Contents

1 Modulo I - Machine Learning for Software Engineering (Falessi)

2 Ripasso: introduzione a SVN e Github

Un sistema di controllo delle versioni tiene traccia dei cambiamenti ad un file ed inoltre permette di tornare indietro nelle versioni. Ci sono due tipologie:

- Centralizzata: SVN, utenti condividono una repository che è su un solo server centralizzato
- Distribuito: Git, ognuno ha una copia della repo.

Differenze con classici sistemi di storage cloud sono varie:

- SVN e Git sincronizzano solo se c'è richiesta, mentre per sistemi cloud avviene in automatico
- I merge sono a grana fine per SVN e Git, per Dropbox/Drive è a grana più spessa.
- La storia delle versioni è mantenuta da SVN e Git, mentre poterebbe non esserlo per Dropbox/Drive

Working copy: versione su cui è possibile lavorare in locale. Non si lavora mai sulla risorsa condivisa, bensì su quella locale.
Revisione: particolare stato della risorsa condivisa, su

cui è possibile tornare indietro. La versione è spesso una revisione particolare, che può essere offerta agli utenti (es versione 1.0 può corrispondere alla revisione 150).

2.1 Issue tracking systems

Sistema che permette di creare, assegnare e tenere traccia dei problemi (issues). Tutto nasce da Bugzilla (per progetti open source), un bug in un codice è molto simile a descrivere un requisito: nasce il concetto di ticket. Un ticket è un informazione rilevante al progetto, come un requisito da implementare o un bug.
Molti sistemi sono gestiti attraverso i ticket (medie-piccole dimensioni), ogni ticket ha un workflow: creato, assegnato, sviluppato, testato, approvato, chiuso. Tra gli esempi di sistemi per issue tracking ci sono Jira, Github, Redmine.

3 Tecniche di Machine Learning ed analisi software a supporto della quality assurance e del testing

I bug software costano circa 2.84$ dollari ogni anno. Il codice viene scritto in diversi linguaggi, da tantissime persone, per poter fixare bug, aggiungere nuove feature e migliorare la qualità del codice. Il software è rilasciato con grande velocità, si vuole prevenire di avere bug in modo da avere technical debt basso.
Failure: comportamento osservato dall'utente e che non corrisponde alle specifiche del sistema. Un difetto software è quella parte di codice che può dare luogo ad una

failure, questo non avviene sempre, ma solo sotto determinate condizioni. Per evitare le failure si cerca di individuare i bug prima che questi possano essere eseguiti dagli utenti.
È importante capire, avendo tempo limitato, come poter priorizzare le risorse di analisi. Si parla di software analytics come analisi di dati che riguardano progetti software.
Un aspetto importante è ML per poter predire ed evitare i bug futuri. In generale:

- Misuro i dati
- Analizzo i dati e creo il modello di ML
- Identifico quale classe/metodo è buggy

I dati sono gestiti da i version control systems ed issue tracking systems.
Importanti i commenti dei commit: invece di descrivere la modifica effettuata, faccio riferimento al ticket sul sistema. Dovrebbe esserci una relazione 1-a-1 tra ticket e commit: prendo il ticket e lo sviluppo per intero, in modo poi da fare il commit delle mie modifiche.
Se ho il tracciamento preciso tra ticket e codice che ho implementato per quel ticket, posso vedere se l'ammontare di linee di codice richieste per realizzare il ticket è maggiore o minore di quelle richieste per rimuovere il bug. Posso differenziare i ticket in base a bug e requisito e vedere il numero di righe cambiate.
Una volta estratti i dati, posso collezionare molte metriche a riguardo:

- Quante righe di codice modificate

- Chi le ha modificate

3.1 Metriche per individuare bug

Posso usare metriche per fare delle stime su quale classe è buggy:

- relative al codice: classe con molte righe è potenzialmente buggy
- processi: se file è stato affetto da molti cambiamenti
- fattori umani: se file è stato toccato da sviluppatori esperti o non.

Posso leggere tutte queste info tal ticket di JIRA, ogni commit avrà un suo identificativo.
Si cerca di andare a capire quali file di una release sono defective o non defective. Avrò quindi dei dati che darò in pasto a modelli di machine learning black-box, con cui potrò capire dato un nuovo file quanto questo sarà defective. È anche importante capire perché un file è difettoso, ci sono anche regole di normativa per cui se si usa una predizione bisogna anche fornire il perché della predizione. È possibile anche vedere il perché una classe ha avuto una certa probabilità di essere difettoso.

