ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA, INFORMATICA E TELECOMUNICAZIONI

STUDIO, ANALISI E PROGETTAZIONE DI UN PROTOTIPO DI INFRASTRUCTURE AS CODE

Elaborato in
SISTEMI DISTRIBUITI

Relatore
Prof. Andrea Omicini

Presentata da
Alessandro Mazzoli

Sessione II
Anno Accademico 2014-2015

INDICE

INTRODUZIONE

Infrastructure as Code è un fenomeno emerso da circa sei anni scaturito da un insieme di tecnologie che hanno influenzato pesantemente la rete come la conosciamo oggi.

Il punto di partenza sono proprio gli anni '60 durante i quali IBM sviluppa il concetto di virtualizzazione, come tecnica di astrazione delle risorse a vari livelli di un entità computazionale.
La virtualizzazione consiste nell'esecuzione di un certo numero di ambienti identici, detti guest su un singolo computer chiamato host. Ognuno dei quali emula il comportamento di una macchina fisica avendo a disposizione solamente una parte virtuale ed isolata dagli altri utenti. [1]

La virtualizzazione è inoltre la chiave per un altro modello di sistema distribuito, il Cloud Computing, per rendere disponibili attraverso la rete un set di risorse computazionali condivise, sia hardware che software, velocemente configurabili con minimo effort senza alcun intervento del provider. La definizione del NIST (National Institute of Standards and Technology) individua cinque caratteristiche principali del cloud:

1. **Self-service su richiesta**: un cliente può unilateralmente richiedere nei vincoli fissati dal contratto risorse computazionali, come tempo macchina, risorse di memorizzazione o altro, quando necessario per svolgere i suoi task, senza richiedere un intervento umano dei fornitori dei servizi stessi.
2. **Accesso a banda larga**: le risorse sono raggiungibili tramite la rete, la cui banda deve essere adeguata all'uso specifico richiesto, e vengono accedute con meccanismi che ne permettono l'uso con piattaforme client diversificate ed eterogenee sia leggere che complesse (ad esempio telefoni cellulari, computer portatili, o computer palmari).
3. **Risorse comuni**: le risorse di calcolo del fornitore vengono organizzate per servire più clienti, utilizzando il modello multi-tenant, in cui le risorse fisiche e virtuali sono

assegnate dinamicamente a seconda della richiesta dei clienti. Le risorse offerte sono indipendenti dalla loro locazione fisica, ovvero il cliente generalmente non ha né il controllo né la conoscenza dell'esatta locazione fisica delle risorse a lui fornite. Tuttavia, il fornitore potrebbe permettere all'utente di specificare dei vincoli sulla locazione delle risorse a lui assegnate in termini di area geografica, Paese o anche singolo data center. Esempi di risorse sono: risorse di memorizzazione, di calcolo, di rete e macchine virtuali.

4. **Elasticità**: le risorse possono essere fornite rapidamente ed elasticamente, ed in alcuni casi anche automaticamente, per incrementare velocemente la capacità computazionale, ed allo stesso modo possono essere rapidamente rilasciate per decrementare la capacità computazionale. Dal punto di vista dell'utente le risorse disponibili appaiono illimitate, e possono essere richieste in qualsiasi quantità ed in qualsiasi momento. 5. Servizi monitorati: i sistemi cloud controllano ed ottimizzano automaticamente l'utilizzo delle risorse, sfruttando la capacità di misurarne l'utilizzo delle risorse al livello necessario per il tipo di servizio (ad esempio servizi di memorizzazione, di calcolo, banda di comunicazione, ed account utente attivi). Il monitoraggio dell'utilizzo dei servizi è molto importante per permettere al fornitore di reagire ad eventuali picchi di richiesta 7 allo scopo di garantire al cliente la Qualità del Servizio promessa. L'utilizzo delle risorse può essere monitorato e riportato trasparentemente sia per il fornitore che per il cliente. [2]

L'avvento del Cloud Computing ha delineato fermamente il mondo ICT, portando all'adozione come può esserlo per un evento di Storia Moderna, i rispettivi termini Iron Age e Cloud Era. [3] La Iron Age, relaziona direttamente il grado di potenza computazionale all'hardware fisico. La Cloud Era poggiandosi sulla virtualizzazione e sui principi del Cloud Computing, astrae da networking, storage e potenza computazionale introducendo numerosi benefici:

- possibilità di suddividere le risorse hardware fra varie applicazioni
- carichi suddivisibili in un pool di risorse computazionali
- risorse velocemente provisionabili e allocabili on demand

- costi computazionali scalabili

Il poter disporre di un set pressoché infinito di potenza computazionale alla sola semplice pressione di un buttone, combinato ad una nuova generazione di web framework, evidenzia una serie di problematiche legate alla scalabilità e alla standardizzazione degli ambienti di sviluppo. Come poter scalare le risorse all'evenienza ricreando gli stessi ambienti, monitorandoli e mettendo in sicurezza i dati critici delle applicazioni?

La tesi risponderà proprio a questa domanda definendo un nuovo paradigma nel concepire le infrastrutture chiamato Infrastructure as Code, cui alla base vi sono due principi cardine:

- L'infrastruttura può e deve essere trattata come codice.
- Gli sviluppatori che trattano l'infrastruttura dovrebbero conoscere le stesse metodologie e pratiche dei classici sviluppatori. [4]

Le nuove tecnologie legate alla virtualizzazione, cloud computing e containers trasformano l'infrastruttura in un agglomerato di software e dati, potendo trattare l'infrastruttura come fosse un sistema software. Nonostante l'avvenuta di una nuova generazione di strumenti e tecnologie, le attività legate ad esse non hanno subito alcuna variazione:

- Task ripetitivi (installazione sistema di backup, vpn, firewall...).
- Realizzazione di script non idempotenti. [5]

Alcune possibili ragioni sono scarsa automatizzazione, deriva delle configurazioni ed erosione. La scarsa automatizzazione è un fattore presente soprattutto nelle grandi aziende, in cui il flusso di un singolo processo quali il provisioning di un nuovo server può protrarsi per un vasto intervallo di tempo.

Nell'ICT, un ambiente è composto sia da componenti hardware che software: è necessario assicurarsi che tutte le parti vengano opportunamente aggiornate. Inoltre i ruoli tra loro comuni facenti parte dell'infrastruttura (webservers, dbs, mailservers) devono avere la stessa configurazione.

