessibile, spesso definita dall'utente, e sono progettati per scenari in cui i dati sono non strutturati o semi-strutturati. Sono particolarmente efficaci per applicazioni con elevati carichi di lettura e analisi, come il trattamento di "big data". I dati possono essere organizzati gerarchicamente invece che in tabelle piatte, migliorando la gestione di grandi volumi di dati e le prestazioni nelle operazioni concorrenti. Esempio: **MongoDB**. Sono adatti a dati non strutturati, grandi volumi di dati o quando la flessibilità e la scalabilità sono prioritari.

Piattaforma di distribuzione

Scegliere se distribuire la propria applicazione su desktop o mobile è fondamentale.

Su telefoni o tablet, devi considerare che deve essere possibile fornire un servizio limitato senza connettività di rete. I dispositivi mobili hanno processori meno potenti, quindi è necessario minimizzare le operazioni che richiedono molta elaborazione. La durata della batteria è limitata, quindi dovresti cercare di minimizzare il consumo di energia della tua applicazione. Le tastiere su schermo sono lente e soggette a errori, quindi dovresti minimizzare l'input tramite tastiera per ridurre la frustrazione.

Per affrontare queste differenze, spesso è necessario avere versioni separate del front-end del prodotto per browser e dispositivi mobili.

Server

Una decisione chiave è se progettare il tuo sistema per funzionare su server di proprietà o sul cloud. Se scegli di sviluppare per il cloud, devi progettare l'architettura come un sistema a servizi e utilizzare le API della piattaforma cloud per implementare il tuo sw.

Per i prodotti destinati ai consumatori, ha quasi sempre senso sviluppare per il cloud. Alcune aziende sono preoccupate per la sicurezza nel cloud e preferiscono eseguire i loro sistemi su server interni. Se prevedono un utilizzo prevedibile, potrebbe non essere necessario progettare il software per gestire grandi variazioni di carico.

Tecnologia di sviluppo

Le tecnologie di sviluppo, come i framework influenzano l'architettura del software. Queste tecnologie incorporano spesso ipotesi e principi predefiniti sull'architettura di sistema, ai quali è necessario adattarsi per sfruttarle correttamente.

I programmati tendono a preferire strumenti e framework con cui hanno maggiore familiarità, ma queste scelte potrebbero non essere sempre le più adatte per garantire la scalabilità e la sostenibilità del progetto a lungo termine. Pertanto, è essenziale valutare le tecnologie non solo per la loro immediatezza d'uso, ma anche per il loro impatto futuro sull'intero sistema.

Cloud-Based Software

Tony: *What is Docker? What is Docker Compose? What are the differences between multi-tenant and multi-instance SaaS systems? What are the factors that influence the design of a cloud software architecture?*

Il **cloud** è un numero molto grande di **server remoti offerti in affitto** da aziende che possiedono questi server.

Puoi **affittare** quanti server desideri, **eseguire** il tuo software su questi server e renderli **disponibili** ai tuoi clienti. I tuoi clienti possono accedere a questi server dai loro dispositivi connessi alla rete. L'hw offerto dal cloud è così potente da poter eseguire facilmente diversi server virtuali contemporaneamente su ogni nodo.

Aziende cloud come Amazon e Google forniscono software di gestione del cloud che facilita l'acquisizione e il rilascio di server su richiesta. Puoi affittare un server per un periodo di tempo contrattualizzato o affittare e pagare i server on demand. Quindi, se hai bisogno di risorse solo per un breve periodo, paghi semplicemente per il tempo di cui hai bisogno. I server cloud che affitti possono essere avviati e spenti a seconda delle variazioni della domanda.

I principali vantaggi del cloud sono scalabilità, elasticità e resilienza.

La **scalabilità** riflette la capacità del sw di **adattarsi automaticamente** quando il carico aumenta a del crescente numero di utenti, così da mantenere le prestazioni e il tempo di risposta del sistema.

Si parla di **scalabilità verticale (scaling up)** quando viene utilizzato un server più potente. Si parla di **scalabilità orizzontale (scaling out)** quando vengono create copie del sw e eseguite su server aggiuntivi dello stesso tipo.

L'**elasticità** riflette la capacità del sw di monitorare la domanda sulla tua applicazione e aggiungere o **rimuovere** server dinamicamente man mano che cambia il numero di utenti, consentendo quindi di **ridurre** la scala oltre che di aumentarla. Ciò significa che paghi solo per i server di cui hai bisogno, quando ne hai bisogno.

La **resilienza** riflette la capacità del sw di tollerare i guasti del server. Possono essere rese disponibili diverse copie del sw contemporaneamente, localizzate in luoghi diversi. **Se una di queste fallisce, le altre continuano a fornire il servizio.**

I **vantaggi** dell'utilizzo del cloud si riflettono in diversi fattori. Per quanto riguarda i costi, si evita l'investimento iniziale di capitale necessario per l'acquisto dell'hardware. Il tempo di avvio è significativamente ridotto, poiché non è necessario attendere la consegna dell'hardware: i server possono essere attivati in pochi minuti. In termini di scelta dei server, il cloud permette una grande flessibilità: se i server affittati non sono sufficientemente potenti, è possibile scalarli. Infine, il cloud facilita lo sviluppo

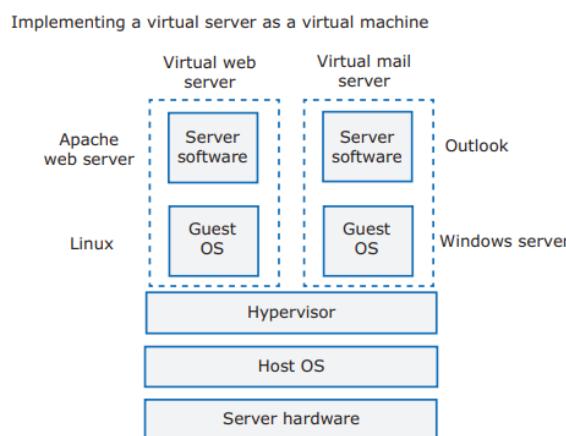
distribuito: i team che lavorano in diverse località possono accedere allo stesso ambiente di sviluppo, condividendo informazioni senza difficoltà.

5.1 Virtualization and containers

Un **server virtuale** è un'istanza software che **simula un server fisico**. Viene eseguito su un **computer fisico** sottostante e include un sistema operativo e una serie di pacchetti software progettati per fornire le funzionalità del server richieste. Essendo indipendente dall'hw fisico, un server virtuale può essere eseguito su qualsiasi hw nel cloud. Tutti i server cloud sono server virtuali, in quanto basati su questa tecnologia.

La caratteristica di "**funzionare ovunque**" è possibile perché il server virtuale non ha dipendenze esterne. Quando esegui software su diversi computer, spesso incontri problemi perché alcuni dei software esterni da cui dipendi non sono disponibili o sono diversi rispetto alla versione che stai utilizzando. Un server virtuale consente di configurare un ambiente in cui carichi tutto il software necessario per le tue applicazioni. In questo modo non dipendi da software fornito da altri.

Una **Virtual Machine (VM)** è un'istanza virtualizzata che **simula un computer fisico**, completa di sistema operativo e risorse hardware simulate. Le VM vengono eseguite su server fisici e sono utilizzate per **implementare server virtuali**, consentendo di isolare e gestire ambienti separati sullo stesso hardware.



Un **hypervisor** è un software che consente la creazione e gestione di Virtual Machines (VM) su un computer fisico, che agisce come uno **strato intermedio** tra l'hardware fisico e le VM, **simulando il funzionamento** dell'hardware sottostante e distribuendo le risorse fisiche tra le VM. Permette a più sistemi operativi di funzionare contemporaneamente su un unico hardware. Ci sono due tipi di hypervisor:

Hypervisor di tipo 1: Opera direttamente sull'hardware fisico, senza un sistema operativo intermedio.

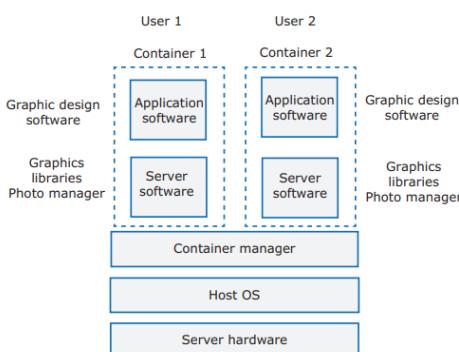
Hypervisor di tipo 2: Funziona come un'applicazione su un sistema operativo preesistente.

Il problema dell'implementazione di server virtuali su VM è che creare una VM comporta l'overhead del tempo necessario per installare l'OS e configurare il sw necessario sulla VM varia tipicamente tra 2 e 5 minuti. Ciò significa che non si può reagire istantaneamente alle variazioni della domanda avviando o spegnendo VM.

In molti casi, non si ha bisogno della generalità di una macchina virtuale. Se si sta eseguendo un sistema basato su cloud con molte istanze di applicazioni o servizi, questi usano tutti lo stesso sistema operativo. Per affrontare questa situazione, possono essere utilizzati i container.

Un **container** è una **tecnologia di virtualizzazione** leggera che consente di eseguire **applicazioni** sullo stesso SO dell'host in **modo isolato**. Ogni container include **tutto ciò che serve** per l'esecuzione dell'applicazione, come codice, librerie, dipendenze e file di configurazione, ma condivide il kernel del SO dell'host, riducendo il consumo di risorse rispetto alle macchine virtuali (VM).

Figure 5.3 Using containers to provide isolated services



Rispetto alle VM, i container sono molto più efficienti: occupano pochi megabyte anziché gigabyte e possono essere avviati o arrestati in pochi secondi, contro i minuti richiesti per una VM. Questo li rende particolarmente adatti per ambienti cloud in cui è necessario scalare rapidamente le applicazioni in risposta alla domanda.

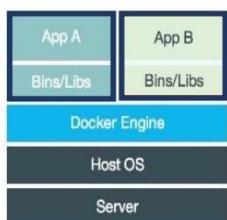
I container virtualizzano solo lo spazio utente e le risorse di sistema, **isolando i processi tra loro** senza la necessità di un sistema operativo separato per ogni istanza. Questo approccio elimina l'overhead associato all'installazione e configurazione di un sistema operativo completo, rendendo i container ideali per eseguire molteplici istanze di applicazioni o servizi sullo stesso sistema operativo.

Per decenni, il comando **chroot** di UNIX ha fornito una semplice forma di isolamento del filesystem. Nel **1998**, l'utilità **jail** di FreeBSD ha esteso il sandboxing di chroot anche ai processi. Nel **2005**, Google ha iniziato a sviluppare **CGroups** per il kernel Linux e a spostare la sua infrastruttura sui container. Nel **2008**, **Linux Containers (LXC)** ha fornito una soluzione completa di containerizzazione.

Nel **2013**, **Docker** ha aggiunto le parti mancanti, come immagini portabili e un'interfaccia user-friendly, rendendo i container una tecnologia diffusa.

Docker

Docker è una piattaforma che utilizza la tecnologia dei container per **sviluppare, distribuire ed eseguire** applicazioni basate su container in un ambiente isolato. Docker utilizza funzionalità come *namespace* e *control groups* del kernel Linux per isolare i processi e limitare l'uso delle risorse.



Componenti principali di Docker

Docker Engine: Il motore che consente di costruire, gestire ed eseguire container.

Docker Hub: Un registro pubblico dove è possibile condividere e scaricare immagini Docker preconfigurate.

Immagini Docker: Template di sola lettura che definiscono tutto l'ambiente e il software necessario all'interno del container. È una sorta di istantanea del sistema in un determinato momento, contenente il sistema operativo, il software e il codice dell'applicazione, tutto raccolto in un file. Le immagini sono create utilizzando file di configurazione chiamati Dockerfile.

Container: Ambienti di esecuzione isolati, istanza di esecuzione di un'immagine. Ogni container è indipendente e può essere avviato o arrestato rapidamente.

Dockerfile: È un file di testo che contiene le istruzioni per creare un'immagine Docker. Definisce da quale immagine di base partire, quali pacchetti installare e come configurare il sistema per eseguire un'applicazione.

Docker sfrutta la virtualizzazione basata su container per eseguire più istanze isolate sullo stesso sistema operativo.

Le immagini vengono definite utilizzando un **Dockerfile**. Scrivi un Dockerfile, lo compilhi per ottenere un'immagine, e poi puoi eseguire quell'immagine per ottenere un container.

Le immagini Docker vengono memorizzate in un Docker registry, che può essere pubblico (come **Docker Hub**) o privato. I registri sono strutturati in **repository**, che contengono diverse immagini per le varie versioni di un software.

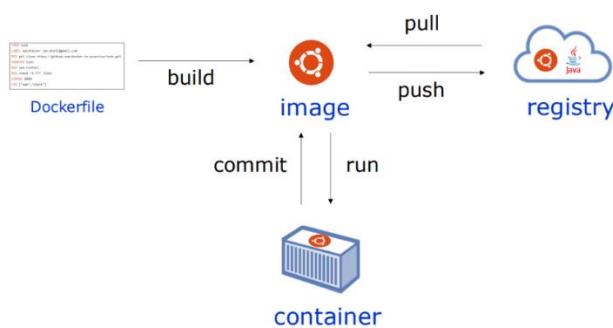
Le immagini Docker sono composte da livelli di sola lettura (layers). Ogni livello contiene modifiche rispetto al livello sottostante. Il **livello più basso** è chiamato **base image**, che fornisce un ambiente di base.

Quando un container viene eseguito da un'immagine, Docker crea un nuovo livello in cima (di lettura-scrittura) dove possono essere fatti cambiamenti. Le modifiche fatte nel container possono essere **committate** in una nuova immagine, che aggiunge un nuovo livello sopra l'immagine originale.

Per garantire la persistenza dei dati, è possibile montare volumi esterni all'interno dei container.

Funzionamento di Docker

Docker commands



Build: Il comando `docker build` prende il Dockerfile e lo usa per creare un'**immagine**.

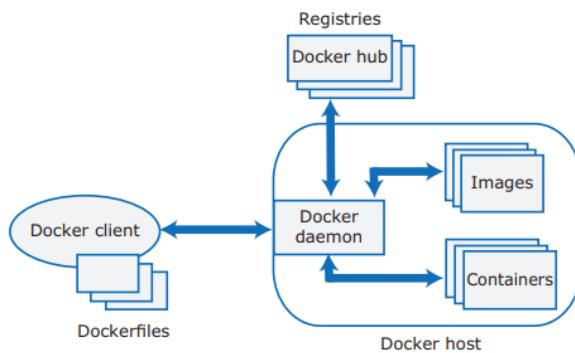
Push: Usando `docker push`, puoi caricare un'immagine in un **Docker registry**.

Pull: Usando `docker pull`, puoi scaricare un'immagine da un registro e utilizzarla per creare container.

Run: Il comando `docker run` avvia un container a partire da un'immagine Docker.

Commit: Se apporti modifiche a un container in esecuzione, puoi usare docker commit per salvare queste modifiche in una nuova immagine.

Figure 5.4 The Docker container system



Docker client: È l'interfaccia che sviluppatori e amministratori di sistema usano per interagire con Docker. Invia comandi al Docker daemon, come `docker build` o `docker run`, per costruire o eseguire i container. Il client lavora con i **Dockerfile**, che contengono le istruzioni per creare immagini Docker.

Docker daemon: È il cuore del sistema Docker. Questo processo in esecuzione sul **Docker host** riceve comandi dal Docker client e si occupa di eseguire tutte le operazioni, come la creazione, gestione e esecuzione dei container. Interagisce anche con i registri e gestisce le immagini.

Docker host: È la macchina fisica o virtuale su cui Docker è in esecuzione, che ospita il Docker daemon, le immagini e i container.

Il **Docker client** invia comandi al **Docker daemon**, che gestisce le immagini e i container sul **Docker host**. Le immagini possono essere scaricate o caricate dai **registri**, come **Docker Hub**, e i container sono creati a partire dalle immagini e gestiti dal daemon.

Docker Compose

Docker Compose è uno strumento offerto da Docker che **semplifica** la gestione di applicazioni che utilizzano **più container**, consentendo di definire l'intera configurazione dell'applicazione in un unico file YAML (`docker-compose.yml`). Questo approccio elimina la necessità di eseguire manualmente più comandi Docker per avviare e configurare i vari container e le risorse associate, come reti e volumi.

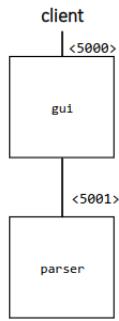
Con Docker Compose, tutte le configurazioni dei servizi (container), delle reti e dei volumi vengono specificate nel file `docker-compose.yml`. Una volta configurato, l'intera applicazione può essere avviata con un singolo comando, **docker-compose up**, che costruisce tutti i container istanze delle immagini specificate nei Dockerfile, crea e gestisce i volumi per la persistenza dei dati, configura le reti per consentire la comunicazione tra i container. E avvia tutti i container seguendo le dipendenze definite.

YAML File

Services: Definisce i servizi (i container) che compongono l'applicazione. In questo caso ci sono due servizi: parser e gui.

YAML File structure

```
services:  
  parser:  
    build: ./parser  
    container_name: parser  
    restart: always  
    volumes:  
      - myvolume:/app:rw  
  
  gui:  
    build: ./gui  
    container_name: gui  
    ports:  
      - 5000:5000  
    depends_on:  
      - parser  
    volumes:  
      myvolume:
```



build: ./parser indica che il Dockerfile per costruire l'immagine di questo servizio si trova nella cartella ./parser.

container_name: parser assegna un nome personalizzato al container.

restart: always fa sì che Docker riavvii automaticamente il container parser in caso di arresto, utile per la resilienza.

volumes: Monta un volume chiamato myvolume nella directory /app del container parser, con permessi di lettura e scrittura (rw). Questo volume permette la condivisione di dati tra i container parser e gui o per la persistenza di dati tra riavvii.

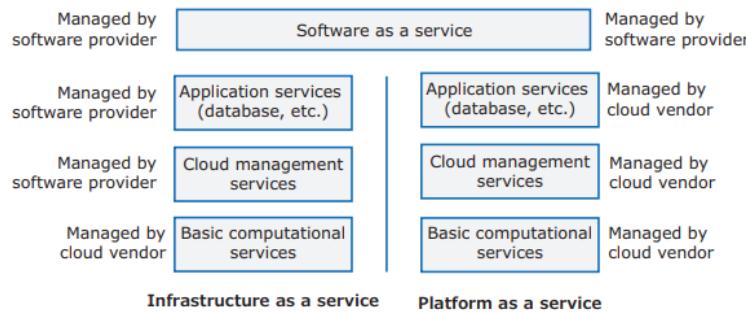
ports: 5000:5000 mappa la porta 5000 del container gui sulla porta 5000 del host, rendendo accessibile l'interfaccia gui all'esterno del container.

depends_on: parser specifica che gui dipende dal servizio parser, quindi Docker avvierà prima il container parser.

5.2 Everything as a Service

"Everything as a Service" (XaaS) è un modello di business che si basa sull'idea di **noleggiare** piuttosto che acquistare risorse, siano esse hw, sw o infrastrutture. In questo approccio, gli utenti possono accedere a ciò di cui hanno bisogno, quando ne hanno bisogno, pagando per l'utilizzo piuttosto che per il possesso. Questo concetto è alla base del cloud computing, dove anziché possedere server fisici o hw costosi, è possibile noleggiarli da un provider cloud, utilizzandoli per ospitare applicazioni o prodotti software da distribuire ai clienti.

Figure 5.6 Management responsibilities for SaaS, IaaS, and PaaS



Infrastructure as a service (IaaS): IaaS è un modello che offre ai clienti **l'accesso a risorse hw** virtuali come server, archiviazione e reti. Il provider si occupa di gestire tutta l'**infrastruttura fisica**, come il mantenimento e l'aggiornamento dell'hardware, la sicurezza a livello di infrastruttura e la connettività di rete. Come cliente, sei **responsabile** di gestire tutto ciò che gira sopra l'infrastruttura virtuale, come l'installazione e la gestione del SO, delle applicazioni e della sicurezza a livello

software. I principali vantaggi dell'utilizzo di IaaS sono che non si sostengono i **costi** di capitale per l'acquisto di hardware e si può facilmente **migrare** il software da un server a un server più **potente**. Alcuni esempi di IaaS sono: **EC2 (Elastic Compute Cloud)** di AWS, che permette di creare server virtuali e **S3 (Simple Storage Service)**, sempre di AWS, che offre spazi di archiviazione scalabili.

Platform as a service (PaaS): PaaS è un modello che offre una **piattaforma** completa per sviluppare, testare e distribuire applicazioni senza preoccuparti dell'infrastruttura sottostante. Il provider PaaS mette a disposizione tutto il necessario per sviluppare un'applicazione, includendo **Virtual Machine, SO, servizi integrati come DB e Software Development Kits**. Come cliente, la tua unica responsabilità è **installare** e gestire la tua **applicazione**. Non devi preoccuparti dell'infrastruttura, dei server o del SO, ma ti concentri solo sul far funzionare correttamente l'applicazione che hai creato. Alcuni esempi di servizi PaaS includono **Heroku, Azure e Google App Engine (GAE)**.

Software as a service (SaaS): SaaS è un modello che fornisce **software** pronto all'uso, accessibile su richiesta tramite un browser web. Il provider di SaaS **gestisce tutto**: l'infrastruttura, il SO e l'applicazione stessa. Come cliente, non hai nessuna responsabilità: Todo quello che devi fare è accedere al software e usarlo. Esempi sono Gmail, Word o saleforces.com.

Function as a Service (FaaS): FaaS è un modello, supportato dal servizio **Lambda** di Amazon, che consente agli sviluppatori di caricare il codice direttamente sul provider cloud, che si occupa automaticamente di creare, avviare e chiudere i server necessari ogni volta che la funzione viene richiesta. Questo elimina la necessità di gestire, configurare o mantenere server fisici o virtuali. Si paga solo per il tempo in cui la funzione è in esecuzione, eliminando i costi fissi associati al noleggio continuo di server.

5.3 Software as a Service (SaaS)

Quando i prodotti software furono introdotti, dovevano essere installati sui PC del cliente. A volte l'acquirente doveva configurare il sw per adattarlo al proprio ambiente operativo e gestire gli aggiornamenti. Il sw aggiornato non era sempre compatibile con gli altri sw aziendali, per cui era comune che gli utenti continuassero a utilizzare versioni più vecchie per evitare problemi di compatibilità. Questo significava che le aziende produttrici dovevano mantenere diverse versioni del loro prodotto contemporaneamente. Molti prodotti sw vengono ancora distribuiti in questo modo, ma sempre più spesso vengono forniti as a Service.

Un'azienda che fornisce il suo prodotto sw come servizio lo **esegue sui propri server**, che potrebbe affittare da un provider cloud. I clienti **non devono installare** il sw e vi accedono tramite browser o app. Il modello di pagamento per SaaS è solitamente un **abbonamento**, in cui gli utenti pagano una quota mensile per utilizzare il sw, garantendo un flusso di denaro per l'azienda costante durante l'anno.

L'azienda ha il **controllo degli aggiornamenti** del prodotto, e tutti i clienti ricevono l'aggiornamento **contemporaneamente**. L'azienda può distribuire nuove versioni del sw non appena le modifiche sono state apportate e testate.

L'azienda può offrire diverse opzioni di pagamento, attirando una gamma più ampia di clienti. Può rendere disponibili **versioni gratuite** o a basso costo del software, permettendo ai clienti di provarlo prima di acquistarlo.

L'azienda può **raccogliere dati** su come il prodotto viene utilizzato, identificando aree di miglioramento.

Il modello SaaS offre numerosi **vantaggi**. I clienti possono accedere al software **ovunque** e su **qualsiasi dispositivo**, come mobile, laptop o desktop, garantendo una grande flessibilità operativa. Non ci sono **costi iniziali**, poiché non è necessario

investire in software o server; si paga solo per l'utilizzo effettivo. Inoltre, i clienti ricevono automaticamente gli aggiornamenti più recenti del software, eliminando la necessità di gestire installazioni o manutenzione, riducendo così i costi di gestione.

Tuttavia, ci sono anche degli **svantaggi**. La conformità alle **normative sulla privacy**, come quelle rigide dell'UE, può rappresentare una sfida per il trattamento dei dati personali. Esistono inoltre preoccupazioni sulla sicurezza, poiché i clienti potrebbero essere **riluttanti a delegare il controllo dei propri dati** a un fornitore esterno. I **vincoli di rete** possono influire sulla velocità di risposta, specialmente in caso di trasferimento di grandi quantità di dati. Anche **l'integrazione con altri sistemi** può risultare complicata se il cloud non dispone di **API adeguate**. Inoltre, i clienti perdono il controllo sugli aggiornamenti, non potendo scegliere quando implementare le nuove versioni del software. Infine, si rischia un **lock-in** del servizio, che rende complesso cambiare fornitore una volta adottato un determinato sistema.

Se stai sviluppando un sistema che non gestisce informazioni personali e finanziarie, SaaS è di solito il modo migliore per distribuire il tuo prodotto. Devi utilizzare un provider cloud che memorizza i dati in **luoghi consentiti** dalle normative. Se questo non è praticabile, potrebbe essere necessario installare il sw e memorizzare i dati sui **server del cliente**.

Un prodotto software può essere progettato in modo tale che alcune funzionalità vengano eseguite **localmente** nel browser o nell'app dell'utente, e **altre** su un server **remoto**. L'esecuzione locale riduce il traffico di rete e quindi aumenta la **velocità** di risposta dell'utente, il che è utile quando gli utenti hanno una connessione di rete lenta. Tuttavia, l'elaborazione locale aumenta la potenza elettrica necessaria per far funzionare il sistema. Le scelte progettuali devono direzionare l'esecuzione secondo le esigenze di **velocità** o **risparmio di energia** in caso di mobile.

Su tutti i sistemi condivisi, gli utenti devono autenticarsi per dimostrare di essere autorizzati a utilizzare il sistema. Molti sistemi consentono agli utenti di autenticarsi utilizzando le credenziali di **Google, Facebook o LinkedIn**. Tuttavia, questo non è solitamente accettabile per le aziende, che preferiscono che i loro utenti si autentichino utilizzando le credenziali aziendali.

La **fuga di informazioni** è un rischio particolare per il software basato su cloud. Se hai più utenti da organizzazioni diverse, un rischio per la sicurezza è che le informazioni possano trapelare da un'organizzazione all'altra. Devi essere molto attento a progettare il tuo sistema di sicurezza per evitarlo.

Sistemi multi-tenant e multi-instance

I sistemi basati sul Cloud possono essere di due tipi, multi-tenant, in cui tutti i clienti sono serviti da un'unica istanza del sistema, o multi-instance, dove ogni cliente è servito da una copia del sistema dedicata.

Sistemi multi-tenant

Un sistema **multi-tenant** è un sistema in cui tutti i clienti sono serviti da un'**unica istanza** del sistema e un database multi-tenant. I clienti interagiscono con ciò che sembra essere un sistema dedicato alla loro azienda. Il database unico è partizionato in modo che ogni azienda cliente abbia il proprio spazio e possa archiviare e accedere ai propri dati.

In un **database multi-tenant**, un **singolo** schema di database è condiviso **tra più** utenti, dove ogni utente o **azienda** è rappresentato da un "**tenant**".

Tenant: Identificatore univoco (prima colonna del database) che rappresenta l'azienda o l'utente che possiede i dati.