3.2 Process control chart

Chart che mostra la stabilità degli eventi nel tempo: è importante che sull'asse verticale ci sia l'elemento di cui si vuole controllare la stabilità, mentre sull'asse orizzontale ci deve essere qualcosa che possa essere misurato.
Voglio vedere se un certo progetto ha dei difetti per in

base al numero di revisione: metto su asse x il tempo (es settimane, o le revisioni), asse y avrò i difetti. Mi chiedo se in un progetto i difetti sono lineari in base alle revisioni. Dobbiamo:

- Selezionare i dati sull'asse y ad esempio il numero di commits o requirements
- Seleziono dati per asse x, asse temporale. Può essere giorni/mesi o anche release
- Colleziono i dati
- Calcolo media e varianza, imposto poi degli assi di limiti superiori ed inferiori (ad esempio):
  - limite superiore (upper control limit): media + 3·deviazione standard
  - limite inferiore (lower control limit): media - 3·deviazione standard. Può andare sotto 0, alle volte conviene andarci ma altre volte no (quindi lo si fa fermare a 0).

Visualizzando il chart, posso capire se qualcosa è andata particolarmente male: quando un punto è al di fuori del limite, è significativamente diverso da tutti gli altri punti. Quindi in questo caso si analizza il perché per una determinata release c'erano dei valori così estremi, ad esempio perché c'è stato un numero così elevato di difetti.

3.3 Ripasso: continous integration e travis

Pratica per cui i membri del team integrano il loro lavoro continuamente. Ogni integrazione è verificata con un build del progetto, viene fatto per evitare errori nell'integrazione

del progetto.
È molto importante che il built sia automatico, in modo che chiunque possa effettuarlo, e non solo alcuni membri del team. Importante anche che la built sia self-testing, e che inoltre i commit avvengano quotidianamente o almeno per ogni feature.
I test vanno effettuati in un ambiente che sia il più possibile simile all'ambiente di production: ad esempio, se ho sviluppato un app, il production environment sarà il dispositivo dei miei utenti.

3.4 Ripasso: technical debt

Applicazione di concetti finanziari al dominio del software engineering. Code smell e regole di qualità: una regola di qualità è un principio che detta come il codice dovrebbe essere, ad esempio:

- alta densità di commenti
- bassa densità di codice
- if statement non difficile da leggere

Una violazione o code smell è una porzione di codice non perfetto. È importantissimo soffermarsi sulla differenza fra difetto e code smell: difetto può dare vita ad un failure (sotto determinate condizioni), mentre il code smell è qualcosa non osservabile dall'utente (ad esempio la lunghezza di un if statement). Il problema è che i code smell impattano gli sviluppi futuri.
Il debito tecnico può essere visto come un qualcosa che darà problemi a sviluppi futuri (impatta la prossima release), può emergere organicamente come molti sistemi

aumentando di complessità, aumentano la loro complessità di gestione. Inoltre, può anche essere scelto in maniera opportunistica, ovvero decidere deliberatamente di avere una certa quantità di debito tecnico.
Il technical debt consiste in due parti:

- principio: costo per eliminare i code smells
- interesse: la penalità da pagare in futuro per il debito non eliminato

Come gestisco i code smell attraverso i ticket: tutto ciò che bisogna fare deve essere scritto in un ticket. Non c'è altro mezzo di informazione che il progetto può usare, linee guida e processi aziendali possono essere documentati su un altra piattaforma (confluence, per completare JIRA).Se c'è un code smell, **dipende dal processo aziendale**: può essercene uno che dice che chi ha introdotto lo smell lo toglie senza ticket, un altro che dice che i problemi rilevati da Sonar Cloud verranno analizzati a parte da specifiche persone che avranno il ruolo di aprire i ticket appositi. È possibile impostare i repository in modo che rigetti i commit che violano alcune regole di qualità (configurando Travis/Jenkins), così saprò che il codice è sempre smell free.