Senza un sistema di registrazione delle configurazioni unito ad un' applicazione disomogenea può portare alla creazione del classico *snowflake server*. [6] Il termine snowflake o fiocco di neve deriva proprio dalla sua irriproducibilità e fragilità nel caso si vogliano apportare modifiche o nuove release del codice. Aggiornamenti del software potrebbero scatenare imprevedibili effetti a catena, rendendo gli addetti ancor più recidivi e conservatori nel mantenere applicazioni significative su release di sistemi operativi datati.

In un mondo ideale, una potenziale infrastruttura automatizzata, dovrebbe mantenersi stabile nel tempo. Tuttavia un nuovo evento come l'apporto di una nuova feature potrebbe intaccarne lo stato, questa caratteristica, coniata da Heroku, viene detta erosione. [7] Alcune delle cause che potrebbero crinare l'infrastruttura in corso sono:

- aggiornamenti del sistema operativo e del kernel, vulnerabilità (i.e Heartbleed, MySQL, Apache2)
- riempimento dello spazio fisico (dati applicativi, logs..)
- servizi instabili
- hardware failure

Il concetto di Infrastructure as Code quindi non nasce solo da un esigenza legata ad un aspetto tecnico, bensì anche da uno spirito rivoluzionario nel campo sistemistico tentando di ovviare ad una serie di attività potenzialmente lesive per la stabilità delle infrastrutture.

Infrastucture as Code è un approccio basato sulla Cloud Era nel costruire ed automatizzare una infrastruttura dinamica. Tratta l'infrastruttura, i servizi e gli strumenti come parti integranti di un sistema software adottando pratiche di Ingegneria del software per gestire i cambiamenti in una forma strutturata e sicura. [8]
Infrastructure as Code delinea i principi per poter essere in grado di costruire una infrastruttura dinamica ed automatizzata:

- **Riproducibilità**, deve essere possibile ricreare o riprodurre senza alcun sfrorzo tutti gli elementi comprendenti l'infrastruttura. La facile riproducibilità pone tra i vari

benefici: lo stato di sicurezza verso i nuovi strumenti di automatizzazione, pratiche di sviluppo software e politiche di scaling.

- **Consistenza**, fidarsi della propria infrastruttura è uno step fondamentale per renderla automatizzata. E' d'obbligo scongiurare qualsiasi deriva riguardanti le configurazioni.
- **Ripetitibilità**
- **Disponibilità,** l'idea che un particolare server possa smettere di funzionare o essere dismesso deve essere considerata come una feature e non come una vulnerabilità.
  La disponibilità è il risultato della somma di tutti i principi soprastanti.
- **Continuità del servizio**, in accordo alla disponibilità, i servizi devono essere sempre disponibili agli utenti. Questo è possibile solo analizzando i requisiti di ogni servizio disaccoppiandoli dalla infrastruttura. In caso di indisponibilità ad esempio di un webserver, dovrà esserne attivo sempre una copia o riproducibile instantaneamente.
- **Versionamento**, versionare l'infrastruttura è la pietra angolare dell'Infrastructure as Code. Rende possibile l'automatizzazione dei processi nell'apportare le modifiche includendo suite di test e auditing. Il VCS (Version Control System), spiegato esaustivamente nei capitoli successivi, è essenziale per la gestione delle infrastrutture fornendo le basi per tracciabilità, rollback, visibilità, trasparenza e trigger di eventi.
- **Autodocumentazione,** è estremamente complicato mantenere una documentazione rilevante, utile ed accurata sia nel caso che il processo coinvolga la sola persona o addirittura un team. Questo a causa della velocità degli aggiornamenti da documentare, dalla modalità in cui viene redatta e dalla mancata visione della criticità delle informazioni scritte da altrui. Un grande vantaggio è dovuto dal fatto che se l'infrastruttura può essere vista come codice, il codice stesso è documentazione.
- **Modifiche incrementali e tests,** applicando pratiche di Continuous Integration in aggiunta ad una suite di test esaustiva**,** si potranno raggiungere numerosi benefici: solidità del codice, velocità nel ricevere feedback, detto anche *fast feedback*, e rapidità nel risolvere bug. [9]

L'introduzione, ha già enunciato alcuni elementi fondamentali dell'Infrastructure as Code, si approfondirà il legame tra infrastruttura e codice, sviscerando concetti di Configuration Management, definizione degli ambienti di sviluppo e classici pattern dei sistemi distribuiti.

In secondo luogo, verranno trattate alcune delle metodologie di Ingegneria del software che maggiormente si legheranno al tema della tesi.

In conclusione verrà proposto un flusso tra i vari ambienti di sviluppo software mettendo in pratica gli argomenti precedentemente trattati.

CAPITOLO 1:
IT CONFIGURATION MANAGEMENT

Per Configuration Management si riferisce al processo di raccolta, recupero e di modifica di tutti gli artefatti e delle relazioni all'interno di un progetto.

La gestione di norma è rapportata ad una base di dati in cui vengono censiti gli oggetti sottoposti a controllo di configurazione chiamati di norma *configuration items*.[10] La gestione tratta tutte le fasi dello sviluppo software archiviando non solo le varie versioni del codice ma anche i prodotti delle singole fasi, i cosidetti artefatti.

Entrando nel dettaglio dello sviluppo web, il processo inizia dall'identificare i requisiti tali che l'applicazione web esegua tutti gli step, dalla build al deploy: un luogo adatto alla gestione degli artefatti e delle dipendenze. [11]

Possiamo definire un'adeguata strategia di Configuration Management se abile nel risolvere una serie di problematiche e casi d'uso :

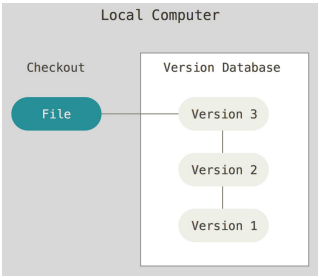
- riprodurre esattamente tutti gli aspetti dell'ambiente partendo dal sistema operativo fino al software di terze parti.
- in caso di modifica deve essere possibile applicarla al sistema di Configuration Management rendendola poi disponibile all'intera infrastruttura.
- ottenere lo storico dello modifiche avvenute nel tempo.
- soddisfare le normative di conformità.
- permettere ad ogni membro del team di ottenere le informazioni richieste senza introdurre nuove barriere.

Un'implementazione di un configuration managment system è il VCS (**Version Control System)**, un sistema che ha la funzione di registrare, mantenere e permettere l'accesso a versioni multiple dei stessi artefatti. [12]

Nel corso degli anni si sono susseguiti una serie di VCS adattandosi al periodo e al contesto IT.