L'architettura sfrutta un identificatore tenant per **distinguere** i dati di ciascun tenant, permettendo a ogni tenant di accedere esclusivamente ai propri dati, simulando l'esperienza di avere un database dedicato, pur utilizzando uno schema condiviso.

An example of a multi-tenant database

Stock management					
Tenant	Key	Item	Stock	Supplier	Ordered
T516	100	Widg 1	27	S13	2017/2/12
T632	100	Obj 1	5	S13	2017/1/11
T973	100	Thing 1	241	S13	2017/2/7
T516	110	Widg 2	14	S13	2017/2/2
T516	120	Widg 3	17	S13	2017/1/24
T973	100	Thing 2	132	S26	2017/2/12

Vantaggi: Il provider SaaS ha il controllo su **tutte le risorse** utilizzate dal software e può **ottimizzarle** per sfruttarle al meglio. La gestione della sicurezza è semplificata poiché c'è **una sola copia** del software del database **da correggere** se viene scoperta una vulnerabilità. È più **facile aggiornare** una singola istanza del software piuttosto che più istanze. A tutti i clienti viene consegnata la **versione più recente** del software.

Svantaggi: I clienti devono tutti utilizzare lo stesso schema di database con possibilità **limitate di adattarlo** alle proprie esigenze individuali. Esiste una possibilità teorica che i **dati** possano **trapelare** da un cliente all'altro. Se si verifica una violazione della sicurezza del database, allora questa **colpisce tutti** i clienti. I sistemi multi-tenant sono solitamente più **complessi** a causa della necessità di gestire molti utenti.

Le aziende di grandi dimensioni raramente vogliono utilizzare sw multi-tenant generico, ma vogliono una **versione personalizzata** adattata ai propri requisiti. Potrebbero volere che gli utenti si autenticassero utilizzando le **credenziali aziendali** piuttosto che le credenziali impostate dalla piattaforma. Potrebbero desiderare un'**interfaccia utente brandizzata** per riflettere l'identità dell'azienda. Potrebbero voler essere in grado di **estendere lo schema** utilizzato nel DB per soddisfare le proprie esigenze. Potrebbero voler definire la propria **politica di controllo degli accessi** che stabilisce quali utenti possono accedere ai dati e le operazioni consentite su di essi.

In un sistema multi-tenant, dove tutti gli utenti condividono un'unica copia del sistema, l'interfaccia utente e il controllo degli accessi devono essere configurabili, permettendo di creare "database virtuali" per ciascun cliente. La configurabilità dell'interfaccia può essere implementata tramite **profili utente**, che includono informazioni su aspetto, permessi di accesso e funzionalità disponibili per quella determinata azienda.

Quando un prodotto SaaS rileva un utente di una specifica organizzazione, utilizza il profilo aziendale per generare una versione personalizzata dell'interfaccia. Questo profilo può includere elementi come il nome, il logo dell'azienda e impostazioni di sicurezza. Inoltre, ogni utente può avere un profilo personale che definisce le funzionalità e i dati accessibili.

Personalizzazione schema multi-tenant

I clienti aziendali potrebbero voler estendere lo schema del DB per soddisfare le loro esigenze specifiche. Ci sono due modi per farlo se si utilizza un sistema di database relazionale:

Aggiungere un numero di campi extra a ciascuna tabella: Si aggiungono alcune colonne extra a ciascuna tabella del database e si consente ai clienti di utilizzare questi campi come desiderano.

Figure 5.12 Database extensibility using additional fields

Stock management								
Tenant	Key	Item	Stock	Supplier	Ordered	Ext 1	Ext 2	Ext 3
T516	100	Widg 1	27	S13	2017/2/12			
T632	100	Obj 1	5	S13	2017/1/11			
T973	100	Thing 1	241	S13	2017/2/7			
T516	110	Widg 2	14	S13	2017/2/2			
T516	120	Widg 3	17	S13	2017/1/24			
T973	100	Thing 2	132	S26	2017/2/12			

Svantaggi: È difficile sapere quanti campi extra includere. Se ce ne sono troppo pochi, i clienti potrebbero non avere abbastanza spazio per le loro esigenze. Se si inseriscono troppi campi per un singolo cliente tiene conto dei clienti, la maggior parte dei restanti clienti non li utilizzerà, causando molto spazio sprecato nel database. Inoltre, diversi clienti potrebbero richiedere tipi di colonne differenti (si può mantenere tutti i campi extra come stringhe).

Aggiungere un campo a ciascuna tabella che identifichi una "extension table"

separata: Consiste nell'includere un campo aggiuntivo nella tabella principale che funge da chiave esterna per una tabella esterna. Questa tabella esterna è composta da due livelli:

Tabella dei metadati: Contiene i dettagli dei campi estesi, quali il tenant dell'azienda, il nome del campo e il tipo di dato associato (ad esempio, stringa, booleano, data).

Tabella dei valori dei campi estesi: Registra i valori specifici per ogni campo esteso. Ogni riga include l'identificativo del record (presente nella tabella principale come chiave esterna), il tenant associato e il valore del campo.

Main database table						
Tab1	Stock management					
Tenant	ID	Item	Stock	Supplier	Ordered	Ext 1
T516	100	Widg 1	27	S13	2017/2/12	E123
T632	100	Obj 1	5	S13	2017/1/11	E200
T973	100	Thing 1	241	S13	2017/2/7	E346
T516	110	Widg 2	14	S13	2017/2/2	E124
T516	120	Widg 3	17	S13	2017/1/24	E125
T973	100	Thing 2	132	S26	2017/2/12	E347

Field names			Field values			Tab3
Tenant	Name	Type	Record	Tenant	Value	
T516	'Location'	String	E123	T516	'A17/S6'	
T516	'Weight'	Integer	E123	T516	'4'	
T516	'Fragile'	Boolean	E123	T516	'False'	
T632	'Delivered'	Date	E200	T632	'2017/1/15'	
T632	'Place'	String	E200	T632	'Dublin'	
T973	'Delivered'	Date	E346	T973	'2017/2/10'	

Extension table showing the field names for each company that needs database extensions

Value table showing the value of extension fields for each record

La principale preoccupazione dei clienti aziendali con i database multi-tenant è la sicurezza. Poiché le informazioni di tutti i clienti sono memorizzate nello stesso DB, un attacco potrebbe esporre i dati di tutti i clienti.

Si intende per **multilevel access control** che l'accesso ai dati deve essere controllato sia a livello organizzativo che a livello individuale. Devi avere il controllo dell'accesso a livello organizzativo per assicurarti che qualsiasi operazione sul database agisca solo sui dati di quell'organizzazione. Quindi, il primo passaggio è eseguire l'operazione sul database selezionando gli elementi etichettati con l'identificatore dell'organizzazione. Gli utenti individuali che accedono ai dati devono anche avere i propri permessi di accesso. Pertanto, devi fare una selezione ulteriore dal database per presentare solo quegli elementi di dati che un utente identificato è autorizzato ad accedere.

La crittografia dei dati in un **database multi-tenant** rassicura le aziende che i loro dati non possono essere visualizzati da persone di altre aziende in caso di guasti. I dati crittografati vengono archiviati e decifrati solo quando sono accessibili con la chiave appropriata. A causa della complessità computazionale, i database multi-tenant crittografano **solo i dati sensibili**.

Sistemi multi-instance

Un'alternativa ai sistemi multi-tenant sono i sistemi multi-instance. Un sistema **multi-instance** fornisce una **copia separata** del sistema e del DB per **ciascun utente**. Questi sono chiamati sistemi **multi-instance**.

Ogni cliente ha il proprio sistema adattato alle proprie esigenze. I sistemi multi-instance basati sono **più semplici** rispetto ai sistemi multi-tenant e evitano preoccupazioni legate alla sicurezza, come la fuga di dati da un'organizzazione all'altra. Esistono due tipi di sistemi multi-instance.

Sistemi multi-instance basati su VM: In questa architettura, ogni cliente dispone di **un'istanza dedicata** del software e del database, eseguiti su una **macchina virtuale** separata. Questo approccio garantisce un elevato livello di isolamento tra i clienti, offrendo loro un controllo completo sull'accesso e sull'utilizzo del proprio sw e dei propri dati. Sebbene possa sembrare un'opzione **costosa**, è ideale per clienti aziendali che richiedono **accesso continuo** e affidabile (24/7) al loro sw, oltre a personalizzazioni specifiche. Tutti gli utenti appartenenti allo stesso cliente accedono a una singola istanza condivisa del sistema.

Sistemi multi-instance basati su container: In questa architettura, ogni utente ha una versione **isolata** del SW e del DB che viene eseguita in un **insieme di container**. Generalmente, il sw utilizza un'architettura a **microservizi**, con **ciascun servizio** che viene eseguito in un container e gestisce il proprio DB. Questo approccio è ideale per prodotti con utilizzo indipendente da parte degli utenti e condivisione limitata dei dati, risultando più adatto per individui o prodotti aziendali non intensivi in termini di dati.

È possibile eseguire container su una macchina virtuale, quindi è anche possibile creare sistemi ibridi in cui un'azienda potrebbe avere il proprio sistema basato su VM e poi eseguire container su di esso per utenti individuali.

Vantaggi: Ogni istanza del software può essere personalizzata e adattata alle esigenze del cliente. Ogni cliente ha il proprio database, quindi non c'è possibilità di fuga di dati da un cliente all'altro. Le istanze del sistema possono essere scalate in base alle esigenze dei singoli clienti. Se si verifica un guasto del software, questo probabilmente interesserà solo un singolo cliente.

Svantaggi: È più costoso utilizzare sistemi multi-instance a causa dei costi per affittare molte VM nel cloud e dei costi di gestione di più sistemi. Molte istanze devono essere aggiornate, quindi gli aggiornamenti sono più complessi.

Architettura software nel cloud

Se decidi di utilizzare il cloud come piattaforma di distribuzione, dovrai prendere diverse decisioni architettoniche specifiche per il cloud. Le domande da porsi sono: Il software dovrebbe utilizzare **un database** multi-tenant o multi-instance? Quali sono i requisiti di **scalabilità e resilienza** del software? La struttura del software dovrebbe essere **monolitica** o orientata ai **servizi**? Quale **piattaforma cloud** dovrebbe essere utilizzata per lo sviluppo e la distribuzione?

Scelta di organizzazione del database

Esistono tre modi possibili per fornire un database per i clienti in un sistema basato sul cloud: **multi-tenant**, in cui il DB è condiviso da tutti i clienti del tuo prodotto ed è ospitato nel cloud in server potenti; **multi-instance**, in cui ciascun database cliente viene eseguito su una propria **macchina virtuale**; **multi-instance**, in cui ciascun database viene eseguito in un proprio **container**(il DB potrebbe essere distribuito su più container, uno per ogni servizio).

Se il cliente non è interessato ad avere branding e personalizzazione, l'uso di un sistema di autenticazione locale o permessi individuali variabili, può utilizzare un database multi-tenant con un unico schema. Altrimenti, è consigliato un DB multi-instance.

Se il database deve essere coerente in ogni momento, è richiesto un sistema basato su transazioni. Dovresti utilizzare quindi un db multi-tenant o un db multi-instance eseguito su una macchina virtuale.

Se il tuo cliente necessita di un unico grande db relazionale con molte tabelle collegate, è consigliato l'approccio multi-tenant. Tuttavia, se il db può essere limitato in dimensioni e non ha molte tabelle collegate, potrebbe essere possibile suddividere il database in più piccoli db indipendenti, ognuno dei quali può essere implementato come un'istanza separata che viene eseguita in un proprio container, usando l'approccio multi-instance su container.

I clienti aziendali spesso vogliono trasferire informazioni tra i loro db locali e il tuo db basato su cloud mentre utilizzano il tuo prodotto. Poiché questi clienti non utilizzeranno tutti la stessa tecnologia di database e schemi, è molto più semplice utilizzare un db multi-instance.

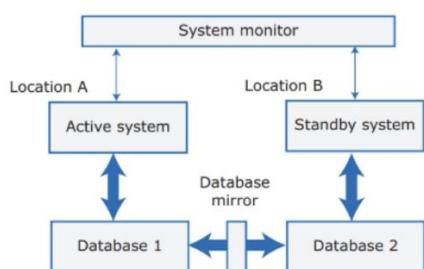
Se stai costruendo il tuo sistema come un sistema orientato ai servizi, l'approccio multi-instance su container è la scelta ideale.

Scelte di scalabilità e resilienza

La **scalabilità** di un sistema riflette la sua capacità di **adattarsi** automaticamente ai **cambiamenti del carico** su quel sistema. La **resilienza** riflette la capacità di un sistema di **continuare a fornire servizi critici** **in caso di guasti** o uso malevolo del sistema.

Per ottenere la resilienza, devi essere in grado di **riavviare** il software **rapidamente** dopo un guasto hw o sw.

La resilienza può essere garantita tramite il seguente approccio: Le repliche del sw e dei dati sono mantenute in diverse località. Gli aggiornamenti del db sono replicati, in modo che il db di riserva sia una copia funzionante del db operativo. Un monitor di sistema controlla continuamente lo stato del sistema e può passare automaticamente al sistema di riserva se quello operativo fallisce.



Se un server fisico fallisce o se c'è un guasto più ampio nel data center, l'operazione può essere commutata automaticamente alle copie del sw eseguite su server fisici situati in datacenter di diversa località. Se le copie del sw vengono eseguite in parallelo, la commutazione potrebbe essere completamente trasparente, senza effetti sugli utenti.

Un sistema di **hot standby** mantiene una sincronizzazione quasi in tempo reale tra i dati delle diverse località, garantendo un ritardo minimo nell'attivazione del sistema secondario in caso di guasto.

Un sistema di **cool standby** prevede il ripristino dei dati da un backup e la ripetizione delle transazioni per aggiornare il sistema allo stato precedente al guasto. Il backup viene effettuato periodicamente e il sistema rimane non operativo fino al completamento del processo di ripristino, risultando in un tempo di inattività maggiore rispetto al modello hot standby.

Scelta della struttura del software

Architettura client-server tradizionale: Architettura costruita attorno a un db condiviso, in cui il sistema è monolitico e distribuito su più server. Tutta la logica di business e il processamento sono implementati in un **unico sistema**, con grandi componenti software che lavorano insieme come un unico programma.

Architettura orientata ai servizi (Service-Oriented): Architettura che scomponete il sistema in **servizi indipendenti**. Ogni servizio non conserva lo stato tra le richieste, rendendolo replicabile, distribuibile e migrabile facilmente tra i server. Questa architettura è particolarmente adatta per software cloud-native con servizi distribuiti in container, migliorando la scalabilità e la flessibilità.

E' raccomandato di utilizzare normalmente un approccio monolitico per costruire il prototipo, poiché i framework includono un supporto che permette di implementare rapidamente sistemi basati su MVC. In prodotti sw basati sul cloud, parti del prodotto diverse potrebbero dover essere aggiornate in momenti differenti e potresti dover scalare rapidamente solo un servizio piuttosto che altri. In tali circostanze, raccomando di utilizzare un approccio basato su microservizi.

Scelta della piattaforma cloud

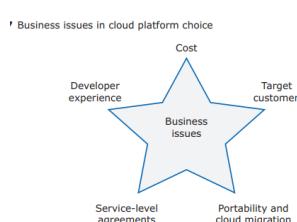
Dovresti scegliere una piattaforma cloud in base alla tua esperienza, al tipo di prodotto che stai sviluppando e alle aspettative dei tuoi clienti.

Le questioni da prendere in considerazione sono le seguenti. Alcuni provider cloud forniscono spesso altri servizi cloud utili, come database, servizi di big data, e così via, che possono ridurre i costi di sviluppo, quindi dovresti scegliere un provider che **fornisce servizi cloud utili** alla tua applicazione o **compatibili** con servizi già utilizzati.

Se è probabile che il tuo sistema subisca picchi di domanda significativi e necessita di **scalare**, dovresti scegliere un provider che offre librerie PaaS che facilitino la scrittura di software elastico. Inoltre, la **resilienza** si basa sulla replica, quindi è necessario utilizzare un provider che abbia data center in **località diverse** e che supporti la replica tra queste.

Alcuni paesi, come l'Unione Europea, hanno requisiti rigidi sulla protezione dei dati e su dove i dati vengono archiviati. Se hai clienti in diversi paesi, devi utilizzare un provider cloud che abbia data center internazionali e che possa fornire **garanzie sui luoghi di archiviazione**.

Questioni commerciali nella scelta della piattaforma cloud



Il **costo** è ovviamente un fattore critico nella scelta di una piattaforma cloud. Sebbene possa essere allettante scegliere il fornitore che sembra offrire i costi più bassi, devi sempre chiederti perché quel fornitore è più economico e quali compromessi sono stati fatti per offrire un servizio a prezzo inferiore.

Se il tuo team di sviluppo **ha esperienza** con una particolare piattaforma cloud, ha senso utilizzare quella piattaforma se possibile, in modo che non dovrà dedicare tempo all'apprendimento di un nuovo sistema.

Considera le **aspettative dei clienti** e l'interoperabilità del tuo software con le piattaforme che già **utilizzano**, come Salesforce, Microsoft Azure o il cloud di fornitori fidati come IBM. Per mercati specifici, scegli la piattaforma cloud più adatta a quel mercato.

I tuoi clienti si aspetteranno un certo livello di servizio e, per garantirlo, avrai bisogno di un livello di servizio comparabile dal tuo provider cloud. I **Service Level Agreement** stabiliscono **il servizio che il provider garantisce di fornire e le penalità se non lo fa**. Se hai requisiti specifici e sei un cliente importante, dovresti scegliere un provider disposto a negoziare gli SLA. I grandi provider come Amazon e Google offrono un SLA "prendere o lasciare" senza spazio per negoziazioni.

Quando scegli un provider, devi considerare la possibilità che in futuro tu possa voler **spostare** il sw su un altro provider. I container hanno semplificato i problemi di migrazione cloud. Tuttavia, se utilizzi servizi della piattaforma del cloud nel tuo codice, questi dovranno essere reimplementati.

Microservizi

Tony: *What are the main (dis)advantages and characteristics of microservices? How do microservices communicate? What is the CAP theorem?*

La decomposizione dei componenti è cruciale perché questi possono poi essere sviluppati in parallelo da team diversi. Possono essere riutilizzati e sostituiti se la tecnologia sottostante cambia e possono essere distribuiti su server.

Per sfruttare i vantaggi del sw basato sul cloud—scalabilità, affidabilità ed elasticità—è necessario utilizzare componenti che possano essere facilmente replicati, eseguiti in parallelo e spostati tra server virtuali. Questo è difficile con componenti, come gli oggetti, che mantengono uno stato locale, poiché è necessario trovare un modo per mantenere la coerenza dello stato tra i componenti. Pertanto, è meglio utilizzare servizi software **stateless** che mantengano informazioni persistenti in un **database locale**.

Un **servizio** sw è un componente software che può essere accessibile da remoto **tramite Internet**. Dato un input, un servizio produce un output corrispondente, **senza effetti collaterali**. Il servizio è accessibile tramite la sua **interfaccia API** pubblicata e tutti i dettagli dell'implementazione del servizio sono nascosti. Il gestore di un servizio è chiamato fornitore del servizio, mentre l'utente di un servizio è chiamato richiedente del servizio.

I servizi non mantengono uno stato interno, ma le informazioni di stato sono memorizzate in un db. Quando viene effettuata una richiesta di servizio, le informazioni di stato possono essere incluse come parte della richiesta e lo stato aggiornato viene restituito come parte del risultato del servizio. Poiché non c'è stato locale, i servizi possono essere **riassegnati** dinamicamente da un server virtuale a un altro e possono essere **replicati** per gestire l'aumento delle richieste, scalando orizzontalmente.

Ripensando cosa è un servizio, Amazon ha concluso che un servizio dovrebbe essere correlato a una **singola funzione di business**. I servizi dovrebbero essere completamente **indipendenti**, con il proprio database. Dovrebbero gestire la propria interfaccia utente.

Sostituire o replicare un servizio dovrebbe essere possibile **senza dover modificare** altri servizi.

I **microservizi** sono servizi su piccola scala, **stateless**, che hanno una **singola responsabilità**. Se è necessario creare prodotti software basati sul cloud che siano adattabili, scalabili e resilienti, allora si consiglia di utilizzare un'architettura a microservizi.

Per identificare i microservizi che potrebbero essere utilizzati, è necessario suddividere le funzionalità a grana grossa in funzioni più dettagliate. Ciascun microservizio deve gestire i propri dati. Se si hanno servizi molto specifici, i dati devono spesso essere replicati tra diversi servizi. Esistono vari modi per mantenere i dati coerenti, ma tutti potenzialmente rallentano il sistema.

6.1 Microservizi

I **microservizi** sono servizi su **piccola scala** che possono essere **combinati** per creare applicazioni. Devono essere **indipendenti**, in modo che l'interfaccia API del servizio non sia influenzata da modifiche ad altri servizi. Deve essere possibile modificare e ridistribuire il servizio senza cambiare o interrompere altri servizi nel sistema.

I microservizi non hanno **dipendenze esterne**. Gestiscono i **propri dati** e implementano la propria interfaccia utente. Comunicano tra loro utilizzando **protocolli leggeri**, quindi i sovraccarichi di comunicazione del servizio sono bassi. Possono essere implementati utilizzando **diversi linguaggi di programmazione** e possono utilizzare tecnologie diverse, come differenti tipi di database. Ogni microservizio viene eseguito nel **proprio processo** ed è **distribuito in modo indipendente**, utilizzando **container**. I microservizi dovrebbero implementare capacità e necessità aziendali, piuttosto che fornire semplicemente un servizio tecnico.

I microservizi comunicano **scambiando messaggi**. Un messaggio inviato tra servizi include alcune informazioni amministrative, una richiesta di servizio e i dati necessari per fornire il servizio richiesto. I servizi restituiscono una risposta ai messaggi di richiesta di servizi, che include i dati che rappresentano la risposta alla richiesta di servizio.

Alta coesione e basso accoppiamento

L'obiettivo nella progettazione di un microservizio dovrebbe essere quello di creare un servizio con alta coesione e basso accoppiamento.

Coesione: Misura del numero di relazioni che le parti di un componente hanno tra loro.

Accoppiamento: Misura del numero di relazioni che un componente ha con altri componenti nel sistema.

Un basso accoppiamento è importante nei microservizi perché porta a servizi indipendenti. Finché si mantiene inalterata l'interfaccia, è possibile aggiornare un servizio senza dover cambiare altri servizi nel sistema. Un'alta coesione è importante perché significa che il servizio è completo e non deve chiamare molti altri servizi durante l'esecuzione. Chiamare altri servizi comporta un sovraccarico di comunicazioni, che può rallentare un sistema.

Principio di responsabilità singola: Ogni elemento di un sistema dovrebbe fare solo una cosa e farla bene.

Dimensione di un microservizio: Deve essere possibile sviluppare, testare e distribuire un microservizio in **due settimane** o meno. La dimensione del team dovrebbe essere 8-10 persone.

Il team è responsabile anche di sviluppare tutto il codice necessario per garantire che un microservizio sia completamente indipendente, di testare i servizi e del supporto del servizio dopo che è stato distribuito.

Ogni servizio deve includere, oltre alle proprie funzionalità, il codice di supporto che può essere condiviso in un sistema monolitico (come Message, Failure e Data Consistency management, GUI). Per molti servizi, è necessario implementare più codice di supporto rispetto al codice che fornisce la funzionalità del servizio.

Architettura a microservizi

Un'architettura a **microservizi** è uno stile architettonico per implementare architetture client-server logiche, in cui il server è realizzato come un **insieme di microservizi** interagenti.

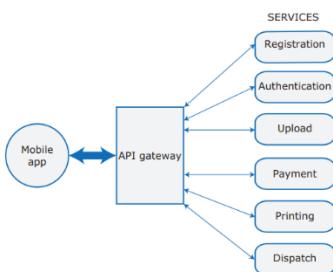
Le motivazioni che hanno portato a questa architettura sono le seguenti. Quando si utilizza un'architettura monolitica, l'intero sistema deve essere ricostruito, ristato e ridistribuito ogni volta che viene apportata una modifica.

Quando la domanda sul sistema aumenta, l'intero sistema deve essere scalato, anche se la domanda è localizzata a un piccolo numero di componenti del sistema che implementano le funzionalità più richieste.

Un monolite viene eseguito come un unico processo. I microservizi sono autonomi e vengono eseguiti in **container separati**. Può essere fermato e riavviato senza influenzare altre parti del sistema. Se aumenta la domanda di un servizio, possono essere create e distribuite rapidamente repliche del servizio, distribuendole su un numero aumentato di server (**scaling out**).

Un **API Gateway** è un servizio centrale che funge da unico **punto di accesso** tra i client e i microservizi del sistema. Traduce le richieste dei client in **chiamate appropriate** ai microservizi, esponendo **un'interfaccia API** con cui è possibile chiamare le **funzionalità dei servizi** e nascondendo i dettagli dei protocolli di comunicazione utilizzati. Consente così di modificare la struttura dei microservizi senza impattare il funzionamento delle applicazioni client.

Figure 6.5 A microservices architecture for a photo-printing system



6.2.1 Architectural design decisions

Uno delle decisioni più importanti è decidere come l'intero sistema debba essere **scomposto** in un insieme di microservizi. Troppi microservizi significano che ci saranno molte comunicazioni tra servizi, rallentando il sistema. Avere troppo pochi microservizi implica che ciascun servizio debba gestire più funzionalità e dipendenze, rendendo più complessa la loro manutenzione.

Nei servizi con singola funzione, le modifiche sono tipicamente limitate a pochi servizi. Tuttavia, se ciascuno dei servizi offre solo un singolo servizio molto specifico, sarà inevitabile avere più comunicazioni tra servizi per implementare la funzionalità utente, portando ad un rallentamento del sistema.

Gli elementi di un sistema che probabilmente saranno cambiati contemporaneamente dovrebbero essere collocati all'interno dello stesso servizio (*common closure principle*)

Una **business capability** è un'area di funzionalità di business che è responsabilità di un individuo o di un gruppo. È necessario identificare i servizi richiesti per supportare ciascuna business capability.

I servizi devono essere progettati in modo che abbiano accesso solo ai dati di cui hanno bisogno. In situazioni in cui i dati utilizzati da diversi servizi si sovrappongono, è necessario avere un meccanismo che garantisca che le modifiche ai dati in un servizio vengano propagate agli altri servizi.

Un possibile punto di partenza per l'identificazione dei microservizi è esaminare i dati che i servizi devono gestire. Di solito ha senso sviluppare microservizi attorno a dati logicamente coerenti. Questo evita il problema di dover coordinare le azioni di diversi servizi per garantire che i dati condivisi siano coerenti.

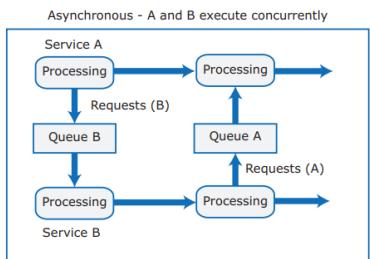
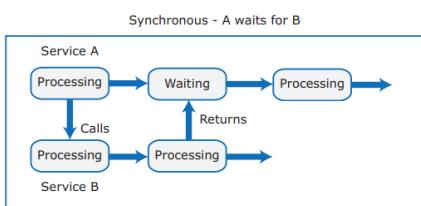
Il modo migliore per identificare i microservizi in un sistema è partire da un'architettura monolitica multi-tier. Una volta che si ha esperienza di un sistema e alcuni dati su come viene utilizzato, è molto più facile identificare la funzionalità che dovrebbe essere incapsulata in microservizi. Si dovrebbe quindi rifactorizzare il proprio software monolitico in un'architettura a microservizi.