3.5 Misure di analisi

Voglio rispondere alla domanda: le linee di codice per le feature, sono state di più o di meno di quelle usate per risolvere i bug? Quando gestisco i dati, posso farlo per due motivi:

- guardare al passato: perché spendo molte LOC per risolvere i bug? Vado a fare delle analisi introspettive sul progetto
- avvento del ML, predico il futuro: prendo delle decisioni nel presente, in modo che l'impatto dei cambiamenti futuri sia controllato

Quando si fa una predizione: un conto è farla binaria, un altro è se vanno predetti dei numeri. Quest'ultima è estremamente più complessa di quella binaria.
esempio: supponiamo di dover predirre il meteo, a seconda del fatto che io dica che piove o no, o che piove 20 cl. Non ho alcune informazioni, ad esempio se piove più o meno di 20 cl o in media 20 cl. Potrei avere predizioni diverse pee giorni diversi. Bisogna fornire **un grado di confidenza** (e quindi intervallo di confidenza), in modo da rendere la previsione più comprensibile.

4 Merging JIRA con Git

4.1 Git: comandi utili

Una volta effettuato il clone di una repository, è possibile avere accesso al log: **git log <options>**, da accesso ai cambiamenti mostrando

- autore del commit
- cambiamento
- commento

se non viene specificato il branch, git ritorna il log di quello corrente. Una cosa interessante può essere comparare due commit fra loro, per vedere i cambiamenti:

tramite il comando **git diff <commit_id>..<altro commit_id>**. Alcune visualizzazioni sono human readable, ma nel codice che fa l'analisi automatica non è importante.
Data la revisione di un file, per ogni riga è possibile risalire al commit che ha inserito quella determinata riga: ho un file di 10 righe, mi chiedo chi ha inserito una riga, quando e perché. **git blame <filename>** (con -w ignora gli spazi bianchi). Fornisce la data dell'ultima modifica con la relativa modifica (ci sono anche meccanismi per andare indietro) e chi l'ha effettuata. È possibile fare il grep sul testo, in modo da cercare solo specifiche parti di interesse del log.

4.2 Cercare ticket in JIRA

Una volta trovato un progetto, è possibile selezionare il tipo di ticket da cercare, lo sviluppatore etc... (numerosi filtri), in advanced è anche possibile creare delle query SQL-like, con cui è possibile creare delle ricerche sofisticate e metterle da parte.

5 Modulo II - Software Testing (De Angelis)

6 Introduzione e concetti generali per software testing

La progettazione di un sistema software è complessa: in primo luogo per l'interazione fra le persone che hanno un interesse nello sviluppo del sistema (ingegneri, programmatori, analisti, committenti etc...). È importante cercare di avere degli strumenti che guidino nel processo di sviluppo del software in modo da poter definire delle garanzie che tutti i membri del team siano sulla strada giusta.
È necessario stabilire dei checkpoint per poter dire che il processo di progettazione e sviluppo sta procedendo nel modo corretto: Boehm definisce e distingue i due concetti di verifica e validazione:

- le attività di verifica servono per poter verificare che stiamo facendo le cose nel modo giusto, cerchiamo di rispondere alla domanda: stiamo costruendo bene il prodotto? Dove per bene si intende in maniera conforme alle sue specifiche e con un certo livello di qualità
- le attività di validazione servono per rispondere alla domanda: stiamo costruendo il prodotto giusto? In questo caso, l'enfasi è sul fatto che il prodotto sia o meno conforme a quanto richiesto dal committente, quindi si strutturano una serie di attività con riferimento alle richieste di quest'ultimo.

I due obiettivi ci impongono due modi diversi di re-

lazione con il prodotto software, potrebbero anche entrare in conflitto fra loro ed in tal caso occorrerà risolvere o quanto meno mitigare tale conflitto.
In un contesto più generale, l'obiettivo è rassicurare l'utente che il sistema è adatto allo scopo per cui è stato progettato.
Chiamiamo le operazioni di verifica e validazione più brevemente $V\&V$: con tali attività, vogliamo tranquillizzare gli attori coinvolti nel processo di design e sviluppo del software, mostrando che ciò che viene fatto è conforme alle specifiche.
Le tecniche di $V\&V$ tendono a dare un'indicazione di confidenza sulla bontà del prodotto software.
In generale, il livello di fiducia è un concetto non assoluto, in quanto dipende da vari fattori:

- è funzione del software nell'organizzazione
- è funzione delle attese dell'utente
- è funzione delle politiche di mercato
- etc...

Il peso attribuito alle operazioni di verifica o di validazione dipende dal contesto e dalle fasi in cui ci si trova, il punto è capire in base a cosa frasi guidare per arrivare al livello di qualità del software desiderato, a quel punto si sceglie se porre l'enfasi sulla verifica o sulla validazione.

Definizione: sia S una specifica, ovvero una funzione che prende elementi da un certo dominio D e li mappa su un certo codominio C. Un programma P è una particolare implementazione di S, che lega gli elementi di D e di C.