Local Version Control Systems

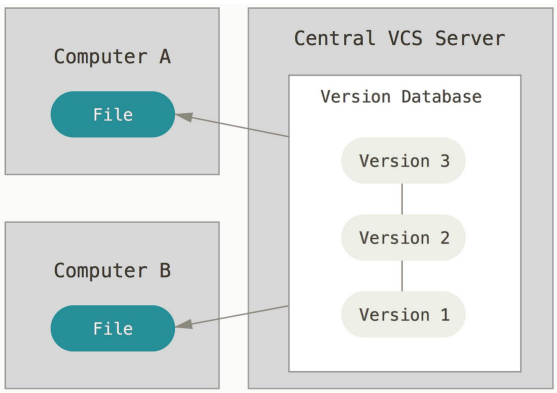
Il Local Version Control System predispone un database in grado di registrare le versioni dei file, evitando errori accidentali di sovrascrizione di file.



Centralized Version Control Systems

Il Local VCS tuttavia non è applicabile al mondo odierno in cui è richiesta collaborazione fra diversi soggetti.
I centralized CVS, quali Subversion e Perforce, avendo una classica architettura client-server permettono il multiaccesso. Questo setup garantisce numerosi vantaggi:

- tutti i membri del team, compreso il project manager, hanno conoscienza dello stato del progetto a cui stanno lavorando
- la gestione di un database centralizzato è preferibile rispetto a singoli database su ogni client



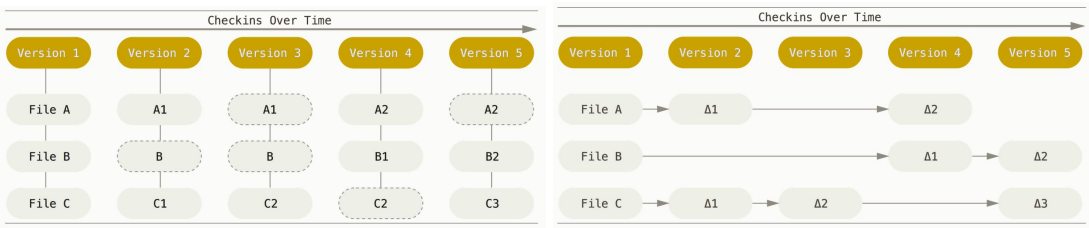
Per i sistemi distribuiti, questa topologia garantisce un single point of failure.
Un single point of failure rappresenta un singolo componente con la facoltà di rendere indisponibile tutta l'infrastruttura. In caso di failure, non sarà possibile per gli utenti accedere e modificare i dati all'interno del CVCS, interrompendo tutte le attività di sviluppo software.

Dieci anni fa, Linus Torvalds iniziò a sviluppare un sistema di version control chiamato Git in grado di mantenere le differenze all'interno del source code di Linux Kernel. [13]

Gli obiettivi di Git erano:

- Velocità
- Design semplice
- Supporto completo a flussi di sviluppo non lineari
- Pienamente distribuito
- Capace di gestire grandi progetti

Verranno introdotte alcune funzionalità di Git utili per realizzare l'obiettivo finale della tesi.
Git rispetto ai numerosi VCS, ha una diversa concezione di immagazzinazione delle informazioni, non più pensabile come una lista di files modificati, ma come un limitato set di snapshot incrementali.

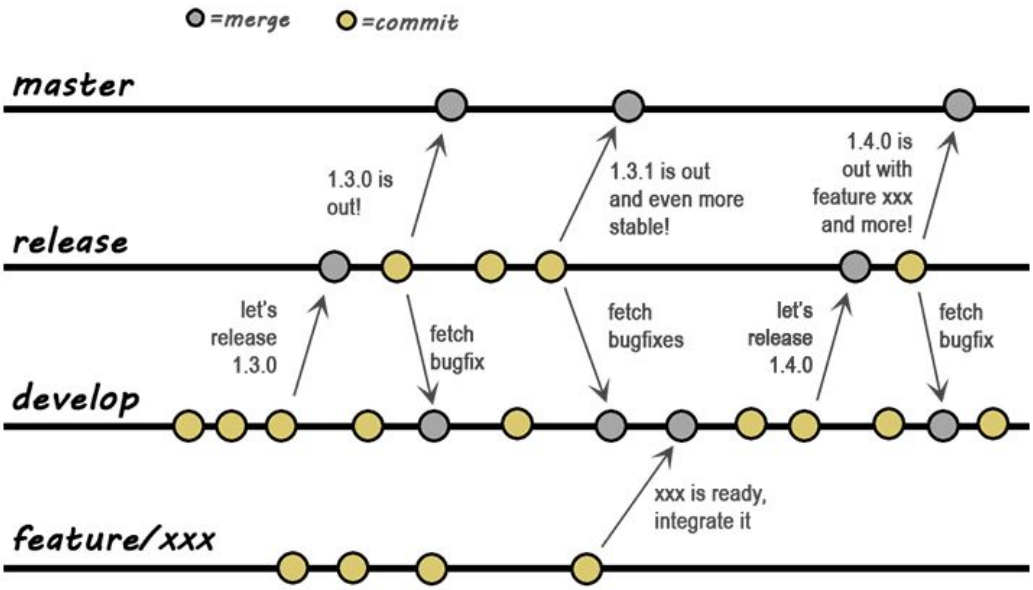


Git mantiene tutte le informazioni della repository in locale; in caso di mancata connettività, gli utenti potranno comunque applicare modifiche in locale e in un secondo luogo in remoto, favorendo la decentralizzazione del sistema.

Uno degli obiettivi di Git è lo sviluppo non-lineare, suddividendo il flusso in più rami del tutto indipendenti chiamati branch. Questa feature risolve una serie di scenari quali:

- *Switch di contesto senza attriti*, viene creato un branch per testare una patch, nel caso sia necessaria una modifica alla mainline, si può effettuare uno switch al branch principale fino a task completato per poi ritornare al branch di testing.

- *Linee di codice basato sul ruolo*, ogni ambiente dallo staging a produzione può essere associato ad un branch.
- *Workflow basato su funzionalità*, viene creato un branch per ogni nuova funzionalità, applicate le modifiche dal branch alla mainline, si elimina il branch relativo.
- *Branch sperimentali*, in caso di esito negativo il branch viene cancellato senza aver applicato nulla alla mainline.



Git, si dimostra quindi uno strumento estremamente utile a supporto sia dello sviluppo web sia della creazione e gestione di una infrastruttura abilitando versionamento, sviluppo non lineare e pratiche di sviluppo del codice, affrontabili in seguito. [14]

Dopo aver adottato il Version Control System, è di dovere gestire tutte le possibili dipendenze in cui l'applicazione possa incorrere. Una dipendenza esiste quando una parte di codice dipende strettamente da un'altra, il classico esempio sono gli applicativi JAVA dipendenti dalla JVM.