Comunicazioni tra servizi

Quando si progetta un'architettura a microservizi, è necessario stabilire uno standard per le comunicazioni tra microservizi.

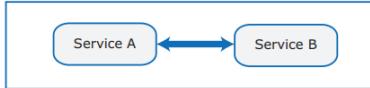
In un'interazione **sincrona**, il servizio A invia una richiesta al servizio B. Il servizio A sospende quindi l'elaborazione e aspetta fino a quando il servizio B restituisce le informazioni necessarie, prima di riprendere l'esecuzione.

In un'interazione **asincrona**, il servizio A invia la richiesta che viene messa in coda per essere elaborata dal servizio B. Il servizio A continua quindi a elaborare senza attendere che il servizio B termini i suoi calcoli. Più tardi, il servizio B completa la richiesta inviata dal servizio A e mette in coda il risultato per essere recuperato dal servizio A. Pertanto, il servizio A deve controllare periodicamente la sua coda per vedere se un risultato è disponibile.



I programmi sincroni sono più facili da scrivere e conterranno meno bug. L'interazione asincrona è spesso più efficiente, poiché i servizi non rimangono inattivi in attesa di una risposta. I servizi che interagiscono in modo asincrono sono debolmente accoppiati, il che rende più facili le modifiche a questi servizi.

Direct communication - A and B send messages to each other



Indirect communication - A and B communicate through a message broker



La **comunicazione diretta** tra i servizi richiede che i servizi coinvolti conoscano gli indirizzi URL reciproci. I servizi interagiscono inviando richieste direttamente a questi indirizzi.

La comunicazione **indiretta** prevede la denominazione del servizio richiesto e l'invio della richiesta a un message broker, che è responsabile di trovare il servizio che può soddisfare la richiesta e inoltrarla ad esso.

La comunicazione diretta tra i servizi è solitamente più veloce, ma implica che il servizio richiedente debba conoscere l'URI del servizio richiesto. Se quell'URI cambia, la richiesta del servizio fallirà.

La comunicazione indiretta richiede un message broker aggiuntivo, ma i servizi vengono richiesti per nome anziché per URI. Il message broker trova l'indirizzo del servizio richiesto e dirige la richiesta.

L'uso di un **message broker** permette l'interazione sia sincrona che asincrona tra i servizi, semplificando la modifica e la replica dei servizi grazie alla separazione delle loro responsabilità. Tuttavia, introduce maggiore complessità nel sistema e può ridurre le prestazioni a causa della latenza aggiuntiva nella comunicazione.

Un protocollo di messaggio è un accordo tra i servizi che stabilisce come devono essere strutturati i messaggi. Il message broker rifiuta i messaggi che non seguono la definizione.

I servizi RESTful seguono lo stile architettonico REST con i dati del messaggio rappresentati in JSON.

6.2.3 Distribuzione e condivisione dei dati

Ogni microservizio dovrebbe gestire i **propri dati**. Nel mondo reale, l'indipendenza completa dei dati è impossibile, ci saranno sempre sovrapposizioni tra i dati utilizzati in diversi servizi. La **condivisione** dei dati deve essere **ridotta** al più possibile. Se la condivisione dei dati è inevitabile, devi progettare i microservizi in modo che la maggior parte della condivisione sia in **sola lettura**, con un **numero minimo** di servizi responsabili degli **aggiornamenti** dei dati. Se i servizi sono replicati nel tuo sistema, devi includere un meccanismo che possa mantenere **coerenti le copie** del db utilizzate dai servizi replicati.

Senza **controlli di coerenza**, se due servizi A e B stanno aggiornando gli stessi dati, il valore di quei dati dipende dal **timing** degli aggiornamenti. Una transazione **ACID** raggruppa un **set** di aggiornamenti di dati in **un'unica unità**, in modo tale che o tutti gli aggiornamenti vengono completati o nessuno di essi viene eseguito. In questo modo il db è sempre coerente perché in caso di errore, non ci sono aggiornamenti parziali ai dati.

In qualsiasi sistema distribuito, c'è un compromesso tra consistenza dei dati e prestazioni. Più sono rigorosi i requisiti di consistenza dei dati, maggiore è il lavoro computazionale necessario per garantire che i dati siano coerenti.

Due tipi di inconsistenza devono essere gestiti:

Incoerenza dei dati dipendenti: Le azioni o i fallimenti di un servizio possono causare l'**incoerenza** dei dati gestiti da un altro servizio.

Incoerenza delle repliche: Diverse repliche dello stesso servizio possono essere eseguite contemporaneamente. Ognuna ha la propria copia del database e aggiorna la propria copia dei dati del servizio. È necessario un modo per rendere questi database "eventualmente consistenti", in modo che tutte le repliche lavorino sugli stessi dati.

La **consistenza eventuale** significa che il sistema garantisce che i database diventeranno alla fine coerenti. Quando viene effettuata una modifica al db, essa viene registrata in un **log** degli "aggiornamenti in sospeso". Altre istanze del servizio consultano questo log, aggiornano il proprio database e indicano di aver apportato la modifica. Dopo che tutti i servizi hanno aggiornato il proprio database, la transazione viene rimossa dal log.

Quando un servizio inizia a elaborare una richiesta di servizio, la replica del servizio che gestisce la richiesta **controlla il log** per vedere se i dati necessari in quella richiesta sono stati aggiornati. Se sì, aggiorna i propri dati dal log e poi avvia la propria operazione. Altrimenti, il database può essere aggiornato dal log quando il carico sul servizio è relativamente **leggero**.

CAP Theorem

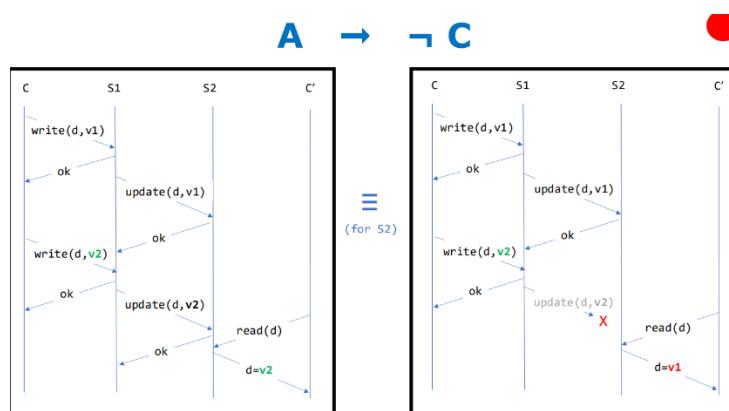
Il **CAP Theorem** afferma che è impossibile per un sistema distribuito soddisfare simultaneamente tre proprietà fondamentali:

Consistenza (Consistency): Ogni richiesta riceve una risposta con i **dati più aggiornati**, cioè i dati sono corretti e consistenti in tutto il sistema.

Disponibilità (Availability): Ogni richiesta **riceve una risposta**, anche se non garantisce che i dati siano aggiornati.

Tolleranza alle partizioni (Partition-tolerance): Il sistema continua a funzionare anche se la rete diventa inaffidabile o ci sono partizioni, cioè i messaggi possono essere persi o ritardati tra nodi.

$$\begin{aligned} &\equiv \neg(C \wedge A \wedge P) \\ &\equiv \neg(C \wedge A) \quad [P=\text{true in distributed systems}] \\ &\equiv \neg C \vee \neg A \quad [\text{De Morgan}] \\ &\equiv A \rightarrow \neg C \quad [\text{De Morgan}] \end{aligned}$$



Poiché non è possibile avere tutte e tre le proprietà simultaneamente, un sistema deve scegliere due delle tre. Dando P per sempre vera, ci sono quindi due soluzioni pratiche:

Soluzione 1 (A \wedge P): Possiamo garantire la **disponibilità** e la tolleranza alle partizioni, offrendo il **best-effort** nella consistenza. È spesso usata nei microservizi, dove la

consistenza forte **non è essenziale**. Ad esempio, Netflix può tollerare un ritardo nella consistenza dei dati di qualche ora.

Soluzione 2 (C ∧ P): Possiamo garantire la **consistenza forte** e la tolleranza alle partizioni, offrendo il **best effort** nella disponibilità. (Operazioni bancarie)

Saga Pattern

Il *Saga Pattern* è una soluzione per **gestire transazioni distribuite** che coinvolgono più servizi. Dato che non è possibile utilizzare transazioni distribuite tradizionali (ACID) nei microservizi, il pattern "saga" **scomponete una transazione globale** in una **sequenza di transazioni locali** eseguite in modo **indipendente** da ciascun servizio.

Example



Ogni transazione locale esegue **un'operazione** sul database e, se va a buon fine, **innesca** la transazione locale **successiva** nella saga. Se una delle transazioni locali **fallisce**, la saga esegue delle **transazioni di compensazione** per annullare le operazioni precedenti e riportare il sistema in uno stato coerente. Per coordinare le saghe, ci sono due modi:

Coreografia: Quando un servizio completa la sua operazione, pubblica un **evento** che segnala il **completamento** della sua transazione e **notifica** ad altri servizi che possono procedere con la transazione **successiva**.

Orchestrazione: Un **orchestratore centrale** **coordina** le transazioni, indicando ai servizi quali transazioni locali eseguire e quando.

Modello backward: Annulla le modifiche effettuate dalle transazioni locali già eseguite, ripristinando lo stato precedente.

Modello forward: Se una transazione fallisce, può essere riprovata in seguito, piuttosto che annullata immediatamente.

Approccio di Netflix per la replica dei dati

Netflix utilizza il seguente approccio per la **replica dei dati**. Quando un dato deve essere scritto, il sistema tenta di scrivere su **tutti** i nodi possibili. Tuttavia, se alcuni nodi **non sono raggiungibili** in quel momento, il sistema non si blocca in attesa che tutti i nodi rispondano. Scrive i dati su quelli **disponibili** e, in un secondo momento, corregge o **aggiorna** le repliche sui nodi mancanti quando tornano online.

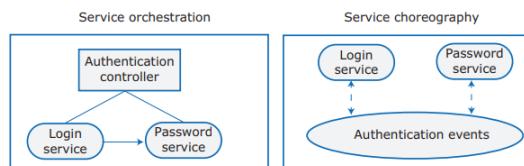
Netflix utilizza il concetto di **quorum** per garantire che una scrittura o lettura sia considerata valida. Il quorum definisce il **numero minimo** di repliche che devono rispondere per considerare **un'operazione** come **riuscita**. Per un sistema con "n" repliche, si richiede che almeno **(n/2 + 1)** delle repliche rispondano.

Coordinazione dei servizi

Orchestration: In un sistema orchestrato, un **controller centrale** gestisce il **coordinamento** dei servizi, indicando **come e quando** eseguire le loro attività. In caso di errore, il controller può identificare il servizio che ha fallito e il punto esatto del processo.

Choreography: In un sistema coreografato, i servizi si **autocoordinano** attraverso **eventi**. Ogni servizio emette un evento quando **completa** il proprio processo, e gli altri **reagiscono** agli eventi rilevati. Non esiste un controller centrale, ma è necessario un software come un **message broker** che implementi il meccanismo di **publish and subscribe**. Il monitoraggio e il recupero dai fallimenti richiedono strumenti aggiuntivi, poiché gli errori non sono immediatamente visibili.

Figure 6.11 Orchestration and choreography



Suggerimento: Preferire l'orchestrazione inizialmente e passare alla coreografia solo se il prodotto diventa rigido o difficile da aggiornare.

Failure management

Anche quando alcuni tipi di guasti hanno una bassa probabilità, se ci sono migliaia di istanze di servizi, i guasti si verificheranno inevitabilmente. I servizi devono essere tolleranti ai guasti.

Internal service failure: Si verifica quando il servizio identifica un problema interno, che può essere comunicato al cliente attraverso un messaggio di errore. Ad esempio, un servizio che accetta un URL come input e rileva che il link fornito non è valido.

External service failure: Questo tipo di guasto è causato da fattori esterni che compromettono la disponibilità del servizio. Può rendere il servizio non operativo e richiedere interventi per il ripristino, come il riavvio del sistema.

Service performance failure: Si verifica quando le prestazioni del servizio scendono sotto un livello accettabile, spesso a causa di un carico eccessivo o di problemi interni.

Il modo più semplice per segnalare i guasti nei microservizi è utilizzare i **codici di stato HTTP**, che indicano se una richiesta è riuscita o meno.

Un modo per scoprire se un servizio richiesto è non disponibile o sta funzionando lentamente è impostare un **timeout** sulla richiesta. Un timeout è un **contatore** associato alle richieste di servizio che inizia a contare quando la richiesta viene effettuata. Una volta che il contatore raggiunge un valore predefinito, come 10 secondi, il servizio chiamante presume che la richiesta **sia fallita** e agisce di conseguenza.

Il problema con l'approccio del timeout è che ogni nuova chiamata a un servizio "fallito" viene **ritardata** dal valore del timeout, quindi l'intero sistema rallenta. Invece di utilizzare esplicitamente i timeout, si suggerisce l'uso di un **circuit breaker**, che nega immediatamente l'accesso a un servizio fallito senza i ritardi associati ai timeout.

Il **circuit breaker** include un meccanismo di timeout che si avvia quando un servizio S1 invia una richiesta a un servizio S2. Se il servizio S2 risponde rapidamente, la risposta viene restituita al servizio chiamante. Se il servizio S2 non risponde dopo diversi tentativi, tuttavia, il circuit breaker **presume** che il servizio S2 abbia fallito e **"scatta"**. Quando viene effettuata una chiamata futura al servizio che sta andando in timeout, il circuit breaker **risponde immediatamente** con un codice di stato di fallimento. Il servizio richiedente non deve aspettare che il servizio richiesto vada in timeout per rilevare un problema.

Periodicamente, il circuit breaker invia una richiesta al servizio fallito. Se il servizio risponde rapidamente, il circuit breaker **"reimposta"** il circuito in modo che le chiamate future vengano nuovamente instradate verso il servizio ora disponibile.

Se qualcosa non funziona correttamente, l'applicazione dovrebbe essere in grado di continuare a funzionare senza impattare negativamente sull'esperienza utente.

In Spotify una ricerca può essere non andata a buon fine, ma anziché interrompere completamente il funzionamento dell'applicazione, il sistema riesce a reagire, correggere il problema, e continuare a operare.

Chaos Monkey è uno strumento sviluppato da Netflix che **interrompe casualmente** istanze di macchine virtuali o **container** all'interno di un ambiente di produzione. Questo tipo di test consente di valutare la resilienza dell'infrastruttura, verificando come si comporta quando alcune parti del sistema falliscono improvvisamente.

Monitoraggio

Un rischio generale nella distribuzione di nuovi servizi software è che problemi imprevisti possano essere causati dalle interazioni tra la nuova versione del servizio e i servizi esistenti. E' necessario monitorare i servizi distribuiti per rilevare eventuali problemi. Se un servizio fallisce, è possibile ripristinare una versione precedente del servizio.

Se utilizzi un **API gateway**, puoi fare ciò accedendo ai servizi tramite un collegamento alla "versione corrente". Quando introduci una nuova versione di un servizio, mantieni la vecchia versione, ma cambi il collegamento della versione corrente per puntare al nuovo servizio. Se il sistema di monitoraggio rileva problemi, reindirizza il collegamento alla versione precedente del servizio.

Conclusioni

Vantaggi: L'adozione dei microservizi consente di sviluppare e rilasciare nuove funzionalità più **velocemente**, perché i singoli team possono lavorare su componenti isolati. I microservizi permettono di **scalare** parti specifiche del sistema in modo indipendente, il che li rende ideali per ambienti cloud e sistemi con grande traffico.

Svantaggi: Poiché i microservizi devono comunicare costantemente tra loro tramite API o eventi, questo aumenta la complessità e l'overhead. La gestione di molti servizi separati aumenta la complessità complessiva del sistema. È necessario un alto livello di coordinazione e automazione per evitare problemi. Se i microservizi sono progettati male o suddivisi in modo inefficiente, può causare problemi di integrazione e di performance. Evitare la duplicazione dei dati pur mantenendo i microservizi isolati è una delle sfide più grandi.

"Un team scadente creerà sempre un sistema scadente." Anche il miglior approccio tecnologico non porterà a buoni risultati senza persone in grado di gestirlo correttamente.

I vantaggi in termini di velocità e scalabilità devono essere bilanciati con la complessità introdotta dall'architettura a microservizi. Non ha senso adottare i microservizi se il sistema è abbastanza semplice da essere gestito come un monolite.

Splitting the monolith

Tony: *How to split code and data when splitting a monolith?*

Il monolite cresce nel tempo acquisendo nuove funzionalità e righe di codice. Il monolite deve essere diviso in microservizi nel **momento** in cui **rappresenta un problema**.

Il monolite tende ad unire codice non correlato. Inoltre, una modifica ad una riga di codice tende ad impattare gran parte del resto del monolite, portando a dover ridistribuire l'intero sistema.

Seam: Un **seam** è un punto nel codice che può essere isolato per apportare modifiche senza influenzare altre parti del sistema. Rappresenta un'opportunità per separare responsabilità o funzionalità. (Una classe in Java che implementa un'interfaccia per comunicare con un database può essere testata o sostituita senza modificare il resto del sistema).

Bounded context: Un **bounded context** è un confine logico che definisce un'area del sistema in cui i modelli e le regole sono coerenti e separati da quelli di altri contesti. (In un sistema di e-commerce, "Inventory" e "Ordini" sono bounded context distinti, ognuno con responsabilità propri).

La maggior parte dei linguaggi di programmazione offre strumenti per raggruppare codice simile, come i concetti di **namespace** o di **package**. Ad esempio, in Java, i package permettono di organizzare il codice in gruppi logici, facilitando la gestione e il riutilizzo.

Split the monolith

Per cominciare, dovremmo identificare i *bounded contexts* di alto livello che pensiamo esistano nella nostra organizzazione. Dopodiché, vogliamo cercare di capire quali *bounded contexts* mappa il monolite. Il primo passo è **creare package** che rappresentino questi contesti e poi spostare il codice esistente al loro interno. Con gli IDE moderni, lo spostamento del codice può essere eseguito automaticamente tramite *refactoring*, ed è possibile farlo in modo incrementale mentre ci dedichiamo anche ad altre attività. Tuttavia, saranno necessari dei test per individuare eventuali problemi causati dallo spostamento del codice, soprattutto se si utilizza un linguaggio tipato dinamicamente, in cui gli IDE hanno maggiori difficoltà nel *refactoring*. Col tempo, iniziamo a vedere quale codice si adatta bene e quale rimane fuori, non trovando realmente posto in nessun contesto.

Il codice che non trova realmente posto in **nessun contesto** individua spesso un *bounded contexts* indipendente.

Durante questo processo possiamo utilizzare il codice per analizzare le dipendenze tra questi *package*. Il nostro codice dovrebbe rispecchiare la nostra organizzazione, quindi i nostri *package*, rappresentando i *bounded contexts* all'interno dell'organizzazione, dovrebbero interagire nello stesso modo in cui i gruppi reali della nostra organizzazione interagiscono.

Un approccio incrementale alla transizione verso i microservizi consente di ridurre l'impatto di eventuali errori. Invece di dividerlo interamente in un'unica fase, si agisce rimuovendo piccoli pezzi alla volta.

Una volta individuati i seam, devono essere considerati come da separare quelli dove è possibile ottenere il massimo vantaggio nel separarli dal *codebase*.

Aree di dominio della suddivisione

Frequenza di modifiche: Quando si prevede un'elevata frequenza di modifiche in una specifica area del sistema, è strategico identificare e separare il relativo seam come servizio autonomo.

Struttura del Team: La suddivisione del codice può essere guidata dalla struttura del team, assegnando ad un team piena proprietà di un'area specifica del sistema.

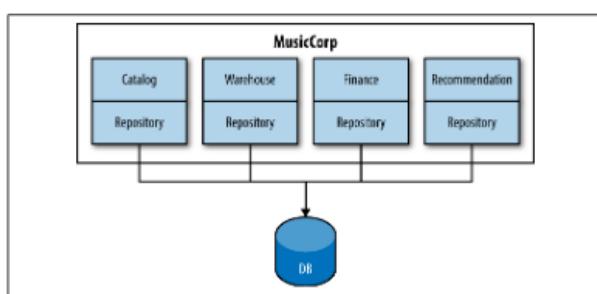
Sicurezza: Isolando le aree del sistema che gestiscono informazioni sensibili, è possibile introdurre livelli di protezione più specifici.

Tecnologia: Separare il codice legato a una funzionalità sperimentale, come una nuova libreria, consente di testare innovazioni senza compromettere la stabilità del sistema.

Dipendenze Intrecciate: I seam con meno dipendenze risultano più facili da isolare. Vogliamo estrarre il **seam meno dipendente** possibile. Rappresentare le dipendenze come un grafo aciclico diretto aiuta a individuare i componenti più intrecciati e a pianificare una separazione graduale ed efficace.

Il Database

Il database rappresenta spesso il punto più critico nella gestione delle dipendenze intrecciate. È necessario identificare i seam in esso in modo da poterlo suddividere. Il primo passo consiste nell'**analizzare il codice** per identificare quali parti **leggono e scrivono** nel database. È utile suddividere il livello repository (accesso al DB) in più parti, ciascuna dedicata ad ogni bounded context identificato.



Tuttavia, è possibile che esistano vincoli come relazioni di chiave esterna tra tabelle utilizzate da moduli diversi, che possono ostacolare la separazione. In questi casi, è fondamentale pianificare attentamente la suddivisione per eliminare o ristrutturare tali vincoli. L'obiettivo è suddividere il database, anche quando le stesse tabelle vengono

Eliminare le Relazioni di Chiave Esterna

In un'architettura monolitica, le relazioni tra tabelle, come i vincoli di chiave esterna, garantiscono la consistenza dei dati. Quando i moduli vengono separati in servizi autonomi, i vincoli di chiave esterna non possono più essere mantenuti a livello di database. Un servizio che **accede** direttamente alla tabella di un **altro servizio** crea una **dipendenza** indesiderata e vanifica l'obiettivo di una separazione netta tra i servizi.

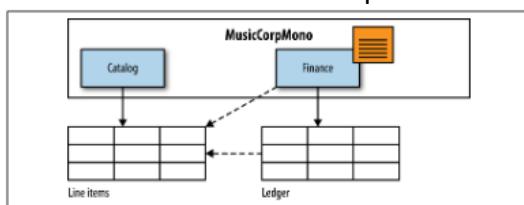


Figure 5-2. Foreign key relationship

La **soluzione** consiste nell'**evitare accessi diretti** al database condiviso. Il servizio dovrebbe esporre i dati necessari all'altro servizio **tramite un'API** che l'altro servizio può utilizzare. La separazione può comportare dover effettuare più chiamate per raccogliere dati distribuiti tra diversi servizi. Sebbene le prestazioni peggiorano, l'impatto dovrebbe essere valutato in relazione ai benefici di modularità e scalabilità. Un lieve rallentamento è un compromesso accettabile.

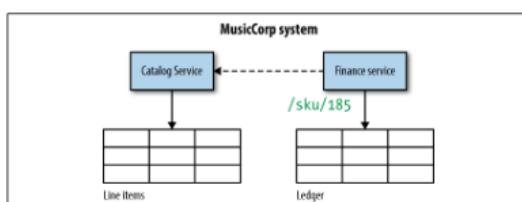


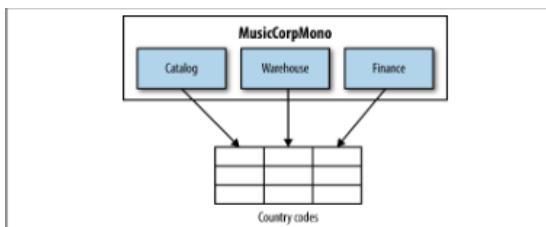
Figure 5-3. Post removal of the foreign key relationship

Con la **rimozione dei vincoli di chiave esterna**, la consistenza tra i servizi deve essere gestita

a livello applicativo. Ciò potrebbe richiedere l'implementazione di meccanismi personalizzati per controllare la validità dei dati o eseguire operazioni di pulizia per mantenere la coerenza tra i servizi.

Gestione dei Dati Statici Condivisi

Quando diversi servizi leggono **dati statici** (di configurazione) provenienti dalla stessa tabella, è necessario decidere come gestire queste informazioni.



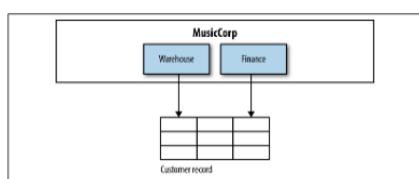
Una possibile **soluzione** consiste nel **duplicare la tabella** all'interno di ciascun servizio. Questo introduce il rischio di inconsistenze quando i dati vengono aggiornati in una tabella. È necessario adottare meccanismi per sincronizzare tutte le copie.

Un'alternativa più semplice è trattare i dati statici come **parte del codice o dei file di configurazione** distribuiti con il servizio. Sebbene i problemi di consistenza rimangano, la distribuzione di modifiche ai file di configurazione è generalmente più agevole rispetto agli aggiornamenti delle tabelle del database.

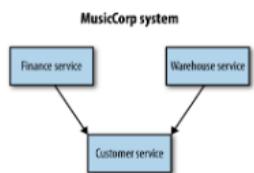
In casi particolari, quando i dati statici sono complessi o hanno regole specifiche associate, è possibile creare un **servizio autonomo** per gestirli. Tuttavia, questa soluzione è eccessiva.

Gestione dei Dati Mutabili Condivisi

Quando sono presenti **dati mutabili condivisi**, ovvero quando più moduli, **scrivono e leggono** dalla stessa tabella, si crea una dipendenza stretta che ostacola la modularità. Questo problema è spesso dovuto all'**assenza di un concetto di dominio chiaro** e modellato esplicitamente **nel codice** (servizio assente), ma **implicito** nel database. E' presente nel db un'unica tabella, che rappresenta il dominio di un servizio mancante, a cui tutti i servizi accedono.

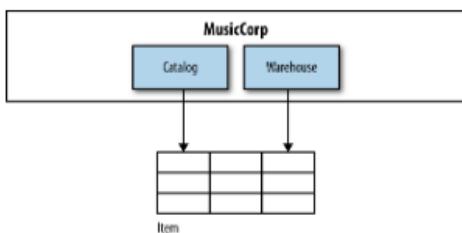


Per risolvere il problema, è necessario rendere **esplicito** il concetto di dominio assente, creando un **servizio dedicato**, che espone API per esporre i dati e le funzionalità agli altri moduli che ne hanno bisogno.

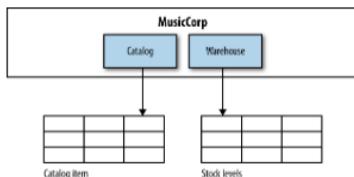


Gestione delle Tabelle Condivise

Le **tabelle condivise** rappresentano un problema in cui moduli diversi memorizzano i propri **dati (completamente) distinti**, in un'unica tabella. Quando il codice è monolitico, non è sempre evidente che una tabella condivisa stia in realtà combinando dati relativi a **concetti di dominio distinti**.