Il programma P è corretto $\leftrightarrow \forall d \in D, P(d) = S(d)$.
Se S è una specifica data ad uno dei partecipanti del progetto per essere realizzata, andare a vedere se P è corretto è un'operazione di verifica, mentre se S si basa sule necessità dell'utente, allora è una validazione.
Affermare che P sia corretto equivale a dire che P soddisfa le specifiche, ma per dire che è corretto bisognerebbe in pratica andare a testare tutti i valori $d \in D$ e mostrare che $P(d) = S(d)$ e tale operazione può essere infinita (se ad esempio $D = \mathbb{R}$.
In generale, questo problema è difficile, gli approcci possono essere due: intanto si distingue la specifica fra una specifica di progetto ed una utente e poi come risolvere l'uguaglianza $\forall d$

- 1° metodo: verifica formale. Si dimostra formalmente, con la logica-matematica, che S effettivamente rispetta la specifica per ogni possibile input.
- 2° metodo: software testing e debugging. Si cerca di scoprire se P nasconde delle imperfezioni.

Ci focalizzeremo sul 2° metodo (anche perché chi cazzo è capace a fare una verifica formale, figuriamoci dimostrare formalmente che sia valida); il 1° metodo ha innumerevoli benefici, quindi quando è possibile bisogna validare formalmente.
Nel 2° approccio, si ammette di avere una discrepanza fra P ed S, la si accetta e si cerca di trovare i punti in cui sussiste tale discrepanza, andando a cercare le porzioni di dominio critiche per trovare tali punti.

6.1 Software testing vs debugging

Per raggiungere una adeguata confidenza sulla presenza o meno di mismatch fra P ed S si può:

- pianificare una strategia che faccia vedere potenziali differenze fra P ed S. Questo è ciò che si fa col software testing (ci focalizzeremo su questo)
- cercare di capire il perché abbiamo delle difformità e come possiamo rimuoverle. Questo è ciò che si fa con il debugging

Nel software testing, l'obiettivo principale è far saltare fuori le differenze possibili fra P ed S, quindi bisogna individuare le parti su cui focalizzarsi per trovare queste differenze. Se si scopre anche il motivo dell'errore è un plus, ma non è l'obiettivo principale. Invece, nel debugging sappiamo di avere un errore e vogliamo scoprire il perché di tale errore, in modo da poterlo risolvere.
Dire che P deve essere corretto rispetto ad S, vuol dire che P rispetta la formula introdotta precedentemente, ovvero P è privo di errori. Ma P privo di errori intende molteplici cose, più formalmente possiamo definirlo facendo la distinzione fra errore, difetto e malfunzionamento:

1. un errore è ciò che genera un difetto
2. un sistema software è pieno di difetti (o fault o bug) ma non è detto che un utilizzatore del software incappi in tutti. Quando questo accade, il fault genera un malfunzionamento (o failure), che viene percepito dall'utente.

3. un malfunzionamento è il risaluto di un fault

Possiamo cercare di evitare i malfunzionamenti pure mantenendo il sistema buggato, ad esempio isolando il bug ovvero facendo in modo che nessun utente vi si imbatta; è possibile organizzare una campagna di $V\&V$ per nascondere tale bug.
Sempre nella sfera del $V\&V$, vorremo cercare di omettere il fault: per afre ciò, applichiamo delle politiche di fault detection o removal, e questa è la parte più grossa del software testing: si pianificano attività il cui scopo è quello di esporre i fault.
Molto più complessa è l'attività di errore removal, in quanto occorre capire perché il progettista/analista/sviluppatore ha avuto una percezione non corretta del software che andava analizzato e sviluppato.
La parte che riguarda l'analisi e rimozione di errori e fault fa parte del debugging, mentre la ricerca di failure del software testing.
esempio:
il metodo *makeItDOuble* mi fa intendere che il risultato sia il doppio del parametro che ho passato come input, ma se passo 3 ottengo 9. Il risultato 9 è la failure, causata dal fault che è il prodotto del parametro con se stesso. Probabilmente, tale fault è dovuta ad un errore umano, come ad esempio un typo di "*" al posto di "+".
Ma ad esempio, se passo 2 come input, non mi accorgo mai del problema e se il codice che utilizzo passa sempre 2, l'utente non sperimenterà mai la failure perché il bug non sussisterà.
Il software testing è composto da attività che vengono

eseguite in un ambiente controllato e sul sistema software così come verrà usato dagli utenti finali, mentre nelle verifiche formali si ragiona su un modello astratto dall'originale.
Il lavoro sul sistema vero e proprio ha come vantaggio il fatto che si prova se il software originale risponde alle esigenze per cui è stato creato, ma come svantaggio il fatto che il test non può dimostrare l'assenza di errori, bensì solo la presenza di essi.
Il mood del test engineer è starno: devi essere contento se trovi errori, non se non li trovi (cit*).