Gestione delle librerie esterne

Le librerie esterne sono le dipendenze che creano maggiori difficoltà in una infrastruttura: sono spesso di natura binaria, spesso installate a livello globale o addirittura nel sorgente.
Vi sono due modi ragionevoli per gestire le librerie esterne: definirle all' interno del CVS o dichiararle ed ottenerle tramite strumenti quali Maven o Ivy.

Utilizzando il CVS si ha chiaramente il beneficio di avere sia codice applicativo che librerie versionate durante tutto il processo di build, incorrendo tuttavia in corpose repository in scenari di applicazioni cross-platform (Debian vs Redhat, Windows, OSX...).

Utilizzando invece Maven come strumento di dependency management, si dichiara esattamente per ogni build le versioni delle librerie esterne necessarie all'applicazione a scapito di build automatiche. [15] Verrà ulteriormente approfondito il concetto di build automatiche o automated builds durante la trattazione della Continuous Integration. Per essere in grado di riprodurre la build è necessario documentare le dipendenze, trascurando per ora la scelta dello strumento.

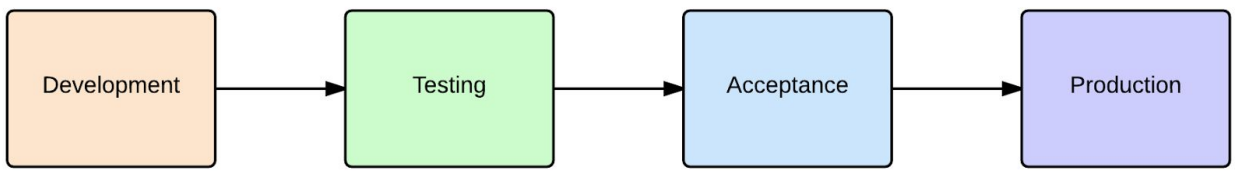
Gestione degli ambienti

"No application is an island"
(Jez Humble, David Farley, Continuous Delivery, Reliable software releases through build, test and deployment automation, 2010, Addison Wesley,p.49)

Un ambiente o environment è lo spazio compreso da tutte le risorse e configurazioni che l'applicazione necessita per il suo corretto funzionamento.
Possono essere di tipo:

- hardware(numero di CPU, ammontare di memoria, infrastruttura di rete)
- software(sistema operativo, webserver, mailserver, database...)

DTAP è una pratica di sviluppo software in cui si definiscono e si delineano ambienti chiari per ogni step di produzione del software: **D**evelopment, **T**esting, **A**cceptance, **P**roduction. [16]



Lo stage di development è di norma gestito individualmente da ciascun sviluppatore sulla propria workstation o su virtual machines in accordo ai sistemi di version control.
I requisiti base per uno sviluppo web sono perlopiù legati al webserver, database, linguaggio di programmazione ed un set limitato di risorse.

Nella fase di testing, come anticipato dal nome, avvengono i vari test sul codice.
Verrà approfondito il concetto di test durante il capitolo legato alla Continuos Integration.

Il codice testato pronto per essere rilasciato è controllato nell'ambiente di acceptance tendente a quello di produzione.

L'ambiente di produzione è lo stage finale dello sviluppo, in cui sia cliente sia gli utenti ne vedono il risultato in cui a differenza dell'ambiente di acceptance, teoricamente non vi possono errori o downtime.

Il peggior e comune approccio nella configurazione di ambienti è l'installazione di software modificando configurazioni rilevanti ad hoc. Questo porta ad uno dei classici punti interrogativi durante la definizione delle infrastrutture: come poter ritornare ad uno stato conosciuto accettabile, detto *known good state* in caso di failure? E' possibile riprodurre l'ambiente in automatico?

Il processo di rollback da una configurazione ad hoc fronteggia una serie di problematiche:

- una modifica non documentata potrebbe compromettere l'intera applicazione.
- in caso di failure, senza un'adeguata fonte di informazione, la risoluzione o la ricreazione dell'ambiente necessita un effort maggiore.

Per ridurre costi e rischi, è d'obbligo orientarsi verso un classico pattern dell'industria, la produzione di massa, in cui la creazione degli artefatti è predicibile e ripetibile.
E' necessario quindi ricreare un processo pienamente automatico per la creazione degli ambienti nell' ordine di:

- aumentare openness, limitando le singole conoscenze e disagi da possibili exit di personale.
- diminuire il downtime in caso di failure, ricreando gli ambienti in un intervallo temporale definito o fornire adeguate informazioni per attività di fixing.

- in ottica production-like, ricreare copie di ambienti di produzione per fini di user-acceptance testing o staging.

Come detto in precedenza, un ambiente comprende una collezione di risorse sia hardware sia software con le relative configurazioni:

- sistema operativo compreso versioni di major e minor release e patch ad esse correlate
- gestione dati (database, backup, retention..)
- software addizionali (mailserver, ftpserver,..)
- topologia di rete (zone DNS, firewall, LDAP)
- servizi esterni (monitoring, analytics..)

Per realizzare, quindi un sistema automatizzato è necessario una combinazione di virtualizzazione, remote installation management e strumenti di Configuration Management, parte di questi argomenti già enunciati durante l'introduzione, altri verranno trattati in seguito durante la tesi.

CAPITOLO 2:
STRUMENTI DI CONFIGURATION MANAGEMENT

Nel capitolo precedente è stato analizzato il processo di Configuration Management e il Version Control System, come strumento di collezione degli artefatti necessari durante l'intero workflow di progetto. Tuttavia ancora non è stato individuato un tool in grado di definire l'insieme dei componenti e delle loro relazioni all'interno di una infrastruttura.

In questo capitolo verrà eseguita un'analisi approfondita sui i tool maggiormente impiegati nella realizzazione e gestione delle infrastrutture odierne. Ciò nonostante prima verranno introdotti dei concetti base che influiranno sulla scelta di uno strumento a dispetto dell'altro.

Scalabilità, è la capacità di un sistema di restare inalterato nelle sue funzioni e proprietà nello scenario in cui le sue dimensioni varino. In un sistema distribuito le dimensioni possono voler significare sia l'aumento degli utenti o delle risorse partecipanti, la distribuzione geografica o l' aggiunta di nuovi domini amministrativi indipendenti.