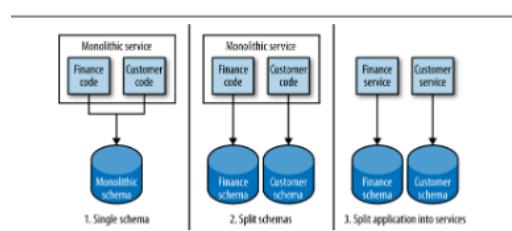
La soluzione consiste nel dividere la tabella condivisa **in più tabelle**, ognuna dedicata a un concetto di dominio specifico.



In questo modo, ogni modulo lavora esclusivamente sui dati rilevanti per il proprio dominio. Le modifiche ai dati di un modulo non influenzano gli altri. I moduli possono essere separati in microservizi senza dipendenze dirette sulle stesse risorse dati.

Pianificare la Separazione di un Sistema Monolitico

La **separazione di un sistema monolitico in microservizi** richiede una pianificazione attenta e graduale per ridurre i rischi e mantenere la stabilità del sistema. Il processo prevede tre fasi principali.



La **prima fase** consiste nell'**individuare i seam** nel codice dell'applicazione, **raggruppando** le funzionalità **attorno a bounded contexts**. Questi seam vengono usati per identificare le dipendenze anche a livello di db.

La **seconda fase** consiste nel creare **schemi distinti** nel **database** per ciascun bounded context, ma mantenere ancora il codice applicativo **unito**.

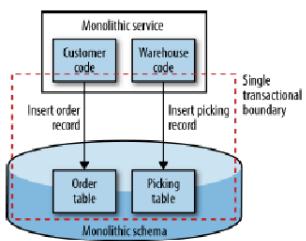
Una volta verificata l'efficacia della separazione degli schemi, nella **terza fase** il **codice** applicativo può **essere suddiviso in servizi distinti**, ciascuno responsabile del proprio schema. Questo passo finale completa la transizione verso un'architettura basata su microservizi.

La separazione graduale consente di sperimentare la suddivisione del database senza impatti significativi sugli utenti e permette di annullare o modificare le scelte fatte con un impatto minimo.

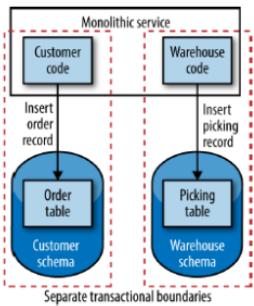
Confini Transazionali

Le transazioni garantiscono che una serie di operazioni avvenga tutta insieme o non avvenga affatto. Sono fondamentali per mantenere i dati consistenti, permettendo di annullare tutte le modifiche in caso di errore.

In un **sistema monolitico**, tutte le operazioni di creazione o aggiornamento avvengono tipicamente entro un **unico confine transazionale**. Questa configurazione garantisce che tutte le operazioni portino il sistema da uno stato consistente a un altro. Se qualcosa va storto, l'intera transazione viene annullata, evitando stati intermedi inconsistenti.



Quando separiamo il database in schemi distinti **perdiamo** la sicurezza di un singolo confine transazionale. Un'operazione si estende ora su più schemi e confini transazionali. Data un'operazione che adesso è divisa in due operazioni a causa della divisione degli schemi, se **una riesce e l'altra fallisce**, il sistema si troverà in uno **statuto incoerente**.



Gestione degli Errori: Ritentare l'Operazione

Ritentare un'operazione fallita è una strategia utile. Una parte dell'operazione che non è andata a buon fine viene registrata in una **coda** o in un **file di log** per essere riprovata successivamente.

Non si garantisce immediatamente uno stato consistente del sistema al termine della transazione, ma si accetta che il sistema raggiunga la consistenza in futuro, dopo che il ritentativo avrà successo (consistenza eventuale). Si presume che il problema che ha causato l'errore è temporaneo e un nuovo tentativo risolverà la situazione.

Gestione degli Errori: Annullare l'Intera Operazione

Annullare l'intera operazione è una strategia che utilizza le **transazioni di compensazione** (istr. `DELETE`) per annullare le operazioni già completate con successo, in modo da riportare il sistema allo stato precedente al fallimento, segnalando il problema all'utente.

In un sistema monolitico, la logica di compensazione è gestita centralmente. In un sistema distribuito, è necessario decidere quale servizio sarà responsabile della logica di compensazione. Se la transazione di compensazione stessa fallisce, il sistema potrebbe rimanere in uno stato incoerente. In questo caso, è possibile ritentare la transazione di compensazione o utilizzare un processo backend (manuale o automatico) per correggere l'incoerenza. La gestione delle transazioni di compensazione diventa più difficile man mano che aumenta il numero di operazioni interdipendenti.

Gestione errori: Transazioni Distribuite

Le **transazioni distribuite** mantengono la consistenza in sistemi distribuiti coordinando operazioni su più sistemi tramite un **transaction manager centrale**. Questo meccanismo unifica più transazioni locali in una singola operazione consistente, spesso utilizzando l'algoritmo del **Two-Phase Commit (2PC)**.

Nella **prima** fase, ogni partecipante comunica al **transaction manager** se la propria transazione può procedere. **Se tutti approvano**, il manager ordina il **commit**; in caso contrario, l'intera transazione **viene annullata**.

Le transazioni distribuite sono vulnerabili ai guasti: l'arresto del transaction manager o il mancato consenso di una coorte può **bloccare** l'intera transazione. Inoltre, l'uso di

lock sulle risorse durante il processo riduce la scalabilità del sistema. La complessità di implementazione richiede l'adozione di strumenti consolidati, come la **Transaction API di Java**, per evitare errori.

E allora che fare?

Chiediti se un'operazione richiede necessariamente una singola transazione o se può essere suddivisa in transazioni locali separate, accettando la **consistenza eventuale**. Se uno stato deve rimanere strettamente consistente, evita di separarlo. Se la separazione è inevitabile, adotta una visione più astratta, rappresentando la transazione come un concetto concreto a livello di dominio piuttosto che come un semplice processo tecnico.

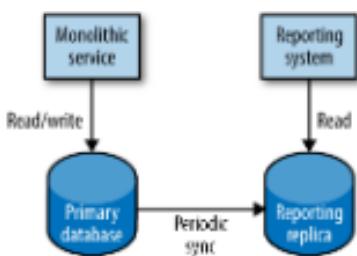
Reporting: Sfide e Soluzioni

Dividere un servizio monolitico in parti più piccole, come avviene nel passaggio ai microservizi, implica spesso la suddivisione dei dati. Tuttavia, questa trasformazione può creare problemi significativi per un caso d'uso comune: il reporting.

Considerazioni per il Reporting

Il reporting richiede l'aggregazione di dati provenienti da diverse parti dell'organizzazione per generare informazioni utili.

In un sistema monolitico, tutti i dati sono centralizzati in un unico database. Ciò rende il reporting semplice, poiché le query SQL possono accedere a tutti i dati necessari in un unico luogo. Per evitare di sovraccaricare il database principale, le query di reporting sono spesso eseguite su una **replica di lettura** sincronizzata periodicamente con il database primario. Lo schema del database serve sia il sistema applicativo sia il sistema di reporting. Qualsiasi modifica allo schema deve essere attentamente coordinata per evitare di interrompere entrambi i sistemi, rendendo i cambiamenti lenti e complessi. Spesso lo schema finisce per essere un compromesso, non eccellendo in nessuno dei due scenari.



I database relazionali tradizionali, pur offrendo compatibilità con strumenti di reporting, potrebbero non essere ottimali per i dati operativi o per analisi avanzate. Opzioni moderne come Neo4j (grafo), MongoDB (documenti) o Cassandra (orientato a colonne) potrebbero essere più adatte per scenari specifici, ma difficilmente coesistono con il modello monolitico tradizionale.

Adattarsi ai Cambiamenti

Il passaggio a microservizi distribuiti richiede un'attenzione particolare per mantenere la funzionalità di reporting.

Reporting nei Microservizi: Recupero Dati Tramite Chiamate di Servizio

Il reporting nei microservizi spesso richiede di **aggregare dati** provenienti da diversi sistemi, ma il **recupero diretto tramite chiamate API** presenta vantaggi e limiti che devono essere attentamente considerati.

Il recupero dei dati direttamente tramite API dai sistemi di origine può funzionare bene. Tuttavia, quando il reporting richiede grandi volumi di dati o analisi complesse, questo approccio diventa rapidamente inefficiente. Per report complessi, il recupero diretto comporta numerose chiamate API ai servizi coinvolti, rallentando significativamente l'elaborazione. Le API dei microservizi sono spesso progettate per supportare casi d'uso applicativi, non analitici. L'uso di proxy inversi con cache può migliorare le prestazioni, ma il reporting richiede spesso dati a lungo termine o poco richiesti, generando cache miss costosi. Molti strumenti di reporting richiedono dati in formato SQL. Estrarre i dati tramite API e convertirli periodicamente in un database SQL aggiunge complessità e latenza.

Per ottimizzare potremmo aggiungere API che consentano il recupero di più risorse in una sola chiamata. Ridurre il carico su HTTP salvando grandi quantità di dati in modalità batch in file scaricabili o condivisi.

Nonostante queste ottimizzazioni, il recupero diretto tramite API rimane inefficiente per il reporting tradizionale o per analisi su larga scala.

Data Pump: Un Approccio Alternativo per il Reporting

Il data pump è un metodo per inviare i dati dal database sorgente a un database di reporting. Invece di eseguire chiamate HTTP per recuperare i dati, un programma indipendente accede direttamente al database del servizio sorgente e trasferisce i dati nel database di reporting, riducendo il sovraccarico legato alle chiamate API.

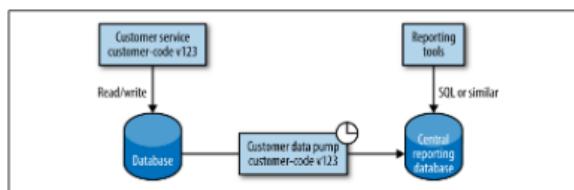


Figure 5-13. Using a data pump to periodically push data to a central reporting database

Si tratta di un programma a riga di comando, spesso gestito da un processo schedulato in background. Il data pump conosce sia il database del servizio sorgente sia lo schema di reporting, mappando i dati da uno schema all'altro. Lo stesso team responsabile del servizio gestisce il data pump, controllando le versioni e distribuendo gli artefatti del data pump insieme a quelli del servizio.

Lo schema del database di reporting deve essere considerato come un'API pubblica stabile, difficile da modificare, per ridurre l'impatto dei cambiamenti.

In un database relazionale, è possibile creare uno schema di reporting centralizzato utilizzando viste per aggregare i dati provenienti da ogni servizio.

Event Data Pump: Popolamento del Database di Reporting Tramite Eventi

L'**event data pump** utilizza flussi di eventi emessi dai microservizi per popolare un database di reporting. Ad esempio, un servizio clienti potrebbe emettere eventi quando un dato viene creato, aggiornato o eliminato. Questi eventi sono sottoscritti da un sistema che li mappa nel database di reporting centrale. Il data pump invia così i dati al sistema di reporting in tempo quasi reale.

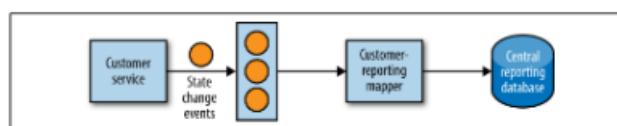


Figure 5-15. An event data pump using state change events to populate a reporting database

Architectural smells and refactoring for microservices

Un *architectural smell* è una **decisione architetturale** comunemente adottata che **influisce negativamente** sulle qualità del ciclo di vita del sistema.

Di seguito illustriamo come ciascun principio di design dei microservizi possa essere influenzato da ciascun *architectural smell* corrispondente, nonché come ciascun *smell* possa essere risolto applicando un *refactoring* corrispondente.

Design principles of microservices

1. I microservizi che formano un'applicazione devono essere indipendentemente distribuibili.
2. I microservizi devono essere scalabili orizzontalmente.
3. I *failures* devono essere isolati.
4. La decentralizzazione deve avvenire in tutti gli aspetti delle applicazioni basate su microservices, dalla gestione dei dati alla governance.

Principle 1: Independent deployability

Ciascun microservizio dovrebbe essere operativamente **indipendente** dagli altri, cioè dovrebbe essere possibile **distribuire** e rimuovere **un microservice indipendentemente dagli altri**.

Questo influenza sul deployment iniziale di un microservizio. Un microservizio può essere avviato senza attendere che altri microservizi siano in esecuzione. Deve essere possibile aggiungere/rimuovere repliche di un microservice a *runtime*.

Smell P1: Multiple services in one container

Ciascun microservizio può essere impacchettato in un'immagine di container, e diverse istanze di uno stesso microservizio possono essere avviate generando diversi container a partire dall'immagine corrispondente.

Un *architectural smell* per l'*independent deployability* dei microservizi consiste nel **posizionare più servizi in un solo container**.

Posizionando due microservizi nello stesso container, questi servizi diventerebbero operativamente dipendenti l'uno dall'altro, poiché non sarebbe possibile avviare una nuova istanza di uno di questi microservices senza avviare anche un'istanza dell'altro, e arrestare un servizio senza arrestare l'altro.

La **soluzione** a questo smell è il *refactoring* dell'applicazione in modo che **ciascun microservizio venga impacchettato in un'immagine di container separata**.



Principle 2 : Scalabilità orizzontale

La possibilità di aggiungere/rimuovere repliche di un microservizio è una diretta conseguenza dell'*independent deployability*. Per garantire la scalabilità orizzontale, tutte le repliche di un microservizio **devono essere raggiungibili** dai microservizi che lo invocano.

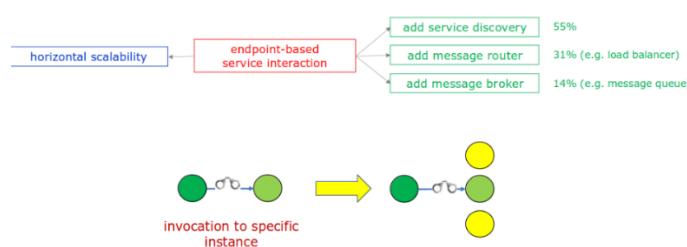
Smell 1 for P2: Endpoint-based service interactions

L'Endpoint-based service interactions **smell** si verifica quando uno o più microservizi in un'applicazione invocano una **specifica istanza** di un altro microservizio. In tal caso, scalando il microservizio tramite nuove repliche, queste non possono essere raggiunte dagli invocatori.

La **prima soluzione** consiste nell'utilizzare un **service discovery**, ossia un servizio che memorizza le posizioni effettive di tutte le istanze dei microservizi in un'applicazione. Le istanze dei microservizi inviano le loro posizioni al service *registry* all'avvio e vengono rimosse alla chiusura. Un client può quindi interrogare il service *discovery* per ottenere la posizione di un'istanza del microservice con cui interagire.

La **seconda** soluzione consiste nell'utilizzare un **message router**, che instrada le richieste verso **tutte** le istanze effettive di un microservizio, evitando l'interazione diretta.

La **terza** soluzione consiste nell'utilizzare un **message brokers**, che è un sistema **intermediario** che gestisce la comunicazione tra microservizi tramite code di messaggi. Invece di interagire direttamente, i microservizi inviano e ricevono messaggi attraverso il broker, creando un disaccoppiamento tra le parti.



Smell 2 for P2: No API gateway

Lo smell **No API gateway** si verifica quando i client di un applicazione devono necessariamente invocare i suoi microservizi **direttamente**, senza passare da un **API gateway**.

Il client interagisce infatti solo con le istanze specifiche dei microservices di cui ha bisogno. Se uno di questi microservices viene scalato e il client continua a invocare la stessa istanza, si verifica uno spreco di risorse.

La **soluzione** a questo **smell** è **aggiungere un API gateway** all'applicazione. Questo funge da unico punto di ingresso per tutti i client e gestisce le richieste, instradandole o distribuendole tra le istanze dei microservices che devono gestirle.



Principle 3: Isolamento dei failure

I microservices possono fallire per vari motivi, diventando quindi indisponibili per servire altri microservizi. Le applicazioni dovrebbero essere progettate in modo che ciascun microservizio possa **tollerare il fallimento** di qualsiasi invocazione verso i microservizi da cui dipende. In questo modo l'architettura risulta più resiliente rispetto a quella monolitica, poiché i *failures* riguardano solo pochi microservices anziché l'intero monolite.

Smell 1 for P3: Wobbly service interactions

Wobbly service interactions: L'interazione di un microservice mi con un altro microservice mf è considerata **wobbly** (instabile) quando un **fallimento** in *mf* può causare un fallimento anche in *mi*.

Questo accade tipicamente quando il servizio mi utilizza direttamente una o più funzionalità offerte dal servizio mf e non dispone di alcuna soluzione per gestire un eventuale fallimento di mf o la sua mancanza di risposta.

In tal caso, il servizio mi potrebbe fallire a cascata e, nel peggior scenario possibile, il fallimento di mi potrebbe causare il fallimento di altri microservizi, che a loro volta attivano ulteriori fallimenti a cascata. Esistono quattro soluzioni possibili a questo smell.

La **prima soluzione** è l'utilizzo di un **circuit breaker** per gestire le invocazioni da un microservizio a un altro. Nello **stato normale** detto "chiuso", il *circuit breaker* inoltra le invocazioni al microservizio interessato e monitora la loro esecuzione per rilevare e contare i fallimenti. Una volta che la frequenza dei fallimenti raggiunge una determinata soglia, il *circuit breaker* si "apre" interrompendo il circuito. A tutte le chiamate successive al microservizio il *circuit breaker* risponderà immediatamente con un messaggio di errore. I microservizi invocanti possono sfruttare i messaggi di errore per evitare di fallire a loro volta.

La **seconda soluzione** consiste nel disaccoppiare l'interazione tra i microservizi invocanti e invocati utilizzando un **message broker**. In caso di fallimento, l'invocatore invia le sue richieste al broker, mentre il microservizio invocato può elaborarle quando è disponibile. In questo modo l'invocatore non fallisce quando il microservizio invocato fallisce, ma continua a inviare richieste al broker.

La **terza soluzione** consiste nell'uso dei **timeouts**, un meccanismo che permette a un microservizio di interrompere l'attesa di una risposta da un altro microservizio quando quest'ultimo non risponde entro un tempo limite predefinito.

La **quarta soluzione** consiste nell'uso delle **bulkheads**. Le **bulkheads** applicano il principio del contenimento del danno, **suddividendo** logicamente e/o fisicamente i microservizi di un'applicazione in **partizioni**. In questo modo, il fallimento di un microservizio può propagarsi solo ai microservizi all'interno della stessa partizione, evitando che l'intero sistema venga influenzato.



Principle 4: Decentralizzazione

La **decentralizzazione** nei microservizi implica la distribuzione di tutti gli aspetti dell'applicazione, inclusa la logica di business. Ogni microservizio deve gestire in modo autonomo la propria logica di dominio, evitando dipendenze centralizzate, per garantire una maggiore indipendenza, flessibilità e scalabilità dell'intero sistema.

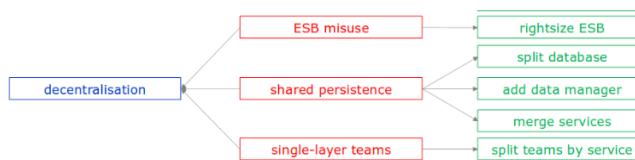
Smell 1 for P4: ESB misuse

Un **Enterprise Service Bus (ESB)** è un software middleware che funge da intermediario per la comunicazione tra servizi in un sistema. Un ESB **centralizza** il traffico dei dati e le interazioni tra componenti software, orchestrando la comunicazione.

L'uso di un **ESB** come intermediario centrale per la comunicazione tra microservizi è uno **smell** perché centralizza la logica di business, riduce l'autonomia dei microservizi.

e introduce un punto unico di fallimento. Questo approccio limita la scalabilità e rende l'architettura più rigida, poiché i microservizi dipendono dall'ESB per comunicare.

Soluzione: Eliminare la dipendenza da un ESB centrale ristrutturando la topologia. Le possibili soluzioni sono le seguenti. Utilizzare **più istanze di middleware** per gestire la comunicazione tra microservizi, evitando un punto unico di fallimento. Implementare una **messaggistica asincrona** basata su code, che si limita a trasportare i messaggi senza eseguire logica elaborativa. Spostare la logica di elaborazione dai middleware centrali ai **microservizi**, che decidono autonomamente quando e come gestire i messaggi ricevuti. Posizionare **componenti infrastrutturali aggiuntivi** accanto a ciascun microservizio tramite sidecar. Un **sidecar** è un componente che viene eseguito accanto a un microservizio per gestire funzionalità generali che non fanno parte della logica di dominio del servizio, come bilanciamento del carico, autenticazione, monitoraggio e gestione del traffico, lasciando i microservizi concentrati sulla logica di dominio.



Smell 2 for P4: Shared persistence

Lo **Shared Persistence smell** si verifica quando **due microservices accedono e gestiscono lo stesso database**.

La **prima soluzione** è quella di suddividere il database condiviso in modo che ciascun microservizio acceda e gestisca solo i dati di cui **ha bisogno**.

La suddivisione del database è raccomandata quando i microservizi che accedono allo stesso database implementano logiche di business separate, lavorando su porzioni distinte di tale database.

La **seconda soluzione** consiste nell'introdurre un microservizio aggiuntivo, che agisca come **data manager**, che diventa l'unico microservizio che interagisce e gestisce il database. I microservizi che accedevano al db ora devono interagire con il **data manager** per richiedere accesso e aggiornamento dei dati.

La **terza soluzione** consiste nella fusione dei microservizi che accedono allo stesso database. L'idea è che, quando più microservizi accedono allo stesso database, ciò potrebbe segnalare che l'applicazione è stata suddivisa in modo eccessivo, con microservizi troppo dettagliati che elaborano gli stessi dati.

Smell 3 for P4: Single-layer teams

L'approccio classico di **suddividere i team per livello tecnologico** (ad esempio, team di interfaccia utente, team di middleware e team di database) è uno **smell**, poiché qualsiasi modifica a un microservizio può trasformarsi in un progetto inter-team.

La **soluzione** consiste nel suddividere i team per microservizi, piuttosto che per layer tecnologico. Ogni team deve contenere membri che coprano tutti i layer tecnologici.

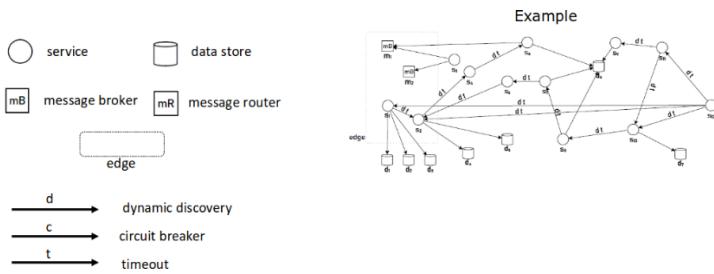
MicroFreshener

MicroFreshener è uno strumento che consente di modellare l'architettura di un'applicazione attraverso una rappresentazione grafica dettagliata. È progettato per supportare gli sviluppatori nell'identificazione e risoluzione degli **architectural smells** dei microservizi.

MicroFreshener automatizza tre principali attività: modellazione delle specifiche architetturali, identificazione degli smells e applicazione di **refactorings architetturali** per correggerli.

La rappresentazione grafica modella: **Servizi** (indicati con cerchi), **data store** che rappresentano database, **componenti di comunicazione**, come *message brokers* per la gestione dei messaggi e *message routers* per il loro smistamento, **connessioni specifiche** tra componenti, tra cui **Dynamic Discovery**, per la scoperta dinamica dei servizi, **Circuit Breaker**, che protegge il sistema interrompendo connessioni a servizi non funzionanti e **Timeout**, che previene blocchi nelle comunicazioni impostando limiti di tempo per le richieste.

Graphical representation



Appicare un *refactoring* in *MicroFreshener* riguarda solo il livello architettonico e **non cambia l'implementazione dell'applicazione**. La realizzazione concreta del *refactoring* è lasciata al responsabile dell'applicazione.

PART 2 of the course

TESTING

Il **software testing** è un processo in cui si esegue il proprio programma utilizzando dati che simulano l'input dell'utente. Si osserva il comportamento del programma per verificare se fa ciò che dovrebbe fare. I test hanno esito positivo se il comportamento è quello atteso; i test falliscono se il comportamento differisce da quello previsto.

Se questi input sono rappresentativi di un set più ampio di input, si può dedurre che il programma si comporterà correttamente per tutti i membri di questo set più ampio.

Se il comportamento del programma non corrisponde a quello atteso, significa che il programma ha dei **bug** che devono essere corretti. Ci sono due cause principali di *bug* nei programmi:

1. **Programming errors:** sono errori accidentali inclusi nel codice del programma.
2. **Understanding errors:** si verifica quando si fraintendono o si ignorano alcuni dettagli di ciò che il programma dovrebbe fare.

Il testing è la principale tecnica utilizzata per convincere gli sviluppatori che un prodotto è pronto per essere rilasciato. Tuttavia, il testing non può mai dimostrare che un programma sia privo di difetti e che non fallirà mai. Potrebbero esserci input o combinazioni di input non utilizzati nei test, e il programma potrebbe fallire se questi input vengono utilizzati in pratica.

È essenziale utilizzare sia le *code reviews* sia il testing del programma. Le *code reviews* consistono revisionare il codice con altri sviluppatori, con l'obiettivo di trovare *bug* che il testing non rileva.

Nella seguente tabella sono specificati i diversi tipi di test

Test type	Testing goals
Functional testing	Testare la funzionalità dell'intero sistema. Gli obiettivi del <i>functional testing</i> sono scoprire quanti più bug possibile nell'implementazione del sistema e fornire prove convincenti che il sistema sia adatto al suo scopo.
User testing	Verificare che il prodotto software sia utile e utilizzabile dagli utenti finali. È necessario dimostrare che le funzionalità del sistema aiutano gli utenti a svolgere ciò che desiderano fare con il software e che comprendono come accedere alle funzionalità e utilizzarle efficacemente.
Performance and load testing	Verificare che il software funzioni rapidamente e sia in grado di gestire il carico previsto dagli utenti. Bisogna dimostrare che i tempi di risposta e di elaborazione del sistema siano accettabili per gli utenti finali e che il sistema possa gestire diversi carichi, adattandosi gradualmente all'aumentare del carico.
Security testing	Verificare che il software mantenga la sua integrità e sia in grado di proteggere le informazioni degli utenti da furti e danni.

In particolare, lo *user testing* può essere organizzato in due fasi:

Alpha testing: Gli utenti lavorano con gli sviluppatori per testare il sistema. L'obiettivo dell'alpha testing è capire se gli utenti vogliono davvero le funzionalità pianificate.

Beta testing: Si distribuiscono versioni preliminari del prodotto agli utenti per ricevere feedback sull'usabilità del prodotto e l'assenza di bug in ambiente utente.

Functional testing

Il *functional testing* consiste nello sviluppare un ampio **set di test** in modo che **tutto** il codice di un programma venga eseguito almeno una volta. Gli obiettivi del functional testing sono scoprire quanti più bug possibile nell'implementazione del sistema e fornire prove convincenti che il sistema sia adatto al suo scopo.

Il *functional testing* è un'**attività a fasi**, in cui inizialmente si testano singole unità di codice. Si integrano le unità di codice con altre per creare unità più grandi e quindi si eseguono ulteriori test. Il processo continua fino a creare un sistema completo pronto per il rilascio. Le fasi che compongono il functional testing sono **unit testing, feature testing, system testing e release testing**.