7 Modulo III: Dott. Calavaro - Enterprise IT

8 Introduzione

IBM: azienda nel mercato da 110, nata per cerare dispositivi per il business. Non punta al mercato di largo consumo, bensì alla commercializzazione con altre aziende che acquistano prodotti hardware o software. Enterprise IT: cos'è l'IT di una grande società e quali sono i requisti di una grande società, ovvero adottati da un cliente di tipo enterprise.
Enterprise: impresa che può essere:

- azienda privata
- ente governativo
- etc...

Obiettivi: partendo da un background di sviluppo e progettazione software, come posso metterlo in esercizio in un ambiente enterpire. Vogliamo capire:

- i tipici Enerprise level per ambienti IT
- quali sono i tipici requisiti non funzionali e l'order of magnitude di essi
- introdurre considerazioni base riguardanti il Total Cost of Acquisition (TCA) o Total Cost of Ownership (TCO)

8.1 Cos'è un Enterprise

Un Enterprise è una grande impresa che potrebbe essere un'azienda, un ente pubblico, etc... in generale in ente sofisticato con certe caratteristiche:

- grande compagnia o organizzazione
- gestisce numero elevato di client
- ha milioni di impiegati
- richiede disponibilità 24x7: se erogo un servizio, questo deve essere sempre attivo (365 giorni l'anno)
- operano in una indistry strettamente regolata: in una industry ci sono molte regole, che permettono alle aziende di cooperare. Industry è il mercato in cui si trova ad operare un'azienda, ma è anche l'insieme di regole di business: per entrare nel mercato esiste un insieme di regole a cui bisogna sottostare, non si può aprire, ad esempio, una banca dal nulla: questa è una industry. Nelle grandi Enterprise esistono delle regole, ad esempio di sicurezza per industry del modo delle compagnie aeree, o anche la replicazione dei server che mantengono i dati dei clienti, in modo da far fronte a disastri naturali etc..., magari distanti tot km, per quello che riguarda un industry del mercato bancario
- fornisce servizi sotto degli SLA restringenti
- deve essere in grado di gestire picchi di carico, spesso anche 1000x il solito

esempio: aziende del mondo tel&co. Lavorare e gestire un numero di clienti così elevato porta benefici all'azienda, ma anche ai clienti, che possono avere i servizi a costi minori perché ammortizzati sul numero totale dei clienti. Una Enterpirse level solution è una soluzione in larga

scala che supporta tali concetti, che richiede la cooperazione di esperti IT.
Nel classico ciclo di vita dell'applicazione, ci si focalizza sul develop del codice, eventualmente pensando al middleware, ad esempio utilizzo di DB, application server (che è un server che permette operazioni di tipo transazionale, per avere operazioni ACID) e SO, o per ambienti di virtualizzazione, containers.
Esistono tutta un'altra serie di aspetti per un'applicazione enterprise:

- data replication, partitioning
- monitoraggio dell'ambiente di esecuzione
- sistemi per gestione di sicurezza
- orchestrazione, load balancing

tutto ciò si trova sopra la parte hardware, che può essere on premise o in cloud.
Organizzare un business di tipo Enterprise conviene anche per piccole aziende: se scalo nel numero di prodotti consegnati, il costo di spedizione etc... è più ammortizzato. Lo stesso vale per applicazioni Enterprise: posso partire col classico server tower, passare poi ad un data center con rack (cambia la dimensione a seconda del server, nomenclatura x-unit).
Nelle applicazioni frontali esistono due tipo di sistemi sempre presenti nel back-end:

- system of engagement
- system of record

coinvolgo un grande numero di sistemi Enterprise per ogni transazione, se qualcuno di questi va giù è un problema.
Sistemi-Z IBM (nell'ambito dei systems of record): i vecchi mainframe, che hanno sempre la caratteristica di poter mandare in esecuzione software datati, ma anche modelli.
Per accedere ai sistemi si usano le classiche API, i sistemi hanno degli use case ben chiari e definiti. esempio: use case di acquisto di un biglietto aereo

1. scelgo la meta, la data, l'aeroporto di partenza e di arrivo etc...
2. l'applicazione gira, mi fornisce la lista di voli possibili, i posti disponibili. Hunder the hood: ho "parlato" con un system of engagement, che ha contattato un system of record, avranno contattato DB con una query specifica per filtrare i risultati etc... Una serie di ingaggi sempre più sofisticati verso il system of record, il cui poi risultato viene sempre mandato al mio system of engagement e sul front end della mia applicazione.