"The scale of a system has three dimensions: numerical, geographical, and administrative. The numerical dimension consists of the number of users of the system, and the number of objects and services encompassed. The geographical dimension consists of the distance over which the system is scattered. The administrative dimension consists of the number of organizations that exert control over pieces of the system."
(B. Clifford Neuman, Scale in Distributed Systems, Readings in Distributed Computing Systems, IEEE Computer Society Press, 1994)

Orchestration, è un'attività che lega processi e risorse IT uniformandoli in flussi di lavoro, nell'ottica di fornire disponibilità di un ampio set di risorse identificati come servizi. Tipicamente un servizio di orchestration fornisce numerose features quali interfaccia da cui gestire tali flussi, layer in cui gli utenti possono modificare in real time i propri prodotti, un' orchestration layer che gestisce, governa ogni aspetto deciso dall'interfaccia grafica e infine un layer che astrae dall'effettiva risorsa computazionale.

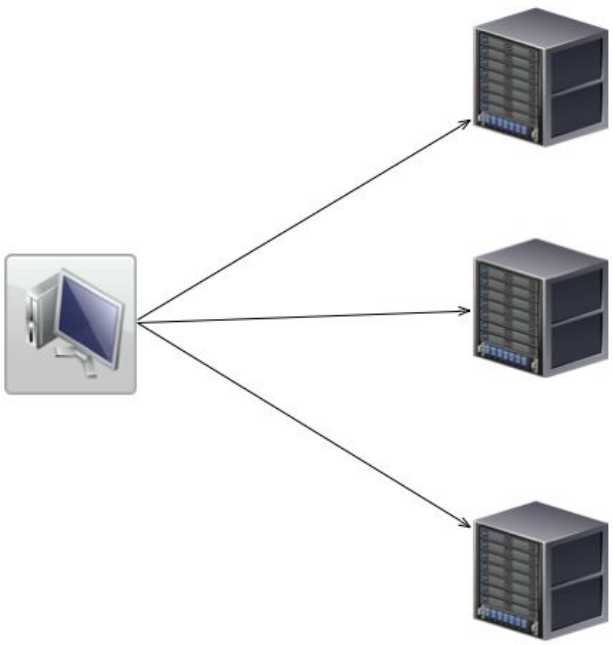
Idempotenza, nel contesto ICT è una proprietà attribuibile ad una operazione che può essere eseguita per un numero illimitato di volte il cui risultato ottenuto è pari alla singola esecuzione. Gli strumenti che verranno illustrati garantiranno questa fondamentale proprietà rendendo obsoleti qualunque altra soluzione legata allo shell scripting.

Tecnologie Pull vs Tecnologie Push

Push e Pull rappresentano una sostanziale separazione nella definizione delle architetture distribuite. In questi due modelli è identificabile un componente attivo ed uno passivo, definito in base a chi effettua la prima iterazione.

Nel modello push, un nodo server invia contenuto informativo verso gli altri nodi interessati senza alcun richiesta alcuna.
I maggior vantaggi di un approccio push sono:

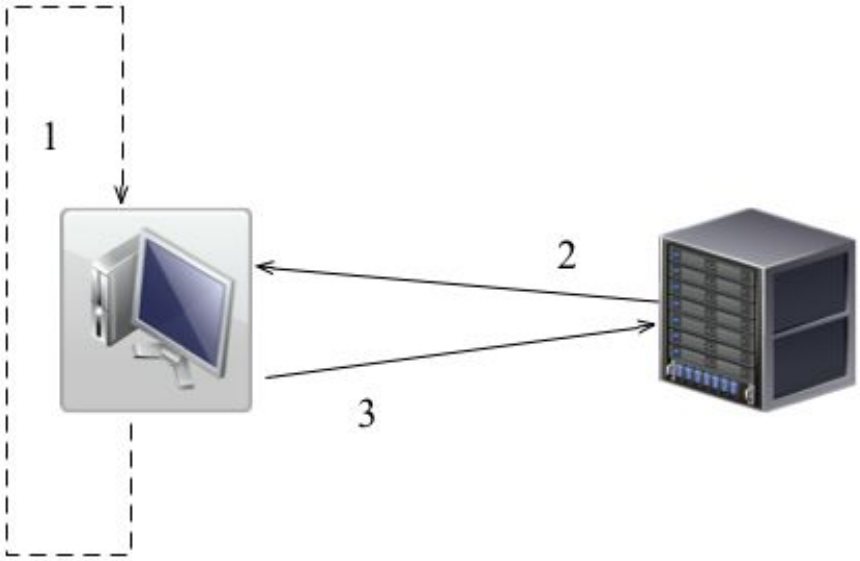
- **controllo**: ogni azione è sincrona sotto controllo dell' utente.
- **semplicità**: non implementando politiche di retrieving del dato, l'architettura è assimilabile ad un singolo utente che lancia comandi verso macchine remote. Strumenti di tipo push sono mediamente più semplici rispetto a strumenti di tipo pull.
- **minor richieste**, rispetto il modello pull, nel modello il componente attivo fornisce il dato, liberandosi da scenari di tipo DoS (Denial of Service).



Tuttavia il modello push nasconde alcuni svantaggi:

- **mancanza di full automation**: per configurare nuovi host è necessario l'intervento dell'utente.
- **minor scalabilità**: strumenti di tipo push hanno mostrato i propri limiti per numero crescente di hosts da configurare.

Nel modello pull, il nodo server attende passivamente(1) fino a quando un nodo client non effettua una richiesta di contenuto informativo (2), il server ne invia poi il contenuto(3).



Oltre ad essere soggetto ad attacchi DoS, il nodo server è gravato dalla complessità legata alla gestione dei contenuti, dovendo memorizzare il contenuto per ogni tipo di richiesta prevista, verificando se il demander è il destinatario corretto monitorando lo stato del buffer per scongiurare scenari di buffer overflow o presenza di contenuti superflui.

Il vantaggio più evidente in un approccio pull è la possibilità di *full automation*, fornendo al momento del provisioning uno strumento per l' instaurazione della connessione verso il server senza alcun intervento dell'utente. Si vedrà il caso di Puppet, in cui sarà preposto un processo agent che contatterà il server in questione.