Il testing dovrebbe iniziare il giorno in cui si inizia a scrivere il codice. Si dovrebbe testare man mano che si implementa il codice, in modo che anche un sistema con poche funzionalità venga testato. Con l'aggiunta di nuove funzionalità, il ciclo sviluppo/test continua fino a ottenere un sistema finito. Questo ciclo di sviluppo/test è semplificato se si sviluppano test automatizzati, così da poter ripetere i test ogni volta che si apportano modifiche al codice.

9.1.1 Unit testing

Lo **unit testing** consiste nel testare una **singola unità di codice** in modo **isolato**. I test devono essere progettati per eseguire tutto il codice in un'unità almeno una volta.

Un'**unità di codice** è una porzione di codice che abbia una responsabilità chiaramente definita, come una funzione, un metodo di classe o un modulo che include un piccolo numero di altre funzioni. È generalmente possibile automatizzare gli unit test.

Principio generale: Se un'unità di codice si comporta come previsto per un set di input con caratteristiche comuni, si comporterà allo stesso modo per un set più ampio i cui membri condividono queste caratteristiche(Correct {1, 5, 17, 45, 99} → Correct(1...99)).

I set di input che verranno trattati allo stesso modo nel codice sono chiamati **equivalence partitions**. Le equivalence partitions scelte per testare le unità devono includere sia input che producono valori corretti, che input con valori volutamente errati. Questi test verificano che il programma rilevi e gestisca correttamente gli input errati. Bisogna testare il programma usando diversi input provenienti da ciascuna equivalence partition.

Se possibile, si dovrebbero identificare i confini delle partizioni e scegliere input su questi confini. Questo perché un errore di programmazione molto comune è l'errore **off-by-1**, in cui l'elemento iniziale o finale di un ciclo non viene gestito correttamente. È anche possibile identificare partizioni di equivalenza dell'output e creare input di test che generano risultati in queste partizioni.

Gli input devono seguire le seguenti linee guida, in OR. Se la tua partizione ha limiti superiori e inferiori (es. lunghezza delle stringhe, numeri, ecc.), scegli input ai margini dell'intervallo. 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. Scegli input di test che facciano traboccare tutti i buffer di input. Ripeti lo stesso input più volte. Se il tuo programma utilizza puntatori o stringhe, testa sempre con puntatori e stringhe nulli. Se usi sequenze, testa con una sequenza vuota. Per input numerici, testa sempre con zero. Quando lavori con liste e trasformazioni di liste, conta il numero di elementi in ogni lista e verifica che questi siano coerenti dopo ogni trasformazione. Se il tuo programma gestisce sequenze, testa sempre con sequenze che contengono un singolo valore.

9.1.2 Feature testing

Le unità di codice vengono integrate per creare una feature. Una **feature** di un prodotto implementa una funzionalità utile per l'utente. Le **feature** devono essere testate per dimostrare che la funzionalità è implementata come previsto e soddisfa i reali bisogni degli utenti. Il processo di feature testing dovrebbe includere due tipi di test:

Interaction tests: Testano le interazioni tra le unità che implementano la *feature*.

Usefulness tests: Verificano che la *feature* implementi ciò che gli utenti vogliono probabilmente ottenere.

Un buon modo per organizzare il *feature testing* è sviluppare una serie di test per la che eseguono verifiche basandosi su un insieme di user stories che compongono la feature.

Table 9.5 User stories for the sign-in with Google feature

Story title	User story
User registration	As a user, I want to be able to log in without creating a new account so that I don't have to remember another login ID and password.
Information sharing	As a user, I want to know what information you will share with other companies. I want to be able to cancel my registration if I don't want to share this information.
Email choice	As a user, I want to be able to choose the types of email that I'll get from you when I register for an account.

Table 9.6 Feature tests for sign-in with Google

Test	Description
Initial login screen	Test that the screen displaying a request for Google account credentials is correctly displayed when a user clicks on the "Sign-in with Google" link. Test that the login is completed if the user is already logged in to Google.
Incorrect credentials	Test that the error message and retry screen are displayed if the user inputs incorrect Google credentials.
Shared information	Test that the information shared with Google is displayed, along with a cancel or confirm option. Test that the registration is canceled if the cancel option is chosen.
Email opt-in	Test that the user is offered a menu of options for email information and can choose multiple items to opt in to emails. Test that the user is not registered for any emails if no options are selected.

Per sviluppare feature test, è necessario comprendere la *feature* dal punto di vista dei rappresentanti degli utenti. Il *feature testing* è una parte integrante del behaviour driven

development (*BDD*), in cui il comportamento di un prodotto è specificato usando un linguaggio specifico per il dominio, e i feature test vengono automaticamente derivati da questa specifica.

9.1.3 System testing

Le feature vengono integrate per creare una versione funzionante (anche se incompleta) di un sistema.

Il **system testing** prevede **di testare il sistema nel suo complesso** piuttosto che le singole funzionalità. Il *system testing* inizia nelle prime fasi del processo di sviluppo del prodotto, appena si ha una versione funzionante, anche se incompleta, del sistema. Il *system testing* deve: Verificare se ci sono interazioni inattese tra le feature del sistema. Verificare se le feature funzionano insieme in modo efficace per supportare ciò che gli utenti desiderano fare. Verificare che il sistema funzioni come previsto nei diversi ambienti in cui verrà utilizzato. Verificare la reattività, la capacità di throughput, la sicurezza e altri attributi di qualità del sistema.

Per testare un sistema ogni volta che ne viene rilasciata una nuova versione si può utilizzare uno scenario, tramite cui puoi identificare una serie di percorsi end-to-end che gli utenti potrebbero seguire quando utilizzano il sistema. Un percorso end-to-end è una sequenza di azioni dall'inizio dell'utilizzo del sistema per un compito fino al completamento del compito. Per ciascun percorso, è necessario verificare che le risposte del sistema siano corrette e che vengano fornite informazioni appropriate all'utente.

Dovresti cercare di automatizzare il più possibile i test, eseguendoli ogni volta che viene creata una nuova versione del sistema. Tuttavia, poiché i percorsi end-to-end coinvolgono l'interazione con l'utente, potrebbe non essere pratico automatizzare tutti i test di sistema.

9.1.4 Release testing

Il sistema viene preparato per il rilascio ai clienti. Il sw può essere rilasciato come servizio cloud o come download da installare sul pc o mobile del cliente.

Il **release testing** è un tipo di testing di un sistema destinato al **rilascio** ai clienti, che **testa il sistema nel suo ambiente operativo reale** piuttosto che in un ambiente di test, **con i dati reali** degli utenti.

L'obiettivo del *release testing* è decidere se il sistema è **sufficientemente buono** per il **rilascio**, non rilevare *bug* nel sistema.

La preparazione di un sistema per il rilascio implica l'impacchettamento di quel sistema per la distribuzione (es. in un container se è un servizio cloud) e l'installazione di software e librerie utilizzati dal prodotto. Bisogna definire i parametri di configurazione, come il nome di una directory radice, il limite di dimensione del database per utente e così via. Tuttavia, si potrebbero commettere errori in questo processo di installazione. Pertanto, è necessario eseguire nuovamente i test di sistema per verificare di non aver introdotto nuovi *bug* che influenzano la funzionalità, le prestazioni o l'usabilità del sistema.

Se il prodotto è distribuito nel cloud, si può utilizzare un processo di rilascio continuo ogni volta che viene apportata una modifica. Ciò è praticabile solo se si effettuano modifiche frequenti e di piccola entità e si utilizzano test automatizzati per verificare che queste modifiche non abbiano introdotto nuovi *bug* nel programma.

9.2 Test automation

Il **test automation** si basa sull'idea che i test dovrebbero essere **eseguibili**, così da essere automatizzati. Un test eseguibile include i dati di input per l'unità da testare, il risultato atteso e un controllo che l'unità restituisca il risultato atteso. Si esegue il test, e il test passa se l'unità restituisce il risultato atteso.

Oggi sono disponibili *testing frameworks* per tutti i linguaggi di programmazione più usati, come JUnit per Java. Una suite di centinaia di *unit tests*, sviluppata utilizzando un framework, può essere eseguita in pochi secondi. Un rapporto di test mostra quali test sono stati superati e quali falliti.

Per creare un test automatizzato, si definisce la propria classe di test come sottoclasse di una classe base di test fornita dal framework. I *testing frameworks* si occupano di eseguire tutti i test definiti nelle sottoclassi e riportare i risultati dei test.

```
def test_zero_principal (self):  
  
    #Arrange - set up the test parameters  
    p = 0  
    r = 3  
    n = 31  
    result_should_be = 0  
  
    #Action - Call the method to be tested  
    interest = interest_calculator (p, r, n)  
  
    #Assert - test what should be true  
    self.assertEqual (result_should_be, interest)
```

È una buona pratica strutturare i test automatizzati in tre parti:

Arrange: si configura il sistema per eseguire il test. Questo implica definire i parametri del test e, se necessario, oggetti *mock* che emulano la funzionalità di codice non ancora sviluppato.

Action: si chiama l'unità da testare con i parametri del test.

Assert: si fa un'asserzione su ciò che dovrebbe essere vero se l'unità in fase di test ha funzionato correttamente, come la funzione di assert.

Idealmente, si dovrebbe avere solo un'asserzione in ogni test. Se si hanno più asservzioni, potrebbe non essere chiaro quale di queste sia fallita. Tuttavia, questa non è una regola assoluta.

Il *testing framework* fornisce un **test runner** che esegue i test e riporta i risultati. Per utilizzare un *test runner*, bisogna configurare i test in file che iniziano con un nome riservato, tipo ‘test_’. Il *test runner* trova tutti i file di test ed esegue i test contenuti.

Quando si scrivono test automatizzati, è importante mantenerli il più semplici possibile, perché il codice dei test potrebbe includere dei **bug**.

Poiché lo scopo dei test automatizzati è evitare il controllo manuale dei risultati dei test, non è realistico scoprire errori nei test eseguendoli. Pertanto, bisogna adottare due approcci per ridurre la probabilità di errori nei test:

Rendere i test il più **semplici** possibile. Più complesso è il test, maggiore è la probabilità che contenga *bug*. Rivedere tutti i **test insieme al codice** che testano. Qualcuno diverso dal programmatore che ha scritto il test dovrebbe controllare i test per verificarne la correttezza.

Il **regression testing** è il processo di riesecuzione dei test precedenti quando si apporta una modifica a un sistema. Dopo ogni modifica al codice, è sempre consigliabile rieseguire tutti i test per assicurarsi che tutto continui a funzionare come previsto.

Il 70% dei test automatizzati dovrebbe essere costituito da *unit tests*, il 20% da *feature tests* e il 10% da *system tests* (*In ordine di costo per test*).

I test basati sulla Graphical User Interface sono costosi da automatizzare, quindi è meglio progettare il prodotto in modo che le sue funzionalità possano essere accessibili direttamente tramite un'API, non solo dall'interfaccia utente, in modo da automatizzare i *feature tests*.

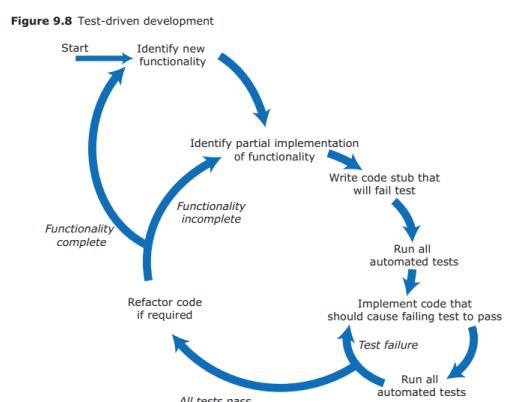
Per i *feature tests* a volte è necessario proprio utilizzare un *framework* di testing specializzato per le funzionalità.

Nel *system testing* si cercano interazioni tra le funzionalità che causano problemi, sequenze di azioni dell'utente che portano a crash del sistema. Ripetere sequenze di azioni manualmente è soggetto a errori. Sono stati quindi sviluppati strumenti che registrano una serie di azioni e le riproducono automaticamente quando il sistema viene ritestato. Salvano la sessione di interazione dell'utente e possono riprodurla, inviando comandi all'applicazione e replicandoli nell'interfaccia browser dell'utente.

9.3 Test-driven development

Il **Test-driven development (TDD)** è un approccio basato sull'idea di scrivere uno o più **test eseguibili** per il codice che si sta sviluppando **prima di scrivere il codice stesso**.

Il TDD è stato introdotto dai primi utenti del metodo agile *Extreme Programming*, ma può essere utilizzato con qualsiasi approccio di sviluppo incrementale.



Si assume di aver identificato un incremento di funzionalità da implementare. Il processo è il seguente. Suddividi l'implementazione della funzionalità richiesta in mini-unità più piccole. Scegli una di queste mini-unità per l'implementazione. Scrivi uno o più test automatizzati per la mini-unità che hai scelto di implementare. La mini-unità dovrebbe superare questi test in futuro se sarà implementata correttamente. Scrivi un codice incompleto che verrà chiamato per implementare la mini-unità. Esegui tutti i test automatizzati esistenti. Tutti i test precedenti dovrebbero superare. Il test per il codice incompleto dovrebbe fallire. Scrivi il codice per implementare la mini-unità, che dovrebbe farla funzionare correttamente. Se qualche test fallisce, il codice è errato. Continua a lavorarci finché tutti i test non passano. Se tutti i test passano, puoi procedere all'implementazione della mini-unità successiva. Se vedi modi per migliorare il codice, fallo prima della fase successiva di implementazione.

Il *test-driven development* si basa sul *test automation*. Ogni volta che si aggiunge una funzionalità, si sviluppa un nuovo test. Tutti i test nella suite devono essere superati prima di procedere con lo sviluppo dell'incremento successivo.

I benefici del *test-driven development* sono i seguenti. I test sono chiaramente collegati alle sezioni del codice del programma, quindi si può essere sicuri che i test coprano tutto il codice sviluppato. I test fungono da specifica scritta per il codice del programma, rendendo capibile cosa fa il programma leggendo i test. Il debugging è

semplificato perché, quando si osserva un malfunzionamento del programma, si può immediatamente collegarlo all'ultimo incremento di codice aggiunto al sistema. Si sostiene che il TDD porti a un codice più semplice, poiché i programmatore scrivono solo il codice necessario per superare i test, senza implementare funzionalità complesse non necessarie.

Il TDD funziona meglio per lo sviluppo di singole unità del programma; è più difficile da applicare al *system testing*.

Svantaggi del TDD

Gli sviluppatori sono riluttanti a prendere decisioni di refactoring sapendo che avrebbero causato il fallimento di molti test. Per questo motivo tendono a evitare cambiamenti radicali nel programma.

Un principio fondamentale del TDD è che il design dovrebbe essere guidato dai test scritti. Inconsapevolmente, ridefiniscono il problema che stanno cercando di risolvere per rendere più facile scrivere i test. Questo significa che a volte non implementano controlli importanti perché è difficile scrivere i test in anticipo.

A volte, quando si programma, è meglio fare un passo indietro e guardare il programma nel suo insieme piuttosto che concentrarsi sui dettagli di implementazione. Il TDD incoraggia a focalizzarsi sui dettagli che potrebbero far passare o fallire i test piuttosto che sulla struttura generale del programma. Molti problemi riguardano la gestione di dati disordinati e incompleti. È praticamente impossibile anticipare tutti i problemi di dati che potrebbero emergere e scrivere test per questi in anticipo. Si potrebbe sostenere che si dovrebbe semplicemente rifiutare i dati non validi, ma a volte questo è impraticabile.

9.4 Security testing

Gli obiettivi del **security testing** sono individuare vulnerabilità che un attaccante potrebbe sfruttare e fornire prove che il sistema sia sufficientemente sicuro. I test devono dimostrare che il sistema può resistere ad attacchi, a tentativi di iniettare malware e a quelli che cercano di corrompere o rubare i dati e l'identità degli utenti.

Scoprire vulnerabilità è molto più difficile che trovare *bug*. Quando si testa per le vulnerabilità, si cerca qualcosa che il software **non** dovrebbe fare, quindi esistono infiniti test possibili. Le vulnerabilità sono spesso oscure e si nascondono in codice raramente utilizzato, quindi potrebbero non essere rivelate dai normali test funzionali. I prodotti software dipendono da un *software stack* che include SO, librerie, database, browser, ecc. Questi componenti possono contenere vulnerabilità che influenzano il software.

Un *security testing* completo richiede una conoscenza specialistica delle vulnerabilità software. Molte aziende esterne offrono servizi di **penetration testing**, in cui **simulano attacchi** al software per scoprire modi per violarne la sicurezza. Tuttavia, questi test sono spesso costosi e non accessibili per tutti.

Un modo pratico per organizzare il *security testing* è adottare un approccio basato sui rischi, **identificando i rischi comuni** e sviluppando test per dimostrare che il sistema è **protetto** da questi rischi. Si possono anche usare strumenti automatizzati per scansionare il sistema alla ricerca di vulnerabilità note, come porte HTTP aperte inutilizzate. La seguente tabella mostra esempi di rischi che potrebbero essere testati.

Un attaccante non autorizzato accede a un sistema usando credenziali autorizzate.
Un individuo autorizzato accede a risorse a lui vietate. Il sistema di autenticazione non rileva un attaccante non autorizzato. Un attaccante accede al database usando

un attacco di SQL poisoning. Gestione impropria delle sessioni HTTP. I cookie di sessione HTTP vengono rivelati a un attaccante. I dati confidenziali non sono crittografati. Le chiavi di crittografia vengono trapelate a potenziali attaccanti.

Una volta identificati i rischi, si progettano test per verificare se il sistema è vulnerabile. Alcuni di questi test possono essere automatizzati, ma altri richiedono inevitabilmente controlli manuali del comportamento e dei file del sistema.

Per testare la sicurezza di un sistema, bisogna pensare come un attaccante piuttosto che come un normale utente finale.

Ciò significa provare deliberatamente a fare cose sbagliate, poiché le vulnerabilità del sistema spesso si nascondono in codice usato raramente che gestisce situazioni eccezionali. Si possono ripetere azioni più volte, poiché a volte ciò porta a comportamenti diversi.

Code reviews

Il testing è la tecnica più utilizzata per trovare *bug* nei programmi. Tuttavia, presenta tre problemi fondamentali: Si può testare il codice solo in base alla propria **comprensione** di cosa dovrebbe fare. Se si fraintende lo scopo del codice, questo si rifletterà sia nel codice sia nei test. I test a volte sono **difficili** da progettare, con il rischio che **non coprano** tutto il codice scritto. Il TDD evita questo problema, poiché ogni test è associato a una porzione di codice. Tuttavia, il TDD cambia solo il problema: al posto dell'incompletezza del test, si può avere incompletezza del codice perché non si considerano eccezioni rare. Il testing non fornisce **informazioni** su altri attributi del programma, come la **leggibilità**, la struttura o l'evolvibilità.

Per ridurre gli effetti di questi problemi, molte aziende richiedono che tutto il codice venga sottoposto a **code review** prima di essere rilasciato. Le *code review* completano il testing, risultando efficaci nell'individuare errori che potrebbero emergere solo quando vengono eseguite sequenze di codice inusuali. Le attività coinvolte nel processo di *code review* sono le seguenti.

Il programmatore contatta un revisore e organizza una data per la revisione. Il programmatore raccoglie il codice e i test per la revisione e li annota con informazioni per il revisore sullo scopo del codice e dei test. Invia il codice e i test al revisore. Il revisore controlla sistematicamente il codice e i test in base alla propria comprensione di cosa dovrebbero fare. Il revisore annota il codice e i test con un report sui problemi da discutere durante la riunione di revisione. Il revisore e il programmatore discutono i problemi e concordano sulle azioni da intraprendere per risolverli. Il programmatore documenta l'esito della revisione come una lista di cose da fare e la condivide con il revisore. Infine modifica il codice e i test per affrontare i problemi emersi durante la revisione.

Viene utilizzato un solo revisore, che può essere parte dello stesso team di sviluppo o lavorare in un'area correlata. Oltre a controllare il codice sottoposto a revisione, il revisore deve esaminare i test automatizzati sviluppati, verificando che la suite di test sia completa e coerente con la comprensione dello scopo del codice.

Oltre a cercare errori e fraintendimenti nel codice, il revisore può commentare la leggibilità e la comprensibilità del codice. Se l'azienda ha uno standard di codifica, la revisione dovrebbe verificare la conformità a questo standard.

La revisione dovrebbe durare circa un'ora, in modo da poter rivedere tra 200 e 400 righe di codice in una singola sessione. Poiché le persone commettono errori simili, è efficace preparare una checklist per i revisori da utilizzare durante la revisione del

codice. Le checklist possono contenere una combinazione di elementi generali e specifici, in base agli errori caratteristici del linguaggio di programmazione utilizzato.

Sono disponibili diversi strumenti di *code review* per supportare il processo. Utilizzando questi strumenti, sia il programmatore sia il revisore possono annotare il codice in fase di revisione e documentare il processo creando liste di cose da fare. Questi strumenti di revisione possono essere configurati in modo che, ogni volta che un programmatore invia codice a un repository come GitHub, venga automaticamente predisposta una *code review*. Gli strumenti di revisione possono anche integrarsi con un sistema di tracciamento degli *issue*, sistemi di messaggistica come Slack e sistemi di comunicazione vocale come Skype.

DevOps and Code Management

Tradizionalmente, **team separati** erano responsabili dello **sviluppo software**, del **rilascio** e del **supporto** software.

Il team di sviluppo consegnava una versione “finale” del software a un team di rilascio. Quel team poi creava una versione di rilascio, la testava e preparava la documentazione di rilascio prima di distribuirla ai clienti. Un terzo team forniva supporto ai clienti. Il team di sviluppo originale era talvolta responsabile dell’implementazione delle modifiche al software, oppure il software poteva essere mantenuto da un team di manutenzione separato.

In questi processi, i ritardi nella comunicazione tra i gruppi erano inevitabili. I vari team utilizzavano strumenti diversi, avevano competenze diverse e spesso non comprendevano i problemi dell’altro. Anche quando veniva identificato un bug urgente, potevano essere necessari diversi giorni per preparare e distribuire una nuova versione ai clienti.

DevOps (development + operations) è un approccio in cui un **unico team** è responsabile delle attività di **sviluppo, distribuzione e supporto** del sw.

DevOps mira a creare un unico team responsabile sia dello sviluppo che delle operazioni. Gli sviluppatori si assumono anche l’**responsabilità** di installare e mantenere il proprio software. Tre fattori hanno portato alla diffusione di DevOps: L’ingegneria del sw agile ha ridotto il tempo di sviluppo del software, ma il processo di rilascio tradizionale ha introdotto un collo di bottiglia tra sviluppo e distribuzione. Amazon ha riprogettato il proprio sw attorno ai microservizi e ha introdotto un approccio in cui un team unico si occupava sia dello sviluppo sia del supporto. È diventato possibile rilasciare il sw come servizio, eseguibile su cloud pubblici o privati, senza l’utilizzo di supporti fisici o tramite download.

I **tre principi fondamentali** che costituiscono la base per un DevOps efficace sono i seguenti. Tutti i membri del team hanno la **responsabilità congiunta** di sviluppare, distribuire e supportare il sw. Tutte le attività coinvolte nei test, nella distribuzione e nel supporto dovrebbero **essere automatizzate** se possibile. Il DevOps dovrebbe essere guidato da un programma di misurazione in cui si raccolgono **dati sul sistema** e sul suo funzionamento. I dati raccolti vengono poi utilizzati per prendere decisioni sui **cambiamenti ai processi** e agli strumenti **DevOps**.

I benefici universali del DevOps sono i seguenti. Il software può essere **distribuito** in produzione più **rapidamente** perché i ritardi di comunicazione tra le persone coinvolte nel processo sono notevolmente ridotti. L’**incremento** delle funzionalità in ogni rilascio è **piccolo**, quindi c’è meno probabilità di interazioni tra funzionalità e altri cambiamenti che causano guasti o interruzioni del sistema. I team DevOps **lavorano insieme** per far funzionare di nuovo il software il più velocemente possibile. Non è necessario scoprire **quale team** fosse responsabile del problema e attendere che lo risolva.

Creare un team DevOps significa riunire una serie di competenze diverse, che possono includere ingegneria del sw, design UX, ingegneria della sicurezza, ingegneria delle infrastrutture e interazione con i clienti.

Un team DevOps di successo ha una cultura di rispetto reciproco e condivisione. Tutti i membri del team dovrebbero partecipare agli Scrums e ad altre riunioni di squadra. I membri del team dovrebbero essere incoraggiati a condividere la propria esperienza con gli altri e a imparare nuove competenze. Gli sviluppatori dovrebbero supportare i servizi software che hanno sviluppato. Se un servizio fallisce durante un weekend, quello sviluppatore è responsabile di farlo tornare operativo. Se quella persona non è disponibile, tuttavia, gli altri membri del team dovrebbero intervenire piuttosto che aspettare il suo ritorno. La priorità del team dovrebbe essere riparare i guasti il più rapidamente possibile, piuttosto che attribuire colpe a un membro del team.

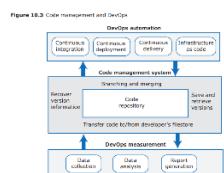
10.1 Gestione del codice

DevOps dipende dal sistema di gestione del codice sorgente utilizzato dall'intero team. Durante lo sviluppo di un prodotto software, il team di sviluppo probabilmente creerà decine di migliaia di righe di codice e test automatizzati. Senza supporto automatizzato, è impossibile per gli sviluppatori tenere traccia delle modifiche apportate al software.

La gestione del codice è un insieme di pratiche supportate da software utilizzate per gestire una base di codice in evoluzione. La gestione del codice è necessaria per garantire che le modifiche apportate da sviluppatori diversi non interferiscano tra loro e per creare diverse versioni del prodotto.

Due membri di un team che lavorano sullo stesso modulo sw senza coordinarsi, possono portare a una situazione in cui uno pulla una versione precedente alla correzione di un bug da parte dell'altro membro e la sovrascrive. Il bug risulta globalmente come risolto e testato e viene rilasciato nel programma.

Gli strumenti di automazione e misurazione DevOps interagiscono tutti con il sistema di gestione del codice (Figura 10.3).



10.1.1 Fondamenti della gestione del codice sorgente

Tutti i file di codice sorgente e le versioni sono archiviati nel repository, insieme a tutte le librerie utilizzate. Il repository include un database con informazioni sui file archiviati, come versioni, autori delle modifiche, descrizioni delle modifiche e tempi delle stesse.

Il sistema di gestione del codice fornisce un insieme di funzionalità a supporto di quattro aree principali:

Trasferimento del codice: Gli sviluppatori prelevano il codice nel loro spazio di lavoro personale per modificarlo e poi lo restituiscono al sistema di gestione del codice condiviso.

Archiviazione e recupero delle versioni: I file possono essere salvati in diverse versioni. Le versioni gestite di un file di codice sono identificate in modo univoco quando vengono inviate al sistema e possono essere recuperate utilizzando il loro identificatore e altri attributi del file.

Merging e branching: Possono essere create branch di sviluppo parallele per il lavoro simultaneo. Le modifiche apportate dai diversi sviluppatori in branch diverse possono essere mergeate.