8.2 Requisiti non funzionali Enterprise

I fondamentali sono:

- reliability ad availability
- serviceability
- security
- performances

- scalability

In una Enterprise, contano anche gli ambienti, che sono diversi ed hanno diversi requisiti:

1. sviluppo
2. test
3. pre-production aka QA (Quality Assurance): ambiente copia del production env, per testare che tutto sia ok
4. produzione: ambiente fondamentale, è quello che riceve le query per la mia applicazione etc... Se va giù anche per poco tempo, è complicato
5. high availability: serve per assicurarsi che se un ambiente cade, ce ne sia un altro che ha e stesse informazioni
6. distaster recovery

8.2.1 Reliability ed availability

Spesso reliability = affidabilità, se è al 99% è alta? Manco per nulla, anche 99.9999%, per ad esempio un volo aereo non è alta.
Un affidabilità del 99.99999% (i cinque "9") è considerata il minimo (rivedi appunti SDCC), a seconda del tipo di applicazione, crescerà l'affidabilità.
Se ho due sistemi che hanno un'affidabilità del 99.9%, l'affidabilità del sistema dato dai due che cooperano decresce (è il prodotto). È importante lavorare sull'affidabilità, quando si parla di affidabilità occorre specificarne il livello, non basta dire che è up 24x7.

Molte industry hanno dei requisiti di affabilità dettati dagli SLA e che sono dettati ed imposti a chi sta nell'industry, mentre altre scelgono i loro requisiti privatamente.
Al crescere del numero di sistemi, cresce la probabilità di failure: avere una macchina che ha affidabilità de 99.999% è diverso da averne 1000 che hanno la stessa. Il sistema avrà un costo diverso, ma anche il servizio erogato sarà diverso e quindi bisognerà eseguire l'analisi economica.

8.2.2 Serviceability

Si riferisce all'abilità del supporto tecnico di poter risolvere problemi in real-time, invece di fermare il sistema ho delle copie di backup. L'obiettivo è ridurre il costo operazionale e mantenere la continuità di business.
Ad esempio, la possibilità di poter fare disk swapping, se uno si rompe.

8.2.3 Security

Un altro requisiti non funzionale importante, elemento più critico che può esistere nell'Enterprise. Ormai gli attacchi sono presenti e costanti, bisogna poter gestire la sicurezza dei sistemi e spesso viene dato per scontato. È anche necessario per poter essere compliant a determinati requisiti: GDPR multa del 4% se un data breach non viene denunciato entro 24h.
Un altro requisito importante è l'encryption dei dati:

- sia perché chi arriva e copia i dati li trova cifrati
- sia per il valore dei dati stessi

internamente solo le persone autorizzate possono accedervi. Ma quali dati cifrare e quando cifrarli, inoltre dove vanno salvati i dati cifrati e che si occupa della cifratura? Problemi con cui le aziende hanno a che fare, l'approccio tipico era quello del cifrare i dati personali. Selected encryption: se cifro solo quei dati, sto dicendo all'attacker che i dati sensibili sono lì. Un altro metodo può essere il full disk encryption, ma non risolve tutti i problemi: cifrare/decifrare grava sulla CPU, l'operazione può essere costosa ed inoltre i dati passano in chiaro appena escono dal disco. L'approccio Enterprise è la pervasive encryption: cifro tutto, evito le modifiche dell'applicazione senza inserire la cifratura nell'applicazione. Diversi benefici:

- si cifra tutto e tutto viaggia cifrato nel sistema (i dati non sono solo cifrati nel disco)
- l'impatto di processamento dei dati viene fatto su hardware specifico e solo quando necessario
- tutto in container/servizi che permette di amministrare bene le chiavi di cifratura

Se ho i dati anche sulle repliche per ridondanza, il sistema di cifratura permette di recuperarli e di recuperare le chiavi di decifratura

8.2.4 Performances