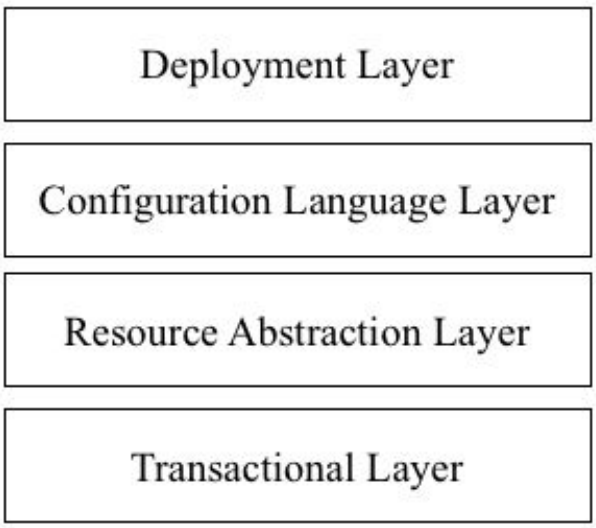
Il modello pull spesso prevede un linguaggio domain specific alzando notevolemente la curva di apprendimento.
Nonostante il modello pull abbia un grado maggior di scalabilità, in questo sistema la scalabilità dipende proprio dalla struttura centralizzata del modello: a meno di distribuire un numero maggiore di nodi server, il sistema sarà caratterizzato da un collo di bottiglia o *bottleneck* proprio in corrispondenza del server. [17]

Puppet

Puppet nato nel 2005 è uno strumento open source che permette la gestione dei vari ambienti durante l'intero life cycle sfruttando un linguaggio domain specific basato su Ruby per portare il sistema in uno stato desiderato. Inoltre dal 2008 è affiancato anche da una versione enterprise chiamata Puppet Enterprise con notevoli funzionalità aggiuntive. [18]

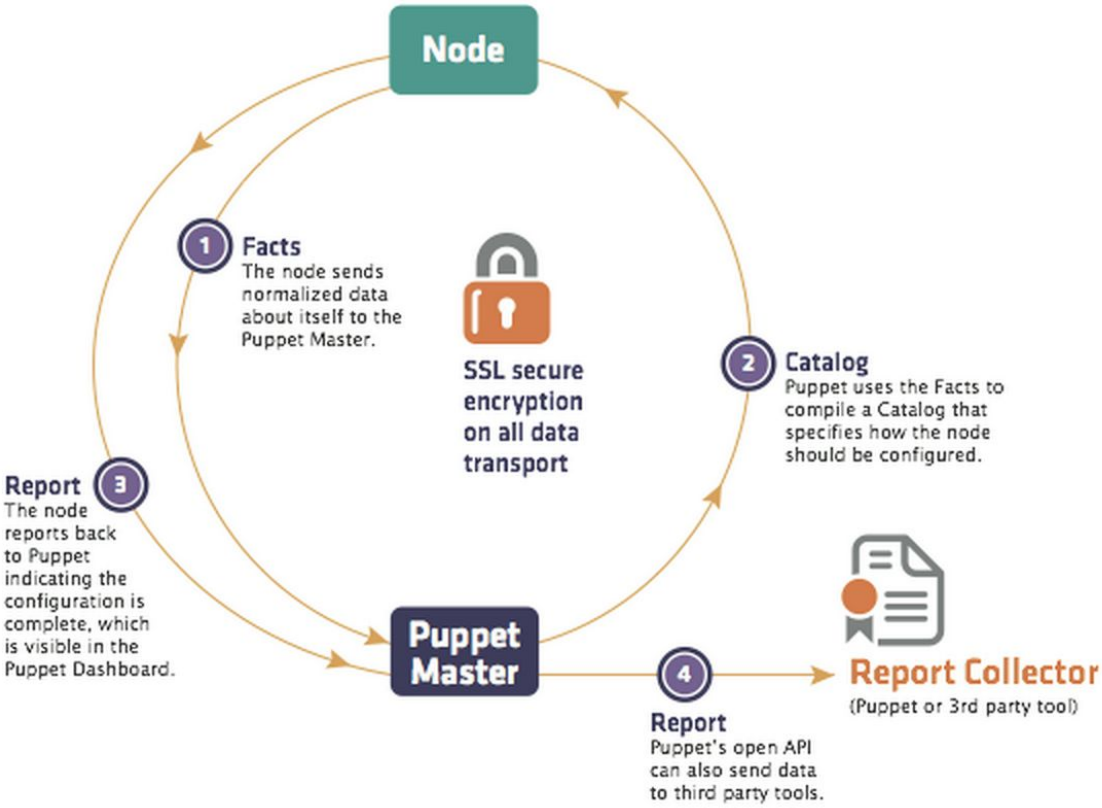
Modello:

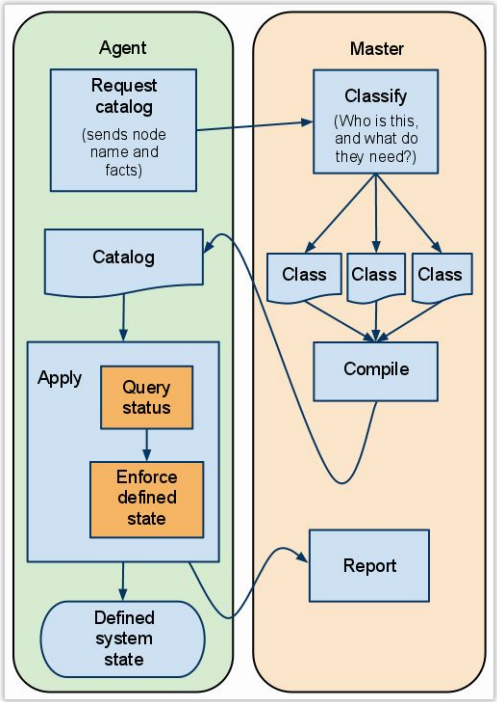
Seguendo un approccio top-down, in Puppet è possibile evidenziare quattro layer:



- **Deployment Layer**, Puppet si sviluppa, a discrezione dell'utente, in due possibili modalità: client-server sia serverless. Nella modalità serverless sia la fase di compilazione sia la fase di applicazione delle modifiche risiedono sull'agent.
  Nella modalità client-server invece, il server o puppetmaster mantiene un processo daemon in attesa delle connesioni provenienti dagli agent quali in un intervallo predefinito contattano il puppetmaster via HTTPS ottenendo le varie configurazioni

cui una volta applicate, forniscono feedback al master. Andando nel dettaglio, le interazioni tra puppetmaster e singolo nodo, formano un ciclo riassumibile in quattro passi: il nodo invia le proprie caratteristiche(OS, distribution, IP, MAC...) dette *facts*(1). Il master compila il *catalog* in base alle informazioni fornite dal nodo(2). Il nodo, ricevuto il catalog, tenta di applicare le modifiche (3) per poi fornire feedback al master (4).





Osservando la seconda figura si nota subito un particolare nella gestione delle configurazioni in accordo ai temi di IT security e al principio di *least privilege:* solo il master possiede le configurazioni generiche in formato chiaro text-based dette *manifest,* i nodi hanno solo una versione compilata detta *catalog*, specifica per quel nodo. Solo il master ha conoscenza delle configurazioni applicate nei singoli nodi.

"Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job. "
(Jerry H. Saltzer, Mike D. Schroeder, The protection of information in computer systems, 1975, *Proceedings of the IEEE*)

- **Configuration Language Layer,** Puppet utilizza un linguaggio dichiarativo per definire gli elementi caratteristici delle proprie configurazioni, sfruttando come concetto top level quello di risorsa. Ogni risorsa del sistema operativo (packages, users, files) è riassumibile, come in figura da un tipo, un titolo ed una serie di attributi.

```
<TYPE> { '<TITLE>':
    <ATTRIBUTE> => <VALUE>,
}
```

```
package { 'apache2':
    ensure => present,
}
```

Una classe, in Puppet è assimilabile ad un interno servizio o ad un'intera applicazione, che tramite gli elementi di *ordering* (before, after..) definiscono le relazioni fra le risorse coinvolte.

- **Resource Abstract Layer (RAL)**, il Resource Abstract Layer modella il concetto di risorsa. Il Resource Abstract Layer garantisce all'utente di astrarre dal know-how richiesto dalle varie distribuzioni e piattaforme incapsulando le varie implementazioni all' interno del tipo di risorsa (i.e package) e al provider relativo (apt o yum). Visto il flusso precedente, lato agent fornire i propri facts, significa permettere al master di compilare il manifest, selezionando il provider relativo alla distribuzione.
- **Transactional Layer**, ha la funzione di configurare ogni host associato al puppetmaster. Il Transactional Layer aggiunge anche la funzione di idempotenza, multiple esecuzioni di catalog forniscono in output lo stesso risultato. Il layer, tuttavia nonostante il nome lo possa far intuire, non accompie alla piena transazionalità vista nelle basi di dati, mancando della funzionalità di logging delle transazioni, esso non prevede alcun tipo di rollback. Il layer racchiude quattro fasi principali:
    a. Interpretazione e compilazione della configurazione.
    b. Invio della configurazione all'agent.
    c. Applicazione della configurazione nell'agent.
    d. Invio feedback del risultato al puppetmaster. [19]

Compilazione e interpretazione

Nello scenario classico, master / agent, l'agent invia una chiamata REST over HTTPS chiamata *find catalog*, il master in base ai facts forniti compilerà la specifica

configurazione per l'agent.
La compilazione richiede una serie di elementi:

- Agent-provided data
    - URI (i.e/puppet/catalog/web01.example.com?environment=production)
    - certificato SSL
    - facts
    - environments
- External data (LDAP, ENC)
- Puppet manifest (files , templates and modules)

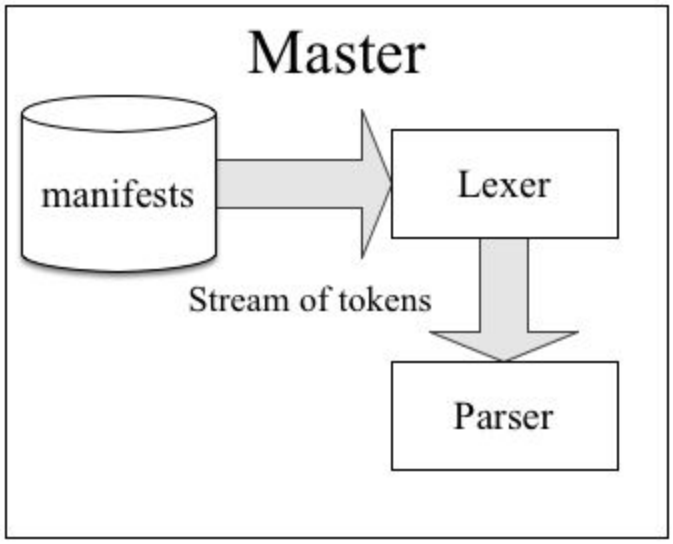
Come accennato soprastante, Puppet usa un linguaggio dichiarativo basato su Ruby organizzando le configurazioni in file di testo .pp detti *manifest.*

In questa fase le configurazioni text-based generiche vengono convertite in istanze di Abstact Syntax Tree specifiche per ogni nodo dette *catalog*. Il main manifest è un manifest in cui vengono inserite le node specification, le configurazioni attribuibili a ciascun nodo. Essendo comunque un manifest è possibile sia definire o includere classi.

```
node 'test.example.com' {
    class test {
        file { "/tmp/a" :
            content => "test!"
        }
    }
}

-- main.pp --
```

```
node 'text.example.com' {
    include test
}
```

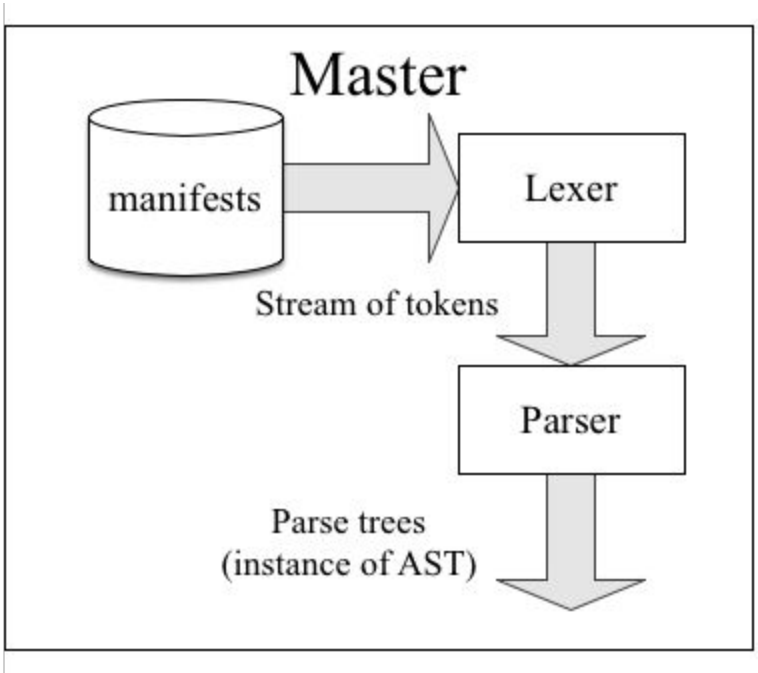
Ogni risorsa compresa nella node specification, è inclusa al catalog e quindi al processo di compilazione. Il compiler di Puppet è composto da quattro componenti: lexer, racc-based parser, modello AST e da una classe Compiler per gestire le interazioni tra di essi. [20]