Denotiamo alcuni **vantaggi**. I motivi per cui sono state apportate modifiche a un file di codice vengono registrati e mantenuti. Diversi sviluppatori possono lavorare sullo stesso file di codice contemporaneamente. Quando questo viene inviato al sistema di gestione del codice, viene creata una nuova versione per garantire che i file non vengano mai sovrascritti da modifiche successive. Tutti i file associati a un progetto possono essere estratti contemporaneamente. Non è necessario estrarre i file uno alla volta. Il sistema di gestione del codice include meccanismi di archiviazione efficienti per evitare di conservare più copie di file che differiscono solo per piccoli dettagli.

Nei **sistemi di gestione del codice distribuiti**, il repository è **replicato** su ciascun computer degli sviluppatori. I sistemi distribuiti presentano diversi vantaggi rispetto ai sistemi centralizzati:

Resilienza: tutti i partecipanti a un progetto hanno una propria copia del repository. Si può lavorare offline se non si ha una connessione. Se il repository condiviso viene danneggiato il lavoro può continuare e i cloni possono essere utilizzati per ripristinare il repository condiviso.

Velocità: il commit delle modifiche al repository è un'operazione veloce e locale, che non richiede il trasferimento di dati tramite la rete.

Flessibilità: sperimentare localmente è molto più semplice. Gli sviluppatori possono provare in sicurezza approcci diversi senza esporre i propri esperimenti agli altri membri del progetto.

Git è organizzato attorno al concetto di repository distribuito e cloni privati di quel repository sui computer di ciascun sviluppatore.

10.1.2 Utilizzo di Git

Git init: Inizializza un nuovo repository Git in una directory esistente o vuota.

Git clone: Crea una copia locale di un repository remoto

git add: Prepara i file per il prossimo commit, aggiungendoli all'area di staging.

git commit: Registra le modifiche presenti nell'area di staging nel repository.

Un **branch** è una versione indipendente, creata quando uno sviluppatore desidera modificare un file. Il repository assicura che i file dei branch modificati non possano sovrascrivere i file del repository senza un'operazione di merge. Le modifiche apportate dagli sviluppatori nei propri branch possono essere unite per creare un nuovo branch condiviso.

Git confronta le versioni dei file riga per riga. Non c'è conflitto se gli sviluppatori hanno modificato righe diverse del file. Tuttavia, se hanno apportato modifiche alle stesse righe, viene segnalato un conflitto di merge.

git checkout -b nomebranch: Consente di spostarsi tra i rami di un repository.
git merge nomebranch

git merge: Il comando git merge unisce due rami in uno solo, integrando le modifiche del ramo sorgente (branch di origine) nel ramo attuale.

git push: esamina il tuo repository e quello esterno, calcola quali file sono stati modificati e invia le modifiche al repository esterno.

Git pull: Scarica e unisce gli aggiornamenti dal repository remoto al branch locale. Combina due comandi: git fetch (scarica le modifiche) e git merge (integra le modifiche).

Nel modello open-source, molte persone diverse possono lavorare indipendentemente sul codice senza sapere cosa stanno facendo gli altri. La versione master del software open-source è gestita da un individuo che decide quali modifiche includere.

GitHub utilizza un meccanismo molto generale chiamato *Webhooks* per attivare azioni in risposta a un aggiornamento del repository di progetto. I Webhooks inviano dati, utilizzando una richiesta HTTP POST, a un URL quando si verifica un'azione. Pertanto, puoi configurare GitHub per inviare messaggi agli sviluppatori sulle modifiche e attivare la build e il test del sistema quando viene aggiunto nuovo codice.

10.2 Automazione DevOps

“Automatizzare tutto ciò che può essere automatizzato” è un principio fondamentale di DevOps. L’automatizzazione riduce i tempi e rende più affidabili i processi di integration, delivery e deployment.

Continuous integration: Ogni volta che uno sviluppatore effettua un commit di una modifica al branch master del progetto, viene creata e testata una versione eseguibile del sistema.

Continuous delivery: Viene creata una simulazione dell’ambiente operativo del prodotto e la versione eseguibile del software viene testata.

Continuous deployment: Una nuova release del sistema viene resa disponibile agli utenti ogni volta che viene apportata una modifica al branch master del software.

Infrastructure as code: Utilizzare un modello dell’infrastruttura scritto in un linguaggio elaborabile da una macchina per installare e aggiornare il sw sui server in modo automatizzato.

Le informazioni di automazione sono codificate in script che possono essere verificati, revisionati, versionati e archiviati nel repository del progetto. La distribuzione non dipende da un amministratore di sistema che conosce le configurazioni del server.

10.2.1 Continuos Integration

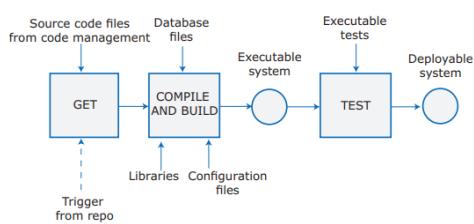
L'integrazione di sistema è il processo di raccolta di tutti gli elementi necessari per un sistema funzionante, spostandoli nelle directory corrette e **assemblandoli** per creare un sistema funzionante. Le **fasi** che include sono le seguenti. Installazione e configurazione del db con lo schema appropriato; caricamento dei dati di test nel db; compilazione dei file che compongono il progetto; collegamento del codice compilato con le librerie e altri componenti utilizzati; verifica che i servizi esterni utilizzati siano operativi; spostamento dei file di configurazione nelle posizioni corrette; esecuzione di una serie di test di sistema per verificare che l'integrazione sia riuscita.

Se un sistema viene integrato raramente, molti dei suoi componenti subiscono modifiche, talvolta significative, tra un'integrazione e l'altra, portando a problemi.

Ogni volta che uno sviluppatore effettua un commit di una modifica al branch master del progetto, viene creata e testata una versione eseguibile del sistema.

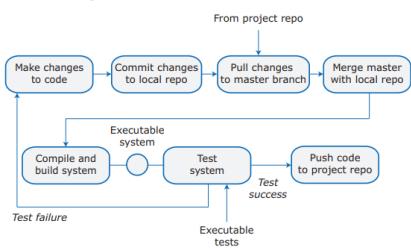
La Continuos Integration significa che una **versione integrata** del sistema viene creata e testata ogni volta che viene effettuata una **modifica** al repository di codice condiviso del sistema. Al termine dell'operazione di push, il repository invia un messaggio a un server di integrazione per costruire una nuova versione del prodotto.

Figure 10.9 Continuous integration



Rompere la build significa inviare codice al repository di progetto che, una volta integrato, causa il fallimento di alcuni test del sistema, bloccando gli altri sviluppatori. Per evitare di rompere la build, è consigliato adottare un approccio di "integrare due volte" all'integrazione del sistema: **integrare e testare sul proprio computer** prima di inviare il codice al repository del progetto per attivare il server di integrazione.

Figure 10.10 Local integration



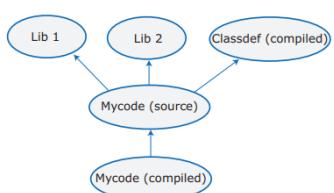
Vantaggi. L'integrazione continua permette di trovare i bug nel sistema più velocemente, perché se fai una piccola modifica e poi falliscono alcuni test di sistema, il problema è quasi certamente nel nuovo codice che hai committato. Se integri continuamente, un sistema funzionante è sempre disponibile per l'intero team, che può essere utilizzato per testare idee e dimostrare le funzionalità del sistema alla direzione e ai clienti.

L'integrazione continua è efficace **solo se** il processo di integrazione è **veloce** e gli sviluppatori non devono aspettare i risultati dei loro test sul sistema integrato. Tuttavia, alcune attività nel processo di build, come il popolamento di un database o la compilazione di centinaia di file di sistema, sono intrinsecamente **lente**. È quindi essenziale avere un processo di **build automatizzato** che minimizzi il tempo impiegato in queste attività.

La costruzione automatizzata veloce è possibile perché le modifiche fatte tra un'integrazione e l'altra sono di solito **piccole**, quindi solo pochi file di codice sorgente sono stati modificati. Gli strumenti di integrazione del codice utilizzano un processo di build incrementale in modo da ripetere solo le azioni necessarie, come la compilazione, se i file dipendenti sono stati modificati.

La **prima volta** che si integra un sistema, il sistema di build incrementale compila tutti i file di codice sorgente e i file di test eseguibili. Crea i loro equivalenti in codice oggetto ed esegue i test eseguibili. Successivamente, però, i file di codice oggetto vengono creati solo per i nuovi test e per quelli modificati e per i file di codice sorgente modificati.

File dependencies



I file di codice sorgente raramente sono indipendenti, ma si basano su altre informazioni, come librerie.

Un sistema di build usa il **timestamp** di modifica del file per decidere se un file di codice sorgente è stato modificato dopo la creazione del file di codice oggetto associato. Il sistema di build ricompila il codice sorgente se e solo se la **data di modifica** del codice sorgente (o di una sua dipendenza) è **successiva** a quella del relativo codice oggetto.

Registrare manualmente le dipendenze dei file è noioso. Esistono tool per creare automaticamente un modello di dipendenze da utilizzare per la costruzione del sistema, come Maven.

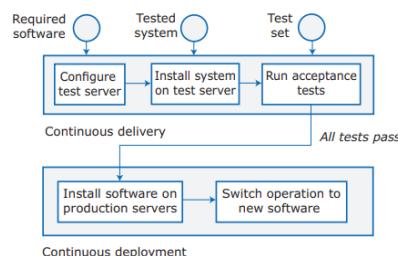
10.2.2 Continuos Delivery and Deployment

La continuos integration costruisce il sistema ed esegue i test sul server di sviluppo del progetto. Tuttavia, il server di sviluppo potrebbe avere una diversa organizzazione dell'ambiente (file system, librerie, applicazioni presenti) rispetto all'ambiente operativo reale, dove potrebbero emergere bug che non si sono presentati nell'ambiente di test.

Continuous delivery: Viene creata una *simulazione dell'ambiente operativo* del prodotto e la versione eseguibile del software viene testata, in modo che sia pronta per la consegna dei clienti.

Il modo più semplice per creare una replica di un ambiente reale è eseguire il software in un container. Per creare un ambiente di test, basta creare un altro container utilizzando la stessa immagine.

Figure 10.13 Continuous delivery and deployment



Dopo i test di integrazione iniziali, viene creato un ambiente di test a fasi. Questo è una replica dell'effettivo ambiente di produzione. Vengono quindi eseguiti i test di accettazione del sistema, che includono test di funzionalità, carico e prestazioni, per verificare che il software funzioni come previsto. Se tutti questi test vengono superati, il software modificato viene installato sui server di produzione.

Per **deployare** il sistema, trasferisci il software e i dati aggiornati ai server di produzione. Poi sospendi momentaneamente tutte le nuove richieste di servizio e lasci che la versione precedente elabori le transazioni in sospeso. Una volta completate, passi alla nuova versione del sistema e riavvii l'elaborazione.

Continuous deployment: Una nuova release del sistema viene resa disponibile agli utenti ogni volta che viene apportata una modifica al branch master del software.

Vantaggi. L'adozione del continuos deployment comporta l'investimento in una pipeline automatizzata, riducendo i costi a lungo termine grazie alla maggiore efficienza rispetto alla distribuzione manuale, che è lenta e soggetta a errori. Questo approccio semplifica l'individuazione e la risoluzione di problemi, limitandone l'impatto a una piccola parte del sistema e rendendo più rapido l'intervento rispetto a rilasci complessi e cumulativi. Inoltre, consente di distribuire nuove funzionalità appena pronte,

raccogliendo feedback diretto dai clienti per miglioramenti mirati. Infine, supporta l'A/B testing.

A/B testing: Puoi distribuire una nuova versione del software su alcuni server e lasciare la versione precedente su altri. Usando il load balancer, puoi indirizzare alcuni clienti verso la nuova versione mentre altri continuano a usare quella vecchia, misurando e valutando l'uso delle nuove funzionalità per vedere se soddisfano le aspettative.

Motivi aziendali per cui un'azienda potrebbe non voler distribuire ogni modifica del software ai clienti. Potresti avere funzionalità incomplete pronte per essere distribuite, ma vuoi evitare di dare informazioni ai concorrenti fino a quando la loro implementazione non è completa. I clienti potrebbero essere infastiditi da un software che cambia continuamente, specialmente se ciò influisce sull'interfaccia utente. Potresti voler sincronizzare i rilasci del software con cicli aziendali noti.

Gli strumenti di CI come Jenkins e Travis possono essere utilizzati anche per supportare la continuos delivery e deployment.

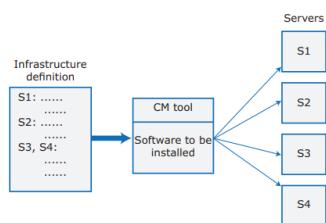
10.2.3 Infrastructure as a code

Tenere traccia del software installato su ogni macchina server è complicato. Quindi, mantenere manualmente un'infrastruttura con centinaia di server è costoso.

Infrastructure as code: Utilizzare un modello dell'infrastruttura scritto in un linguaggio elaborabile da una macchina per installare e aggiornare il sw sui server in modo automatizzato.

Gli strumenti di **configuration management**, come **Puppet** e **Chef**, possono installare automaticamente sw sui server in base alla definizione dell'infrastruttura. Quando devono essere apportate modifiche, il modello di infrastruttura viene aggiornato e lo strumento CM apporta le modifiche a tutti i server.

Figure 10.14 Infrastructure as code



L'uso di un CM tool assicura che le operazioni di installazione vengano sempre eseguite nella stessa sequenza, creando sempre lo stesso ambiente. Come qualsiasi altro codice, il modello di infrastruttura può essere versionato e archiviato in un sistema di gestione del codice. Se le modifiche all'infrastruttura causano problemi, puoi facilmente tornare a una versione precedente e reinstallare l'ambiente che sai che funziona.

Il modo migliore per distribuire molti servizi basati su cloud è utilizzare i container. Il **Dockerfile** diventa in pratica una definizione della tua infrastructure as a code. L'utilizzo dei container rende molto semplice fornire ambienti di esecuzione identici. Quando aggiorni il tuo software, esegui nuovamente il processo di creazione dell'immagine per creare una nuova immagine che includa il software modificato.

10.3 DevOps measurement

E' necessario raccogliere e analizzare dati sul prodotto e sul processo DevOps, in modo da valutare se il processo è efficace e in miglioramento.

Si raccolgono e analizzano dati sui processi di sviluppo, test e distribuzione, per analizzare il processo DevOps; Su prestazioni, affidabilità e accettabilità del software per i clienti, per analizzare il servizio; sui dati relativi a come i clienti utilizzano il prodotto, per analizzare l'utilizzo; sui dati relativi a come il prodotto contribuisce al successo complessivo dell'azienda, per analizzare il successo aziendale

A volte è impossibile misurare direttamente ciò che desideri (come la soddisfazione del cliente). Devi quindi dedurre le informazioni da altre metriche (come il numero di clienti che ritornano). Anche la misurazione del software deve essere automatizzata. Dovresti usare un sistema di monitoraggio per raccogliere dati sulle prestazioni e la disponibilità del sw. Tuttavia, ci sono problemi nella misurazione del processo perché coinvolge anche le persone. Persone diverse possono registrare le informazioni in modo diverso.

Per le **metriche di processo**, vorresti vedere una diminuzione del **numero di distribuzioni fallite**, il tempo **medio di recupero** dopo un guasto del servizio e il **tempo di consegna** dallo sviluppo alla distribuzione. Spereresti di vedere un aumento della **frequenza di distribuzione** e del numero di **righe di codice modificate** che vengono rilasciate. Per le metriche di **servizio**, la **disponibilità** e le **prestazioni** dovrebbero essere stabili o migliorare, il numero di **reclami dei clienti** dovrebbe diminuire e il numero di **nuovi clienti** dovrebbe aumentare.

I dati raccolti dovrebbero essere analizzati **settimanalmente** e presentati in un'unica schermata che mostri le performance della settimana corrente e di quella precedente.

Gli strumenti di integrazione continua come Jenkins possono raccogliere dati sulle distribuzioni, i test riusciti e così via. I fornitori di servizi cloud spesso dispongono di software di monitoraggio, come Cloudwatch di Amazon, che può fornire dati sulla disponibilità e sulle prestazioni.

Oltre a utilizzare questi strumenti, puoi utilizzare i file di log, dove le voci nei log sono eventi con timestamp che riflettono le azioni dei clienti e/o le risposte del software.

Sicurezza e Privacy

La **sicurezza del software** è l'insieme di misure progettate per proteggere il sw da accessi non autorizzati, vulnerabilità, e attacchi malevoli che possono compromettere la riservatezza, l'integrità o la disponibilità dei dati e delle risorse.

L'obiettivo degli attacchi è quello di **rubare dati** o di **dirottare** un computer per fini criminali.

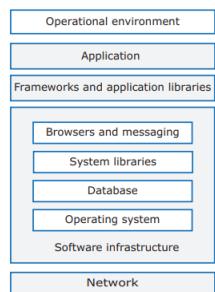
Minacce alla disponibilità: Attacchi volti a impedire agli utenti legittimi di accedere al sistema o ai suoi servizi, in modo da rendere il sistema inutilizzabile. (DDoS)

Minacce all'integrità: Sono attacchi che mirano a modificare, corrompere o distruggere dati o componenti del sistema, compromettendo la loro accuratezza e affidabilità. (Virus che alterano i file o Ransomware che bloccano i dati fino al pagamento di un riscatto).

Minacce alla riservatezza: Si riferiscono a tentativi di accedere a informazioni sensibili o private senza autorizzazione. (Furto di dati sensibili per usi illeciti).

Il sw dipende da una piattaforma di esecuzione che include un SO, un web server, un sistema di runtime del linguaggio, (un framework) e un database. Gli attacchi possono mirare a qualsiasi livello di questa struttura, dai router di rete ai componenti e alle librerie utilizzati dal sw. Tuttavia, gli attaccanti in genere si concentrano sull'**infrastruttura software**.

Figure 7.2 System infrastructure stack



Le **politiche** per minimizzare il rischio di un attacco riuscito sono le seguenti. Dovreste avere procedure di **autenticazione e autorizzazione** che garantiscano che tutti gli utenti abbiano un'autenticazione forte e che abbiano correttamente impostato i permessi di accesso. Il software infrastrutturale dovrebbe essere **correttamente configurato** e gli **aggiornamenti di sicurezza** che correggono le vulnerabilità dovrebbero essere applicati non appena disponibili. Il sistema dovrebbe essere **controllato regolarmente** per rilevare possibili accessi non autorizzati. Se vengono rilevati attacchi, potrebbe essere possibile attuare strategie di resistenza che minimizzano gli effetti dell'attacco. Dovrebbero essere implementate politiche di **backup** per garantire la conservazione di copie non danneggiate dei file di programma e dei dati. Questi possono poi essere ripristinati dopo un attacco.

I **tipi di attacco** dipendono dal tipo di sistema, dalle potenziali vulnerabilità nel sistema e dall'ambiente in cui il sistema è utilizzato.

Un **requisito** è che gli attaccanti siano in grado di **autenticarsi** nel sistema. Questo implica il furto delle credenziali di un utente legittimo. Il metodo più comune per farlo è utilizzare tecniche di **social engineering**, in modo che un utente invii le sue credenziali all'attaccante. In alternativa viene installato un malware, come un key logger, che intercetta le credenziali registrando i tasti digitati dall'utente.

7.1.1 Attacchi di injection

Gli **attacchi di injection** sono un tipo di attacco in cui viene inserito in un campo di input valido del **codice dannoso**.

I linguaggi C/C++ non controllano automaticamente che un'assegnazione a un elemento di un array rientri nella dimensione dell'array fissata.

Un attacco di **buffer overflow** consiste nel **costruire una stringa** di input che include **istruzioni eseguibili**, di lunghezza tale da andare in **overflow**, e inserirla in input. La stringa sovrascrive la memoria e in particolare l'indirizzo di ritorno della funzione in esecuzione, trasferendo il controllo al codice malevolo.

Gli attacchi di **SQL Poisoning** sfruttano il fatto che un input dell'utente viene usato in una query SQL, **iniettando** al suo interno **codice sql** malevolo.

- Attacker can do injection attack when user input is part of an SQL command

```
accNum = getAccountNumber ()  
SQLstat = "SELECT * FROM AccountHolders WHERE accountnumber = '" + accNum + "';"  
database.execute (SQLstat)
```

Please enter your account number:

'34200645' → SELECT * FROM AccountHolders WHERE accountnumber = '34200645'

'10010010' OR '1'='1' → SELECT * FROM AccountHolders

Gli attacchi di SQL poisoning sono possibili solo quando il sistema non controlla la **validità degli input**.

7.1.2 Attacchi di cross-site scripting

Gli attacchi di **cross-site scripting (XSS)** consistono nell'aggiungere **codice JS** malevolo a una pagina web, inserendolo in **sezioni di input** della pagina o sostituendo codice legittimo del sito.

La **pagina modificata** che include lo script malevolo viene inviata dal server al **browser** della vittima, dove il codice malevolo viene eseguito. Lo script malevolo mira a **rubare** i dati degli utenti o reindirizzarli verso un altro sito web. I cookie possono essere rubati, rendendo possibile un attacco di session hijacking.

La **validazione degli input** è una **contromisura** chiave: Non deve contenere codici non desiderati. Inoltre è fondamentale **controllare i dati recuperati dal database** prima di inserirli in una pagina web generata dinamicamente, evitando che script inseriti precedentemente vengano eseguiti. Infine, **utilizzare l'encoding HTML** che permette di interpretare i caratteri speciali (come <, >, e &) come normale testo e non come codice eseguibile.

7.1.3 Attacchi di session hijacking

Una **sessione** è il periodo in cui l'autenticazione dell'utente rimane **valida**, evitando la necessità di ri-autenticarsi. Termina con il logout o dopo un periodo di inattività.

L'autenticazione utilizza un **session cookie**, un token inviato dal server al client all'inizio di una sessione. Questo cookie permette al server di tracciare le azioni dell'utente associando ogni richiesta HTTP alle precedenti.

Il **session hijacking** consiste nell'acquisire un **session cookie** valido e utilizzarlo per impersonare un utente legittimo. Per acquisire il cookie può essere usato il cross-site scripting o il monitoraggio del traffico.

Nel **session hijacking attivo**, l'attaccante prende il controllo di una sessione utente ed **esegue azioni** dell'utente su un server. Nel **session hijacking passivo**, l'attaccante si limita a monitorare il traffico tra il client e il server, rubando dati.

Contromisure: Crittografare il traffico di rete tra il client e il server, utilizzando HTTPS invece di HTTP. Utilizzare l'autenticazione multifattoriale e richiedere la conferma per nuove azioni potenzialmente dannose. Utilizzare timeout relativamente brevi per le sessioni. Le richieste future vanno indirizzate a una pagina di autenticazione.

7.1.4 Attacchi di denial-of-service

Gli attacchi di **denial-of-service (DoS)** sono attacchi a un sistema software con l'intento di renderlo **non disponibile** per l'uso normale.

Gli attacchi di **distributed denial-of-service (DDOS)** coinvolgono computer distribuiti, che inviano centinaia di **migliaia di richieste** di servizio a un'applicazione web, in modo che il servizio diventi non disponibile.

Per **combattere** un attacco DDOS deve essere utilizzato software specializzato che può rilevare e **scartare i pacchetti** in arrivo, in modo da ripristinare i servizi alla normale operatività.

Altri tipi di dos prendono di mira gli utenti. Gli attacchi di **Lockout User** sfruttano una politica che **blocca** un utente dopo un certo **numero di tentativi** di autenticazione falliti, usando user email rubate. Le **soluzioni** sono bloccare un utente solo per un **breve periodo** dopo un'autenticazione fallita, e registrare gli indirizzi IP normalmente usati dagli utenti per accedere al sistema, bloccando tentativi ripetuti da altri indirizzi.

7.1.5 Attacchi di brute force

Gli attacchi di **brute force** consistono nel creare diverse password in base alle informazioni possedute sull'utente e provare ripetutamente il login finché non si ottiene l'accesso. Può essere utilizzato un generatore di stringhe che produce ogni possibile combinazione di lettere.

La soluzione consiste nell'obbligare gli utenti a impostare **password lunghe** che non siano contenute in un vocabolario e non siano parole comuni. In alternativa, può essere adottata l'autenticazione a **due fattori**.

7.2 Autenticazione

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

L'autenticazione **basata sulla conoscenza** si affida al fatto che gli utenti forniscano **informazioni personali** segrete, come **password** o domande personali.

L'autenticazione **basata sul possesso** si basa sul fatto che l'utente possieda un **dispositivo fisico** che può generare informazioni conosciute dal sistema, che l'utente inserirà al momento dell'autenticazione. Un esempio è l'invio di un codice al numero di telefono dell'utente.

L'autenticazione **basata sugli attributi** si basa su un **attributo biometrico** unico dell'utente, come un'impronta digitale, registrato nel sistema.

Per rafforzare l'autenticazione, molti sistemi ora utilizzano l'autenticazione multifattoriale, che combina i diversi approcci.

Nei prodotti forniti come servizi cloud, l'approccio più usato è quello basato sulla conoscenza tramite **password**, possibilmente supportato da altre tecniche. Le debolezze dell'uso di password sono le seguenti: Gli utenti scelgono password facili da ricordare, ma facili da indovinare per gli attaccanti.: Gli utenti inseriscono in un sito falso i dettagli del loro login e della password. Gli utenti utilizzano la stessa password per diversi siti. Se un sito viene violato, gli attaccanti possono utilizzarla sugli altri siti. Gli utenti dimenticano regolarmente le loro password. Gli attaccanti usano il meccanismo di reimpostazione per inserire la propria password.

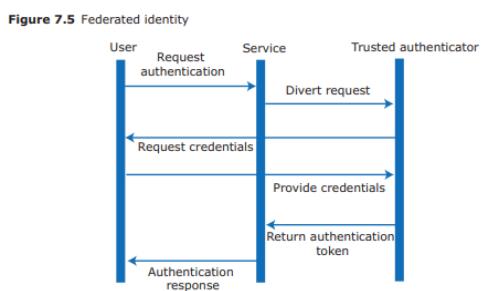
Le **soluzioni** sono obbligare gli utenti a **impostare password complesse** e richiedere agli utenti di **rispondere a domande** oltre a inserire una password.

Il livello di autenticazione necessario dipende dal prodotto. Se non memorizzate **dati sensibili** sugli utenti l'autenticazione basata sulla **conoscenza** potrebbe essere sufficiente. Se, invece gestite dati sensibili, dovete utilizzare l'autenticazione 2FA.

Sebbene toolkit e librerie come OAuth siano disponibili per la maggior parte dei principali linguaggi di programmazione, è necessario un notevole sforzo di programmazione. E' meglio considerare l'autenticazione come un microservizio.

7.2.1 Identità federata

L'approccio di **identità federata** consiste nell'utilizzare un **servizio esterno** per l'autenticazione. Per esempio "Accedi con Google" o "Accedi con Facebook".



Nel caso di “Accedi con Google”, un utente viene reindirizzato al servizio di identità di Google. Il servizio richiede all’utente le credenziali del proprio account Google e effettua l’autenticazione. In caso di successo restituisce un **token** di avvenuto login al sito reindirizzante, che si occupa di comunicare all’utente l’avvenuta autenticazione.

Il vantaggio dell’identità federata per gli utenti è che hanno un unico set di credenziali conservato da un servizio di identità affidabile. Non è necessario mantenere il proprio database di password e altre informazioni riservate degli utenti. Il provider di identità può fornire informazioni aggiuntive sugli utenti che possono essere utilizzate per personalizzare il vostro servizio.

La verifica dell’identità utilizzando Google o Facebook è accettabile per prodotti consumer destinati a clienti individuali. Per i prodotti aziendali, potete comunque utilizzare l’identità federata, con l’autenticazione basata sul sistema di gestione delle identità della stessa azienda.

La maggior parte dei servizi di autenticazione federata utilizza il protocollo OAuth per restituire token al sistema chiamante. Tuttavia, i token OAuth indicano esclusivamente che l’accesso è autorizzato, senza fornire dettagli sull’utente autenticato. Questo limita la possibilità di utilizzarli per decidere quali risorse del sistema devono essere accessibili. Per superare questa limitazione, è stato introdotto OpenID Connect, un’estensione di OAuth che arricchisce i token con informazioni sull’utente, consentendo al sistema chiamante di prendere decisioni più informate sull’accesso alle risorse.

7.2.2 Autenticazione tramite dispositivi mobili

Le tastiere dei dispositivi mobili sono scomode e soggette a errori; se insistete per l’uso di password complesse, è probabile vengano digitate in modo errato.

Come alternativa al login con password può essere utilizzato il seguente metodo: Quando installano l’app, gli utenti si autenticano utilizzando le proprie credenziali, ottenendo un token di accesso che **viene installato** sul device. Ogni volta che l’app viene avviata, il token viene inviato al fornitore del servizio per autenticare l’utente del dispositivo. Il token può scadere dopo un certo periodo di tempo, costringendo gli utenti a ri-autenticarsi.

Se un device viene rubato, qualcun altro potrebbe accedere al prodotto. Si può verificare che il proprietario abbia impostato un metodo di autenticazione per accedere al device. In caso contrario, è necessario richiedere di ri-autenticarsi ogni volta che l’app viene avviata.

L’autenticazione basata su certificati digitali utilizza certificati emessi da autorità fidate come token di autenticazione, offrendo maggiore sicurezza rispetto ai semplici token. Questi certificati permettono anche il single sign-on per più applicazioni, ma richiedono una gestione complessa, che può essere autonoma o delegata a servizi specializzati.

Inoltre, è fondamentale crittografare le informazioni di autenticazione durante la trasmissione tra client e server utilizzando connessioni HTTPS.

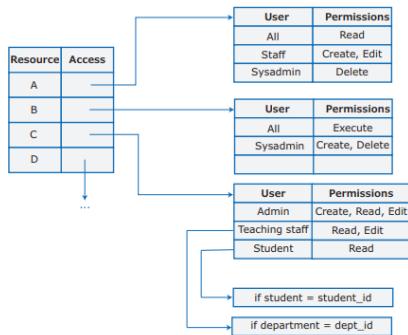
7.3 Autorizzazione

L'autenticazione assicura che un utente è chi dice di essere. **L'autorizzazione** è un processo in cui l'identità viene utilizzata per controllare l'accesso alle risorse del sistema.

La **politica di controllo degli accessi** è un insieme di regole che definiscono le informazioni da gestire, chi ne ha accesso e il tipo di accesso consentito.

Le **liste di controllo degli accessi** (ACL) sono tabelle che collegano gli utenti alle risorse e specificano cosa possono fare quegli utenti.

Figure 7.8 Access control lists



Le ACL basate su permessi individuali possono essere molto grandi. La loro dimensione può essere ridotta assegnando gli utenti a gruppi e poi attribuendo i permessi al gruppo.

Non vale la pena sviluppare il proprio sistema di controllo degli accessi, ma è meglio utilizzare i meccanismi ACL forniti dal file system sottostante.

7.4 Crittografia

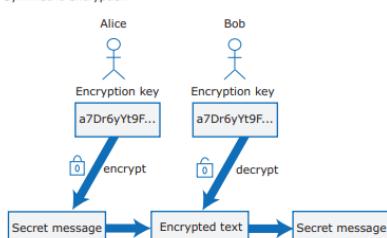
La **crittografia** è il processo di rendere un **documento illeggibile** applicando una **trasformazione algoritmica**. L'algoritmo utilizza una **chiave segreta** come base per questa trasformazione. È possibile decodificare il testo crittografato applicando la trasformazione inversa.

Le moderne tecniche di crittografia sono considerate praticamente inviolabili con la tecnologia attualmente disponibile. Tuttavia, la storia ha dimostrato che una crittografia apparentemente forte può essere violata quando nuove tecnologie diventano disponibili (Computer quantistici).

7.4.1 Crittografia simmetrica e asimmetrica

Nella **crittografia simmetrica**, la **stessa chiave** viene utilizzata sia per codificare che per decodificare il messaggio. Il mittente crittografa il messaggio con questa chiave. Quando il destinatario riceve il messaggio, lo decodifica utilizzando la stessa chiave per leggere il contenuto.

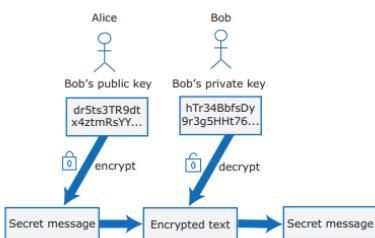
Figure 7.10 Symmetric encryption



Il problema è condividere la chiave in modo sicuro. Se il mittente invia semplicemente la chiave al destinatario, un attaccante potrebbe intercettare il messaggio e ottenere l'accesso alla chiave, decodificando le comunicazioni future.

La **crittografia asimmetrica** utilizza **chiavi diverse** per crittografare e decrittografare i messaggi. Ogni utente ha una chiave **pubblica** e una chiave **privata**. I messaggi possono essere crittografati utilizzando una delle due chiavi, ma possono essere decrittografati solo con l'altra chiave.

Figure 7.11 Asymmetric encryption

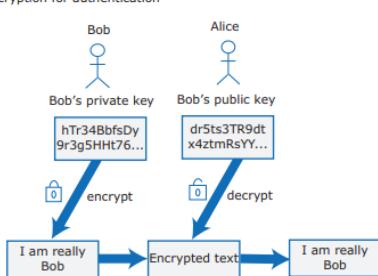


Le chiavi pubbliche possono essere **pubblicate** dal proprietario della chiave. Chiunque può utilizzare una chiave pubblica. Tuttavia, un messaggio può essere decrittografato solo con la chiave privata dell'utente, quindi può essere letto solo dal destinatario previsto.

Per garantire la **confidenzialità**, il mittente crittografa un messaggio utilizzando la **chiave pubblica** del destinatario. Il destinatario decrittografa il messaggio utilizzando la sua **chiave privata**, che **solo lui conosce**. Solo tale destinatario può decrittografare il messaggio.

Per garantire **l'autenticazione** il mittente di un messaggio crittografa il messaggio con la **propria chiave privata**, mentre il destinatario decrittografa con la **chiave pubblica del mittente**. Solo tale mittente può avere inviato il messaggio, essendo decrittografabile solo con la sua chiave pubblica.

Figure 7.12 Encryption for authentication



Per lo stesso livello di sicurezza, la crittografia asimmetrica richiede 1000 volte più tempo rispetto a quella simmetrica. Viene usata solo per crittografare messaggi brevi.

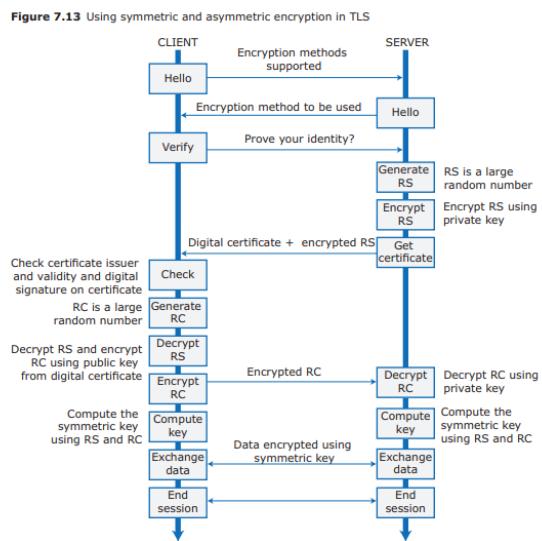
7.4.2 TLS e certificati digitali

I **certificati digitali** sono emessi da un'autorità di certificazione (CA), che è un servizio di verifica dell'identità affidabile. La CA codifica nel certificato digitale le informazioni fornite dall'azienda acquirente. Se un certificato è emesso da una CA riconosciuta, l'identità del server può essere considerata affidabile. I browser web e le app che utilizzano https includono un elenco di fornitori di certificati affidabili.

La CA critta le informazioni nel certificato utilizzando la propria **chiave privata** per creare una firma unica. Questa firma è inclusa nel certificato insieme alla chiave pubblica della CA. Per verificare che il certificato sia valido, è possibile decriptare la firma utilizzando la chiave pubblica della CA (autenticazione asimmetrica). Le informazioni decriptate dovrebbero corrispondere alle altre informazioni nel certificato. Se non corrispondono, il certificato è stato falsificato.

Il protocollo **HTTPS** è il protocollo **HTTP** più un livello di crittografia chiamato **TLS** (**Transport Layer Security**).

TLS viene usato per **verificare l'identità** del server web e per **crittografare** le comunicazioni in modo che non possano essere lette da un attaccante che intercetta i messaggi tra il client e il server.



Quando un client e un server desiderano scambiarsi informazioni in modo sicuro, avviano una connessione TLS. Inizialmente, il server invia al client il proprio certificato digitale, che contiene la sua **chiave pubblica**. Successivamente, il server genera un numero casuale e lo cifra utilizzando la propria chiave privata, inviando il risultato al client. Il client, a sua volta, può decifrare il messaggio usando la chiave pubblica del server, ottenendo il numero casuale originale. Successivamente, il client genera un proprio numero casuale, lo cifra con la chiave pubblica del server e lo invia al server. Quest'ultimo decifra il messaggio con la propria chiave privata, ottenendo così il numero casuale del client.

A questo punto, sia il client che il server possiedono due numeri casuali lunghi, uno generato da ciascuna delle parti. Utilizzando un algoritmo di derivazione della chiave, entrambi calcolano indipendentemente una **chiave segreta condivisa** che sarà utilizzata per **crittografare e decrittografare i messaggi successivi in modo simmetrico**.

7.4.3 Crittografia dei dati

Dovresti crittografare i dati **sensibili** degli utenti ogni volta che è praticabile farlo. I **dati in transito** dovrebbero **sempre** essere crittografati. I trasferimenti di dati su internet devono sempre utilizzare HTTPS. I **dati a riposo**, memorizzati e non utilizzati, devono essere **sempre** crittografati. I dati in uso **non devono** essere crittografati, poiché la crittografia dei dati in uso rallenta il tempo di risposta del sistema.

La crittografia dei dati è possibile a quattro livelli diversi nel sistema. **Applicazione**, in cui l'applicazione decide quali dati devono essere crittografati e li decriptografa immediatamente prima del loro utilizzo; **Database** in cui il DBMS può crittografare l'intero database quando viene chiuso, con la decriptografia che avviene alla riapertura; **File**, in cui il SO crittografa i file individuali quando vengono chiusi e li decriptografa quando vengono riaperti; **Supporti (Media)**, in cui il SO crittografa i dischi quando vengono smontati e li decriptografa quando vengono rimontati.

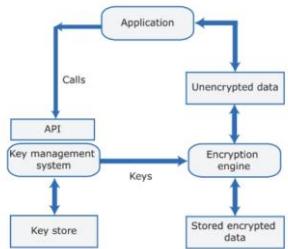
Generalmente, una protezione maggiore viene fornita ai livelli più alti di questa pila, poiché i dati vengono decriptografati per un periodo di tempo più breve.

7.4.4 Gestione delle chiavi

Il processo di **gestione delle chiavi** garantisce che le chiavi siano **generate, archiviate e accessibili** in modo sicuro solo da utenti **autorizzati**.

Se la gestione delle chiavi viene fatta in modo **errato**, utenti non autorizzati potrebbero decrittare dati che dovrebbero essere **privati**. Se si perdono le chiavi, i dati crittografati potrebbero diventare **inaccessibili**.

Un **Key Management System (KMS)** è un **database** specializzato progettato per **archiviare** e gestire in modo sicuro **chiavi**, certificati digitali e altre informazioni riservate. Può offrire funzionalità come la generazione di chiavi, regolare chi può accedere alle chiavi, e il trasferimento sicuro delle chiavi dal KMS ad altri nodi della rete.



Per ridurre i rischi di violazione, le chiavi dovrebbero essere cambiate regolarmente. Ciò significa che i dati archiviati potrebbero essere crittografati con una chiave diversa da quella attuale nel sistema. Un KMS deve quindi mantenere più versioni delle chiavi con timestamp, in modo che i backup di sistema e gli archivi possano essere decrittati se necessario.

7.5 Privacy

La **privacy** è un concetto **sociale** che riguarda **la raccolta, la diffusione e l'uso appropriato** delle informazioni personali detenute da una terza parte, come un'azienda.

In molti paesi, il diritto alla privacy individuale è protetto dalle **leggi sulla protezione dei dati** (es. GDPR), che limitano la raccolta, la diffusione e l'uso dei dati personali agli scopi per i quali sono stati raccolti.

Il **soggetto dei dati** è l'individuo i cui dati vengono gestiti. Possiede il diritto di accedere ai dati memorizzati e di correggere errori. Deve dare il suo consenso all'uso dei propri dati e può chiedere che i dati pertinenti vengano cancellati. Il **responsabile del trattamento** è responsabile della conservazione sicura dei dati in una posizione coperta dalla legislazione sulla protezione dei dati. Deve fornire l'accesso ai dati e usarli solo per lo scopo per cui sono stati raccolti.

I principi di protezione dei dati sono i seguenti. Gli utenti del vostro prodotto devono essere informati su quali dati vengono raccolti durante l'uso del prodotto e devono avere il controllo sulle informazioni personali che raccogliete da loro. Dovete informare gli utenti sul motivo per cui i dati vengono raccolti e non dovete utilizzare quei dati per altri scopi. Dovete sempre ottenere il consenso dell'utente prima di divulgare i suoi dati ad altre persone. Non dovete conservare i dati più a lungo del necessario. Se un utente elimina un account, dovete eliminare i dati personali associati a quell'account. Dovete conservare i dati in modo sicuro, in modo che non possano essere manomessi o divulgati a persone non autorizzate. Dovete permettere agli utenti di scoprire quali dati personali memorizzate. Dovete fornire un modo per correggere eventuali errori nei loro dati personali. Non dovete memorizzare i dati in paesi dove si applicano leggi sulla protezione dei dati più deboli, a meno che non vi sia un accordo esplicito che le regole più rigorose sulla protezione dei dati saranno rispettate.

Se offrite un prodotto direttamente ai consumatori e non rispettate le normative sulla privacy, potreste essere soggetti ad azioni legali. Se il vostro prodotto è un prodotto aziendale, i clienti aziendali richiedono garanzie sulla privacy per non essere a rischio di azioni legali da parte dei loro utenti. Le informazioni che il vostro software deve raccogliere dipendono dalla funzionalità del prodotto e dal modello di business che utilizzate. Non dovreste raccogliere informazioni personali di cui non avete bisogno. Dovreste stabilire una politica sulla privacy che definisca come le informazioni sensibili sugli utenti vengono raccolte, archiviate e gestite. Se raccogliete i dati degli utenti per mirare la pubblicità agli utenti o per fornire servizi pagati da altre aziende, dovreste chiarire che raccogliete dati per questo scopo, consentendo agli utenti di vietare l'uso dei propri dati.

Security and microservices

Nel monolite la comunicazione avviene all'interno di un singolo processo. Ciò significa che la superficie d'attacco è limitata a pochi punti.

Nei microservizi la comunicazione avviene tramite chiamate remote, con un numero potenzialmente **elevato** di **punti di ingresso**. Questo porta a un ampliamento della **superficie d'attacco**, poiché ogni endpoint può rappresentare una vulnerabilità. In questo scenario, la sicurezza dell'intera applicazione è pari alla sicurezza del **suo punto più debole**.

Nel monolite lo screening della sicurezza viene effettuato una sola volta, e la richiesta viene quindi inoltrata al componente corrispondente, riducendo il sovraccarico.

Nei microservizi **ogni** microservizio deve effettuare uno **screening** di sicurezza indipendente. Questo può includere connessioni a servizi remoti per il rilascio di token di sicurezza, aumentando così il carico computazionale.

Le soluzioni possibili sono due: "**Trust-the-network**", che consiste nell'ignorare i controlli di sicurezza per ogni microservizio e affidarsi alla rete, sacrificando così la sicurezza, e "**Zero-trust networking**", che prevede di trattare ogni microservizio come **non affidabile** per default.

La comunicazione tra microservizi deve avvenire su **canali protetti** per garantire la sicurezza.

Se si utilizzano i certificati, ogni microservizio deve essere dotato di un **certificato** e della relativa **chiave privata** per autenticarsi verso un altro microservizio durante le interazioni. Il microservizio ricevente deve essere in grado di **validare il certificato** del microservizio chiamante. Deve essere previsto un sistema per **revocare e ruotare i certificati** in caso di necessità.

Per gestire implementazioni su larga scala che coinvolgono centinaia di microservizi, è essenziale utilizzare sistemi di **automazione**, altrimenti le operazioni di provisioning, validazione e gestione dei certificati diventano insostenibili.

Un **Log** è una registrazione di eventi di un servizio. Può essere aggregato per produrre **metriche** che riflettono lo stato del sistema (ad esempio, il numero medio di richieste di accesso non valide per ora), utili per generare alert.

A differenza delle applicazioni monolitiche, dove la richiesta rimane all'interno di un singolo processo, in un'architettura a microservizi una richiesta può attraversare più servizi. Questo rende difficile **correlare le richieste** tra i microservizi.

I **Traces** permettono di tracciare una richiesta dal punto in cui entra nel sistema fino al punto in cui lo lascia, offrendo una visione del suo percorso.

Strumenti per la tracciabilità che possono essere utilizzati sono **Prometheus** e **Grafana** utili per monitorare le richieste in ingresso e produrre metriche, e **Jaeger** e **Zipkin**, utili per il tracing distribuito, consentono di seguire il percorso delle richieste attraverso più microservizi.

I container non cambiano stato dopo l'avvio, garantendo coerenza ma complicando la gestione delle credenziali e delle politiche di accesso. Ogni servizio deve mantenere una lista dinamica di client autorizzati e aggiornare periodicamente le credenziali per la sicurezza. Le credenziali devono essere ruotate regolarmente e possono essere memorizzate nel file system del container, iniettandole al momento dell'avvio per evitare di hardcodarle nel codice.

Il contesto utente deve essere passato esplicitamente tra microservizi, con la necessità che il servizio ricevente accetti e verifichi il contesto trasmesso. È fondamentale un sistema di fiducia tra microservizi per garantire l'autenticità e la sicurezza del contesto utente condiviso. I JWT permettono di includere il contesto utente in un formato crittograficamente sicuro, autenticando i messaggi e garantendo l'integrità durante la trasmissione tra microservizi.

In un'architettura **poliglotta**, ogni team può adottare pratiche di sicurezza specifiche e strumenti distinti per i test di sicurezza statici (analisi del codice) e dinamici (esecuzione del software). Le responsabilità legate alla sicurezza sono distribuite tra i diversi team, creando potenziali disallineamenti o mancanza di standardizzazione.

Molte organizzazioni adottano un approccio misto, combinando un **team di sicurezza centralizzato** per garantire coerenza e standard, con esperti di sicurezza inseriti nei team di sviluppo per affrontare problemi specifici.

Multivocal review of white and grey literature

Lo studio condotto intende rispondere a due domande di ricerca: Quali sono i **security smells** più riconosciuti nei microservizi e come rifactorizzare un'applicazione per mitigare gli effetti di un **security smell**?

È stata quindi costruita una tassonomia che include **10 refactorings** associati a **10 smells** che potenzialmente impattano sulle **3 proprietà di sicurezza** fondamentali, ovvero confidenzialità, integrità e autenticazione.

Per **Confidenzialità** si intende il grado in cui un sistema garantisce che i dati siano **visibili** solo a utenti autorizzati. Per **Integrità** si intende il grado in cui un sistema previene **modifiche** non autorizzate ai dati. Per **Autenticità** si intende il grado in cui l'**identità** di un soggetto può essere provata come quella dichiarata.

Security smell 1: Insufficient Access Control

Il problema di **Insufficient Access Control** si verifica quando un'applicazione basata su microservizi **non implementa un controllo degli accessi in uno o più microservizi**, **violando** potenzialmente la **Confidentiality** dei dati e delle funzioni di business dei microservizi interessati.

I microservizi possono essere vulnerabili al problema del "**confused deputy**", in cui un attaccante può ingannare un servizio per accedere a dati non autorizzati. Nei microservizi i permessi dei client devono essere verificati ad ogni richiesta. L'identità degli utenti deve essere verificata senza introdurre latenza aggiuntiva o conflitti causati da chiamate frequenti a un servizio centralizzato.

Una **soluzione efficace** è rappresentata dall'uso di **OAuth 2.0**, che è un framework di sicurezza basato su token per il controllo degli accessi delegato, che consente al

proprietario di una risorsa di concedere a un client l'accesso a una determinata risorsa per un periodo e uno scope limitato.

Security smell 2: Publicly Accessible Microservices

Il problema dei **Publicly Accessible Microservices** si verifica quando i **microservizi di un'applicazione sono direttamente accessibili dai client esterni**.

Ogni microservizio espone la propria API e richiede un meccanismo per assicurarsi che ogni richiesta sia autenticata e autorizzata a eseguire le funzioni richieste, rischiando di richiedere tutte le credenziali di un utente per ogni richiesta, aumentando il rischio di esposizione delle credenziali e **violando la Confidentiality**. Inoltre, ciascun microservizio deve applicare politiche di sicurezza comuni a tutte le funzioni dell'applicazione.

Il **refactoring suggerito** è utilizzare un **API Gateway**. L'API Gateway applica **centralmente** la sicurezza per tutte le richieste in ingresso, includendo autenticazione, autorizzazione, throttling e validazione dei contenuti dei messaggi per minacce note. Questa soluzione permette anche di proteggere i microservizi dietro un firewall, consentendo all'API Gateway di gestire le richieste esterne e di comunicare con i microservizi interni.

Security smell 3: Unnecessary Privileges to Microservices

Il problema degli **Unnecessary Privileges to Microservices** si verifica quando **a un microservizio vengono concessi livelli di accesso, permessi o funzionalità non necessari per svolgere le sue funzioni di business**. Questo problema aumenta la superficie d'attacco, poiché un intruso che prende il controllo di un microservizio potrebbe leggere o modificare dati non rilevanti per il servizio, **violando Confidentiality e Integrity**.

Il **refactoring suggerito** consiste nel seguire il **Principio del Minimo Privilegio**, concedendo ai microservizi solo i permessi strettamente necessari per eseguire le loro funzioni.

Security smell 4: Own Crypto Code

L'**uso di Own Crypto Code**, ovvero **soluzioni crittografiche** proprie può **compromettere** le proprietà di **Confidentiality, Integrity e Authenticity**, poiché tali soluzioni spesso risultano inadeguate se non ampiamente testate, portando inoltre a una falsa sensazione di sicurezza e a vulnerabilità gravi.

Il **refactoring suggerito** consiste nell'utilizzare **tecnologie crittografiche consolidate**, minimizzando la scrittura di codice crittografico personalizzato.

Security smell 5: Non-Encrypted Data Exposure

Il problema di **Non-Encrypted Data Exposure** si verifica quando **dati sensibili non vengono crittografati durante l'archiviazione**, esponendoli a potenziali violazioni da parte di un intruso, **compromettendo Confidentiality, Integrity e Authenticity**. Il **refactoring suggerito** consiste nel **crittografare tutti i dati sensibili a riposo**. I dati devono essere decifrati solo quando necessario.

Security smell 6: Hardcoded Secrets

Gli **Hardcoded Secrets** si verificano quando credenziali o chiavi di accesso vengono mantenute **direttamente nel codice sorgente** o negli script di deployment **violando Confidentiality, Integrity e Authenticity**. Questo aumenta il rischio di esposizione accidentale, ad esempio tramite log o processi figli.

Il **refactoring suggerito** consiste nel **crittografare i segreti a riposo**, evitando di memorizzare credenziali insieme al codice sorgente o nei repository. Inoltre, devono adottare pratiche sicure per il passaggio di segreti tra applicazioni.

Security smell 7: Non-Secured Service-to-Service Communications

Il problema di **Non-Secured Service-to-Service Communications** si verifica quando due microservizi comunicano senza utilizzare un **canale sicuro**, anche se si trovano nella stessa rete. Poiché le applicazioni basate su microservizi sono distribuite, le interfacce di comunicazione e i canali proliferano, aumentando la superficie d'attacco. Ogni API esposta da un microservizio e ogni canale di comunicazione rappresentano potenziali vettori di attacco che un intruso potrebbe sfruttare con attacchi come man-in-the-middle, intercettazioni e alterazioni, **compromettendo Confidentiality, Integrity e Authenticity**.

Il **refactoring suggerito** consiste nell'usare il **Mutual Transport Layer Security (TLS)**. Questo meccanismo protegge i dati in transito tramite crittografia bidirezionale, garantendo la riservatezza e l'integrità dei dati, oltre a consentire l'autenticazione reciproca tra i microservizi.

Security smell 8: Unauthenticated Traffic

Il problema di **Unauthenticated Traffic** si verifica quando un'applicazione **non autentica** il traffico API proveniente da sistemi esterni o da richieste tra i microservizi stessi. La mancata autenticazione espone i microservizi a violazioni di sicurezza, come alterazioni dei dati, denial of service o escalation dei privilegi **violando l'Authenticity**. È essenziale che i microservizi possano autenticarsi reciprocamente, specialmente durante il passaggio del contesto utente tra microservizi.

Il **refactoring suggerito** consiste nell'usare il **Mutual TLS** per l'autenticazione dei microservizi e **OpenID Connect** per gestire l'autenticazione degli utenti. **OpenID Connect** utilizza token ID (JSON Web Token) che contengono informazioni autenticate sull'utente, consentendo ai microservizi di verificare l'identità degli utenti in modo distribuito e sicuro.

Security smell 9: Multiple User Authentication

Il problema di **Multiple User Authentication** si presenta quando un'applicazione **gestisce l'autenticazione** degli utenti tramite più punti di accesso. Ogni punto di accesso aumenta la superficie d'attacco e può **compromettere l'Authenticity** dell'applicazione. Inoltre, più punti di autenticazione aumentano i costi di manutenzione e rendono l'applicazione meno usabile.

Il **refactoring suggerito** consiste nell'implementare il **Single Sign-On (SSO)**. Questo approccio prevede un unico punto di accesso per gestire l'autenticazione degli utenti. Per implementare il SSO si può aggiungere un **API Gateway** e utilizzare **OpenID Connect** per condividere il contesto utente tra i microservizi.

Security smell 10: Centralized Authorization

Il problema di **Centralized Authorization** si verifica quando l'autorizzazione è gestita in modo centralizzato, spesso tramite l'API Gateway, senza un controllo dettagliato a livello dei singoli microservizi. Ciò può creare colli di bottiglia, riducendo le prestazioni e aumentando i rischi, come il "confused deputy problem", in cui i microservizi si fidano ciecamente dell'API Gateway. Questo smell **viola la Authenticity**.

Il **refactoring suggerito** consiste nell'implementare un'autorizzazione decentralizzata utilizzando un meccanismo basato su token, come i **JSON Web Token (JWT)**. Questo approccio consente ai microservizi di verificare autonomamente le richieste e applicare le proprie politiche di controllo degli accessi.

Eliminare i security smells

Il processo per eliminare i **security smells** in un'applicazione basata su microservizi si articola in tre fasi principali: **Rilevamento dei security smells, Classificare i security smell, Priorizzarli e Scelta e implementazione del refactoring**, considerando: Rilevanza del servizio: (alto, medio o basso); Impatto sul sistema: (confidenzialità, integrità, autenticità). Sforzo necessario per il refactoring; Interazioni necessarie con altri team.

Kubernetes

I container offrono un meccanismo leggero per isolare l'ambiente di un'applicazione. Le immagini dei container possono essere eseguite su qualsiasi macchina, garantendo portabilità dall'ambiente. Più carichi di lavoro possono essere eseguiti sulla stessa macchina fisica, migliorando l'utilizzo delle risorse (memoria e CPU).

L'**orchestrazione** dei container può essere paragonata al ruolo di un direttore d'orchestra. Un orchestratore coordina l'esecuzione dei container, assicurandosi che funzionino in modo armonioso e in linea con la visione complessiva del sistema.

Kubernetes (K8s) è un **orchestratore** di container che si occupa di gestire l'intero ciclo di vita dei container, **avviando e spegnendo** le risorse secondo necessità. Se un container si arresta inaspettatamente, K8s **avvia automaticamente** un altro container al suo posto. Se la macchina che esegue un container fallisce, k8s **sposta** automaticamente il container su un'altra macchina disponibile nel cluster.

Se il cluster è composto da più macchine, k8s usa uno scheduler per decidere automaticamente su quale macchina eseguire un container, in modo da ottimizzare l'uso delle risorse (CPU/RAM).

Se più container devono comunicare tra loro, k8s crea un'architettura di rete virtuale in cui ogni container ha un indirizzo IP proprio, da utilizzare per comunicare tra loro.

Principi di Design di K8s: Declarative

In Kubernetes (K8s), il principio fondamentale è la **dichiaratività**, che significa definire lo **stato desiderato del sistema**.

Si definisce la configurazione dell'infrastruttura, come il numero di pod che devono essere in esecuzione, quali container devono essere avviati, quali porte TCP devono essere aperte, e sarà **compito di Kubernetes** fare in modo che l'infrastruttura venga mantenuta secondo quel **desiderato stato**.

Per definire lo stato desiderato si usa un file YAML in cui si definisce la configurazione di distribuzione dei container.

Lo stato desiderato del sistema è definito tramite una serie di oggetti (come i pod, i servizi, ecc.). Ogni oggetto ha due componenti: Le **specifiche** che descrivono **come** deve essere l'oggetto (quante repliche devono esserci, ecc.) e lo **status**, che riflette lo **stato attuale** dell'oggetto (se il numero di repliche in esecuzione è corretto).

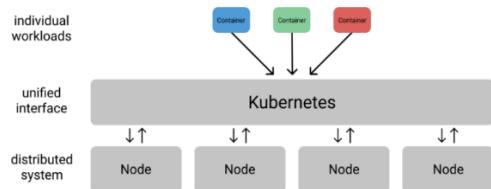
Kubernetes monitora costantemente lo stato degli oggetti. Se rileva che lo stato effettivo non corrisponde a quello desiderato, **interviene** automaticamente per correggere la situazione.

Se un oggetto non risponde correttamente, Kubernetes avvia una nuova versione dell'oggetto per sostituirlo. Se lo stato di un oggetto si discosta dalla specifica definita, Kubernetes invia i comandi necessari per riportarlo allo stato desiderato.

Principi di Design di K8s: Distribuzione

Kubernetes (K8s) offre un'interfaccia unificata per interagire con un cluster di macchine, eliminando la necessità di comunicare direttamente con ciascuna di esse.

Gestisce le comunicazioni tra i nodi, consentendo agli utenti di concentrarsi sulla gestione delle applicazioni senza doversi occupare dei dettagli delle singole macchine.



Principi di Design di K8s: Decoupling

K8s supporta il principio del **decoupling**, che prevede la separazione dei microservizi, ciascuno sviluppato con una singola responsabilità di business. Questo approccio li rende indipendenti, consentendo a ogni servizio di operare in modo autonomo e di essere scalato individualmente rispetto agli altri.

K8s facilita questo approccio grazie alla sua capacità di scalare i servizi **separatamente**, aggiornare un servizio **senza influire** sugli altri e monitorare e orchestrare l'intero sistema.

Principi di Design di K8s: Infrastruttura Immutabile

In un'infrastruttura **immutable**, se è necessario apportare modifiche ad un container, non si modifica il container esistente, ma si crea una nuova immagine del container con le modifiche necessarie. Una volta costruita la nuova immagine, questa viene distribuita come una nuova versione. La distribuzione della nuova immagine avviene senza interrompere il servizio tramite il **rolling update**, in cui i container della vecchia versione vengono gradualmente sostituiti con quelli della nuova.

Se la nuova immagine presenta problemi, Kubernetes ti consente di eseguire un **rollback** rapido, semplicemente aggiornando la configurazione per utilizzare una versione precedente dell'immagine del container.

Oggetti di K8s

Gli oggetti di Kubernetes (K8s) possono essere definiti in file **manifest**. Un **template** in un file **manifest** è una **definizione** in formato YAML che descrive lo **stato desiderato** per un oggetto, in cui è possibile specificare la **configurazione** dell'oggetto, come nome, numero di repliche, container da eseguire e altre risorse.

Esempio di un manifest YAML per un Pod:

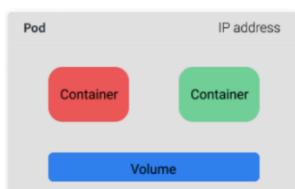
```

apiVersion: v1      // Specifica la versione dell'API di Kubernetes
kind: Pod          // Indica il tipo di oggetto desiderato, in questo caso un Pod.
metadata:          // Contiene metadati che identificano il Pod, come il suo nome
  name: esempio-pod
spec:              // Descrive la configurazione del Pod e lo stato desiderato
  containers:      // Elenco dei container che devono essere eseguiti nel Pod
    - name: nginx-container // Nome del container
      image: nginx:latest   // Immagine del container
      ports:
        - containerPort: 80 // Porta che espone il container
  
```

Oggetti di K8s: Pod

Un **Pod** è l'unità più piccola di esecuzione in k8s ed è composta da uno o più **container** strettamente correlati.

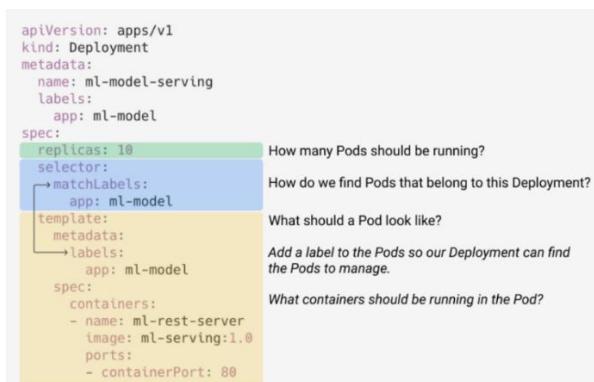
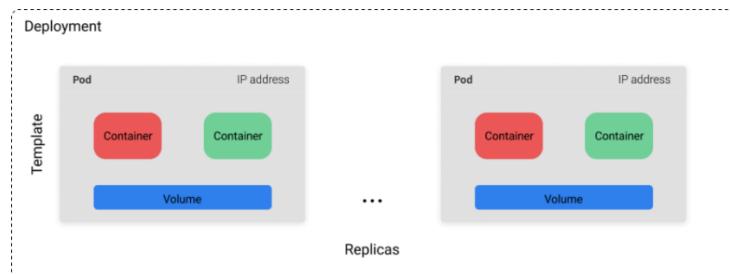
I container all'interno di un pod possono accedere agli stessi volumi, permettendo loro di condividere dati in modo persistente. Tutti i container all'interno di un pod condividono lo stesso indirizzo IP e lo stesso spazio di porta, facilitando la comunicazione tra di loro.



Oggetti di K8s: Deployment

Un **Deployment** in K8s definisce una **collezione di Pod** basata su un **template** di pod e sul numero di **repliche** di quel Pod da eseguire.

Il **Deployment** ha lo scopo di mantenere il numero desiderato di pod sempre in esecuzione. Se un pod smette di funzionare, Kubernetes automaticamente ne avvierà uno nuovo su una macchina diversa per mantenere il numero di repliche definito.

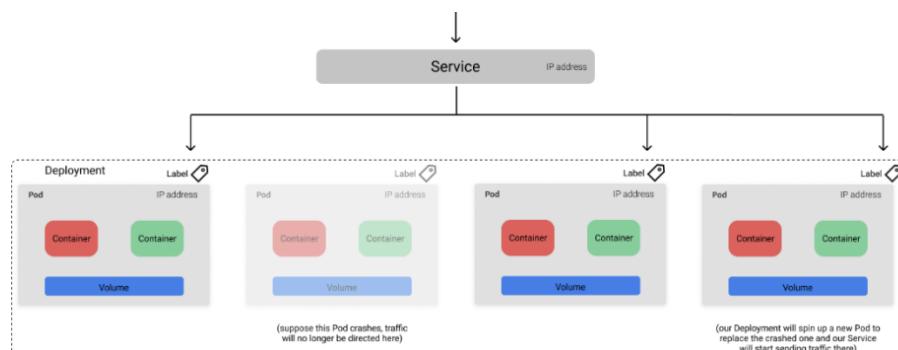


Oggetti di K8s: Service

Ogni **Pod** possiede un **indirizzo IP unico**, che può essere utilizzato per comunicare con esso. Tuttavia, se il set di Pod che eseguono una specifica applicazione cambia (a causa di scalabilità o guasti), la comunicazione diretta con i singoli Pod diventa difficile, poiché gli indirizzi IP potrebbero cambiare. Per risolvere questo problema, K8s offre un **Service**.

Un **Service** in k8s fornisce un **endpoint stabile** attraverso il quale è possibile indirizzare il traffico verso i Pod desiderati, anche se il set di Pod sottostante cambia nel tempo.

I **labels** sono coppie chiave-valore definite nei metadati del Pod. I label permettono al Service di individuare quali Pod devono ricevere il traffico, senza doversi preoccupare degli indirizzi IP specifici dei Pod, che potrebbero variare.



```

apiVersion: v1
kind: Service
metadata:
  name: ml-model-svc
  labels:
    app: ml-model
spec:
  type: ClusterIP
  selector:
    app: ml-model
  ports:
    - protocol: TCP
      port: 80

```

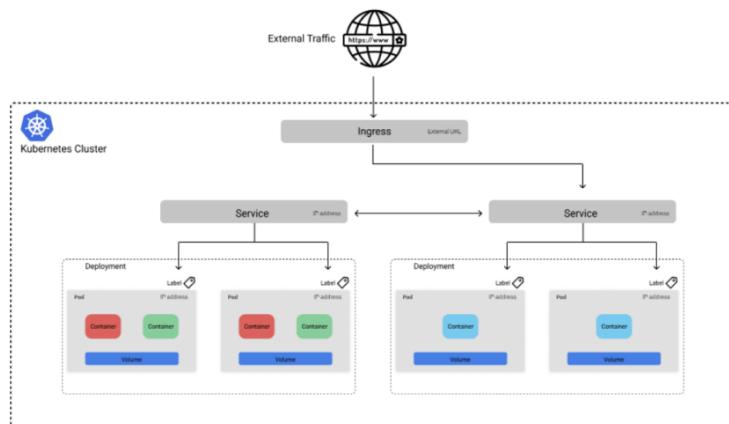
How do we want to expose our endpoint?
How do we find Pods to direct traffic to?
How will clients talk to our Service?

Oggetti di K8s: Ingress

Un Service in Kubernetes consente di esporre un'applicazione dietro un endpoint stabile, ma questo è **disponibile solo per il traffico interno al cluster**. Un **Ingress** è un oggetto che permette di gestire il **traffico esterno** che arriva al cluster e di **instradarlo** ai **Service** specifici all'interno del cluster.

Utilizzando l'Ingress, è possibile specificare quali Services devono essere resi accessibili dall'esterno e definire verso quali service indirizzare il traffico diretto ad uno specifico url.

Esempio. Ingress configurato in modo che il traffico diretto a `http://example.com/app` venga indirizzato al Service1, mentre il traffico verso `http://example.com/api` venga instradato a Service2.



```

apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ml-product-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - http:
        paths:
          - path: /app
            backend:
              serviceName: user-interface-svc
              servicePort: 80

```

Configure options for the Ingress controller.
How should external traffic access the service?
What Service should we direct traffic to?

K8s Control Plane

Il **Control Plane** in Kubernetes è l'**insieme dei componenti** responsabili della gestione e del **controllo** dell'intero cluster. Il Control Plane coordina il cluster, prende decisioni e gestisce il comportamento generale delle applicazioni. Le sue componenti principali sono il kube api server, il database etcd, il kube scheduler e il kube controller manager.

In un cluster Kubernetes, ci sono due tipi di **nodi**:

Master Node: È la macchina dove sono presenti la maggior parte dei componenti del control plane. Si occupa del coordinamento dell'intero cluster. Sebbene spesso sia una macchina singola, può essere configurato in modalità altamente disponibile distribuendo i suoi componenti su più nodi.

Worker Node: È la macchina che ospita i pod che eseguono i container. Si occupa dell'effettiva esecuzione delle **applicazioni**. Ogni worker node ha un componente chiamato **kubelet** che interagisce con il master node per assicurarsi che i pod siano eseguiti correttamente.

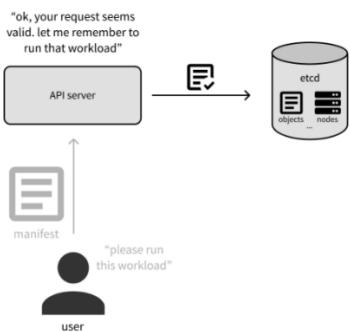
Control plane: API Server

Il **master node** di Kubernetes gestisce l'intero cluster attraverso vari componenti, tra cui l'**API Server**, che è il cuore del Control Plane e il punto di ingresso per tutte le interazioni con il cluster.

L'**API Server** riceve e gestisce le richieste provenienti da utenti o processi, come l'invio di manifest contenenti nuove specifiche di oggetti (Pod o Deployment).

Quando una richiesta viene inviata all'API Server, questa viene validata e processata. L'API Server fornisce inoltre un'interfaccia centralizzata per consultare lo stato del cluster, gestendo tutte le interrogazioni relative a configurazioni, oggetti, nodi e assegnazioni.

Lo **stato del cluster** è memorizzato in **etcd**, un database distribuito key-value che garantisce la persistenza e la sincronizzazione delle informazioni tra i vari componenti. Questo sistema assicura che il cluster mantenga uno stato coerente e affidabile.



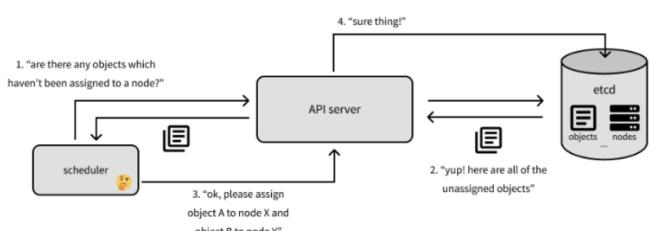
Control Plane: Scheduler

Lo **scheduler** nel **master node** è responsabile di decidere su quale macchina (nodo) devono essere eseguiti gli oggetti (Pod).

Lo **scheduler** interroga l'**API server** per ottenere la lista degli oggetti (Pod) che non sono ancora stati assegnati a nessun nodo del cluster.

Lo **scheduler** esamina i nodi disponibili nel cluster (in base a CPU/RAM e ai requisiti dell'oggetto) e decide quale nodo è più adatto per eseguire l'oggetto.

Una volta determinato il nodo giusto, lo **scheduler** comunica con l'**API server** per aggiornare lo stato e assegnare l'oggetto al nodo selezionato. L'API server riflette questa assegnazione nel db etcd.

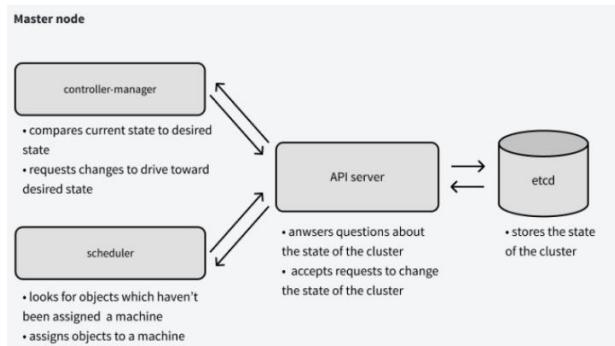


Control Plane: Controller Manager

Il **controller-manager** nel **master node** è il componente che **monitors** continuamente lo stato del cluster.

Il controller-manager osserva costantemente lo stato del cluster tramite l'**API server**. Questo stato comprende la configurazione dei nodi, dei pod, dei deployment e altri oggetti K8s.

Il controller-manager confronta lo **stato attuale** del cluster con lo **stato desiderato**. Se lo stato attuale non corrisponde allo stato desiderato, il **controller-manager** interviene inviando opportuni comandi all'**API server** per modificare lo stato del cluster e avvicinarsi a quello desiderato. Ad esempio, potrebbe ordinare al cluster di creare nuovi pod, riavviare quelli esistenti o scalare i servizi.



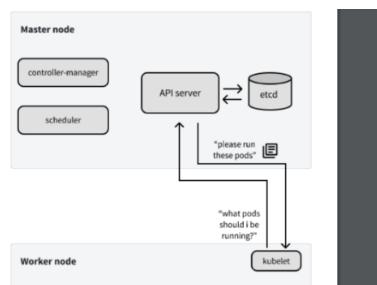
Worker Node: Kubelet

Il **kubelet** è un componente fondamentale che opera su ogni **worker node** di K8s

Il **kubelet** agisce come un "agente" del nodo, comunicando con l'**API server** per ricevere informazioni su quali **carichi di lavoro** (workload) devono essere eseguiti su quella macchina.

Quando il kubelet riceve l'ordine di eseguire un **pod**, è responsabile di **avviare** il pod sul nodo, avviando i container necessari all'interno di quel pod.

Quando un nodo si unisce per la prima volta al cluster, il kubelet **annuncia la sua presenza** all'API server. Questo permette al **scheduler** di sapere che il nodo è disponibile e quindi di iniziare a assegnare i pod a quel nodo.



Worker Node: Kube-proxy

Il **kube-proxy** è un componente eseguito su ogni worker node, responsabile di facilitare la comunicazione tra i container distribuiti tra i vari nodi del cluster. Configura le regole di rete, come le **iptables** o altre tecnologie di rete, per garantire che il traffico venga instradato correttamente tra i Pod e i servizi, indipendentemente dalla loro posizione nel cluster, in modo da assicurare uno scambio di dati fluido tra i container, anche se distribuiti su nodi diversi.



Quando NON usare Kubernetes

Kubernetes potrebbe non essere la scelta giusta, portando ad overhead di gestione e a rendere complessa in modo superfluo l'infrastruttura, nei casi in cui puoi eseguire il tuo carico di lavoro su una sola macchina, le tue necessità di calcolo sono leggere, non hai bisogno di alta disponibilità e puoi tollerare periodi di inattività, non prevedi di fare molte modifiche ai tuoi servizi distribuiti, hai un'applicazione monolitica e non hai intenzione di migrarla verso un'architettura a microservizi.

Docker Swarm vs Kubernetes

Docker Swarm: Ha un processo di installazione più semplice e veloce rispetto a K8s. La configurazione e l'uso di Docker Swarm sono più semplici e intuitivi. È preferito in situazioni dove è richiesta semplicità e sviluppo rapido di applicazioni.

Kubernetes: Kubernetes supporta in modo avanzato il bilanciamento del carico e l'auto-scaling delle applicazioni. Kubernetes è progettato per garantire alta disponibilità e resilienza anche in caso di guasti. Kubernetes ha una grande community di sviluppatori. Kubernetes è ideale per gestire grandi cluster e applicazioni complesse.

Quantum Software Engineering 101

Quantum Computing

Il **Quantum Computing** sfrutta le proprietà della meccanica quantistica per eseguire calcoli che non sono possibili con i computer classici.

Un **qubit** è l'unità base di informazione nei computer quantistici. A differenza dei bit tradizionali, che possono essere solo **0 o 1**, un qubit può essere una **combinazione lineare** di entrambi gli stati.

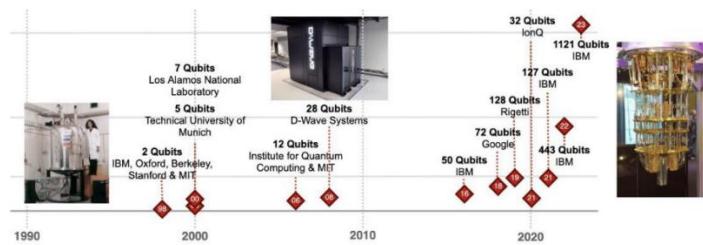
In termini matematici, un qubit è rappresentato come una combinazione complessa dei due stati possibili: $\alpha|0\rangle + \beta|1\rangle$, dove α e β sono **numeri complessi** che rappresentano la **probabilità relativa di trovarsi nello stato 0 o nello stato 1**.

La **superposizione** è una delle caratteristiche distintive dei qubit. Un qubit può esistere in una **combinazione** di entrambi gli stati, 0 e 1, contemporaneamente.

Quando viene misurato, il qubit **"collassa"** in uno degli stati, **0 o 1**, con **probabilità determinate dai coefficienti α e β** .

I qubit possono essere **entangled** (intrecciati), il che significa che lo stato di un qubit può essere direttamente correlato allo stato di un altro qubit. Se due qubit sono entangled, conoscere lo stato di uno dei qubit ti dice immediatamente lo stato dell'altro, anche se sono separati fisicamente da distanze molto grandi.

Significant advancements in Quantum Computing technology



Quantum computers start to become widely available

Limiti della Computazione Quantistica (QC)

Affidabilità: I qubit sono molto sensibili all'ambiente esterno (rumore, vibrazioni, temperatura), il che porta a **errori** durante i calcoli.

Scalabilità: I computer quantistici moderni hanno un numero relativamente **limitato di qubit**. Per risolvere problemi complessi, sarebbe necessario avere milioni di qubit, ma la tecnologia attuale non è ancora in grado di supportare questa scalabilità.

Costo: La costruzione e la manutenzione di computer quantistici sono **costose**. I sistemi attuali richiedono ambienti controllati, come temperature molto basse.

Limiti nello sviluppo del software quantistico:

Attualmente, lo sviluppo di software quantistici si concentra principalmente sulla **combinazione di porte logiche a basso livello** per costruire circuiti quantistici, ma manca ancora una **programmazione più astratta e user-friendly**.

C'è una **limitata interoperabilità** tra i vari software quantistici e le macchine quantistiche. Ogni piattaforma e sistema quantistico può avere il **proprio linguaggio** e le proprie librerie, rendendo difficile l'integrazione.

Gli strumenti di sviluppo per il software quantistico **non supportano** ancora pienamente **l'intero ciclo di vita** di un'applicazione, come la progettazione, il test, il debug e il monitoraggio, come accade per il software tradizionale.

Quantum Software Engineering

Il **Quantum Software Engineering (QSE)** si basa sull'applicazione di principi di ingegneria del sw con l'obiettivo di ottenere software quantistico affidabile ed efficiente per i computer quantistici.

La dichiarazione di Zhao (2020) e il **Talavera Manifesto** indicano le linee guida fondamentali per lo sviluppo del software quantistico.

Indipendenza dalle tecnologie: Il QSE dovrebbe essere **agnostic**, ossia non legarsi a un particolare linguaggio o tecnologia, favorendo la flessibilità e l'adattabilità.

Coesistenza tra computazione classica e quantistica: Il software quantistico dovrebbe essere progettato per **coesistere** con sistemi classici.

Gestione dello sviluppo software quantistico: Il QSE dovrebbe supportare una **gestione efficace** dei progetti di sviluppo di sw quantistico, che includa la pianificazione, il monitoraggio, il controllo della qualità e la documentazione.

Evoluzione del software quantistico: Il sw dovrebbe essere progettato tenendo conto della **continua evoluzione** della tecnologia quantistica, prevedendo aggiornamenti.

Software con zero difetti: QSE dovrebbe puntare a **fornire programmi quantistici** che abbiano un numero minimo di errori, cercando di raggiungere la perfezione sw.

Assicurazione della qualità: La qualità del software quantistico deve essere **garantita** attraverso metodi sistematici di testing, validazione e verifica.

Promozione del riuso del software: Dovrebbe incoraggiare il **riutilizzo** del software, facilitando la creazione di componenti modulari e riutilizzabili.

Sicurezza e privacy: QSE deve **affrontare la sicurezza e la privacy** fin dalla fase di progettazione, integrando misure di protezione dei dati sensibili.

Governance e gestione del software: QSE dovrebbe garantire che le pratiche di sviluppo e le risorse siano gestite correttamente per allinearsi agli obiettivi aziendali e normativi.

Ecco alcuni esempi pratici di come la QSE possa essere applicata:

Miglioramento dei traspiler: I **traspiler** sono compilatori che traducono i programmi scritti in linguaggi di programmazione quantistica (come Qiskit o Quipper) in un formato comprensibile dal computer quantistico. QSE lavora per **migliorare** i traspiler, rendendo più efficiente e accurata questa traduzione, in modo da sfruttare al massimo le capacità del computer quantistico e ridurre gli errori.

Abilitare la riusabilità del codice: Nella **programmazione quantistica**, la creazione di codice riutilizzabile è essenziale per migliorare l'efficienza e ridurre i tempi di sviluppo. La **QSE** promuove lo sviluppo di **moduli e librerie** quantistici riutilizzabili, che possono essere facilmente adattati a diversi contesti o applicazioni.

Abilitare la verifica del programma: Poiché i computer quantistici sono soggetti a **errori probabilistici**, è fondamentale avere metodi di verifica per **assicurarsi che i programmi quantistici funzionino correttamente**. La **QSE** si concentra su adottare strumenti di **debugging, simulazioni** e test che permettano di prevedere il comportamento dei programmi su hw reale, anche se quest'ultimo è imperfetto.

Abilitare la coesistenza di computazione classica e quantistica: La computazione **classica** e quella **quantistica** devono lavorare insieme, poiché i computer quantistici non sono ancora in grado di gestire tutte le operazioni che i sistemi classici gestiscono quotidianamente. La **QSE** permette di eseguire parte del calcolo sui sistemi classici e parte su quelli quantistici, sviluppando **interfacce** che permettono ai programmi classici di inviare e ricevere dati da sistemi quantistici.

Sfruttare meglio i computer quantistici eterogenei e inaffidabili: I **computer quantistici** attuali non hanno una **perfetta affidabilità** e non tutti i computer quantistici sono uguali. La **QSE** si concentra nel **gestire e ottimizzare** l'uso di diverse architetture quantistiche. La QSE cerca di sviluppare algoritmi in grado di **compensare** i guasti e gli errori tipici dei computer quantistici, migliorando la **robustezza** in ambienti eterogenei.