Il lexer, presente anche in altri linguaggi, è un analizzatore lessicale con il compito di interpretare i manifest tramite regular expressions, catturando ogni possibile elemento caratteristico (reserved keyword, identifiers, variables, environments, regular expression..) fornendo in output uno stream di token. La figura sottostante riprende la classe test fornendo una visione in uscita dal lexer.

```
:CLASS(CLASS) {:line=>1, :value=>"class"}
:NAME(NAME) {:line=>1, :value=>"test"}
:LBRACE(LBRACE) {:line=>1, :value=>"{"}
:NAME(NAME) {:line=>2, :value=>"file"}
:LBRACE(LBRACE) {:line=>2, :value=>"{"}
:STRING(STRING) {:line=>3, :value=>"/tmp/a"}
:COLON(COLON) {:line=>3, :value=>":"}
:NAME(NAME) {:line=>3, :value=>"content"}
:FARROW(FARROW) {:line=>3, :value=>"=>"}
:STRING(STRING) {:line=>3, :value=>"test!"}
:RBRACE(RBRACE) {:line=>4, :value=>"}"}
:RBRACE(RBRACE) {:line=>5, :value=>"}"}
```

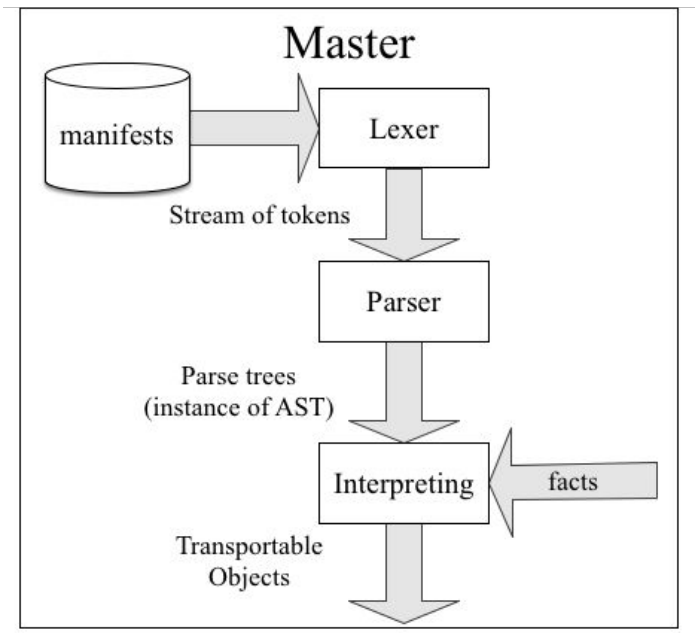
Il parser di Puppet è un LALR parser, chiamato Racc, un porting in Ruby del conosciuto Yacc. La grammatica da cui esso è generato è tipo LR(1), un parser di tipo deterministico, comunemente chiamato Shift-Reduce parser. Puppet, usando il modello AST, ogni dichiarazione facente parte della grammatica corrisponderà ad un' istanza AST di Puppet. [21]




```
AST::ASTarray
    AST::Hostclass
        AST::Resource
            AST::ResourceInstance
                AST::ResourceParam
                AST::String("content")
                AST::String("test!)
```

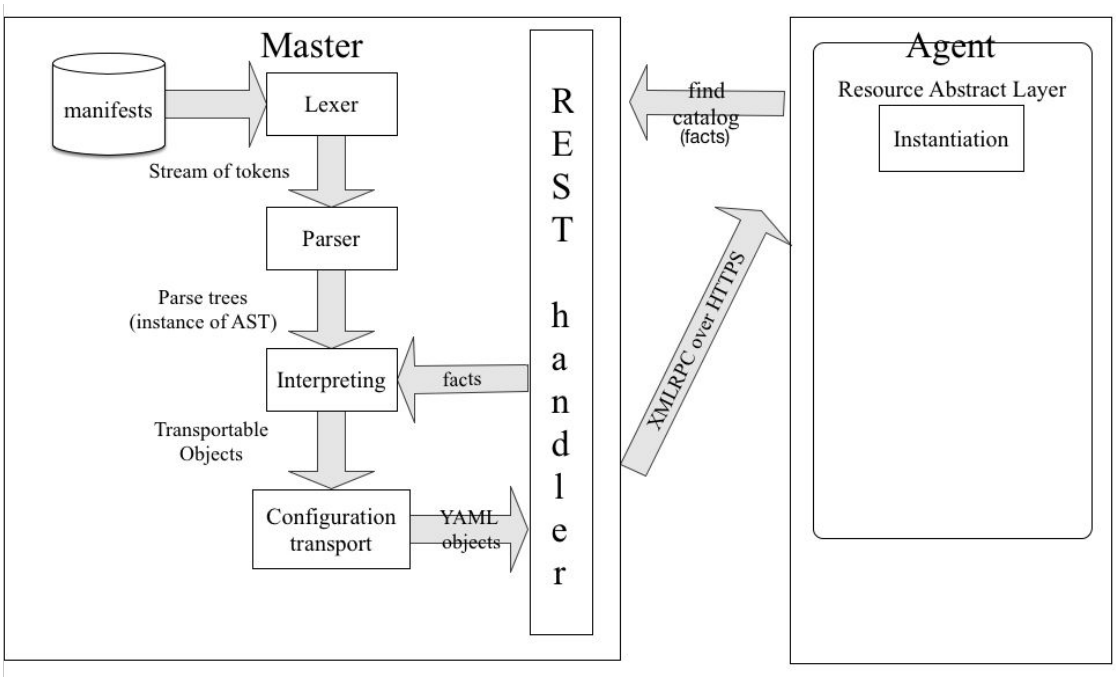
Una volta effettuato il check sintattico attraverso il parser, nella fase di interpretazione vengono iniettate le informazioni del nodo. L'Interpreter combina l'AST o parse tree con le informazioni dell'agent in una nuova struttura ad albero di *simple trasportable objects* che mappa direttamente la configurazione scritta nel manifest, composto dalle risorse e dalle relazioni specifica per quel nodo. Nel contesto di Puppet, un simple trasportable object non è nient altro che un hash seguito da una serie di metadati.

```
class TransObject
        include Enumerable
        attr_accessor :type, :name, :file, :line
        attr_writer :tags
        %w{has_key? include? length delete empty? << [] []=}.each { |method|
            define_method(method) do |*args|
                @params.send(method, *args)
            end
        }
        def each
            @params.each { |p,v| yield p, v }
        end
        def initialize(name,type)
            @type = type
            @name = name
            @params = {}
            #self.class.add(self)
            @tags = []
        end
        ......
```

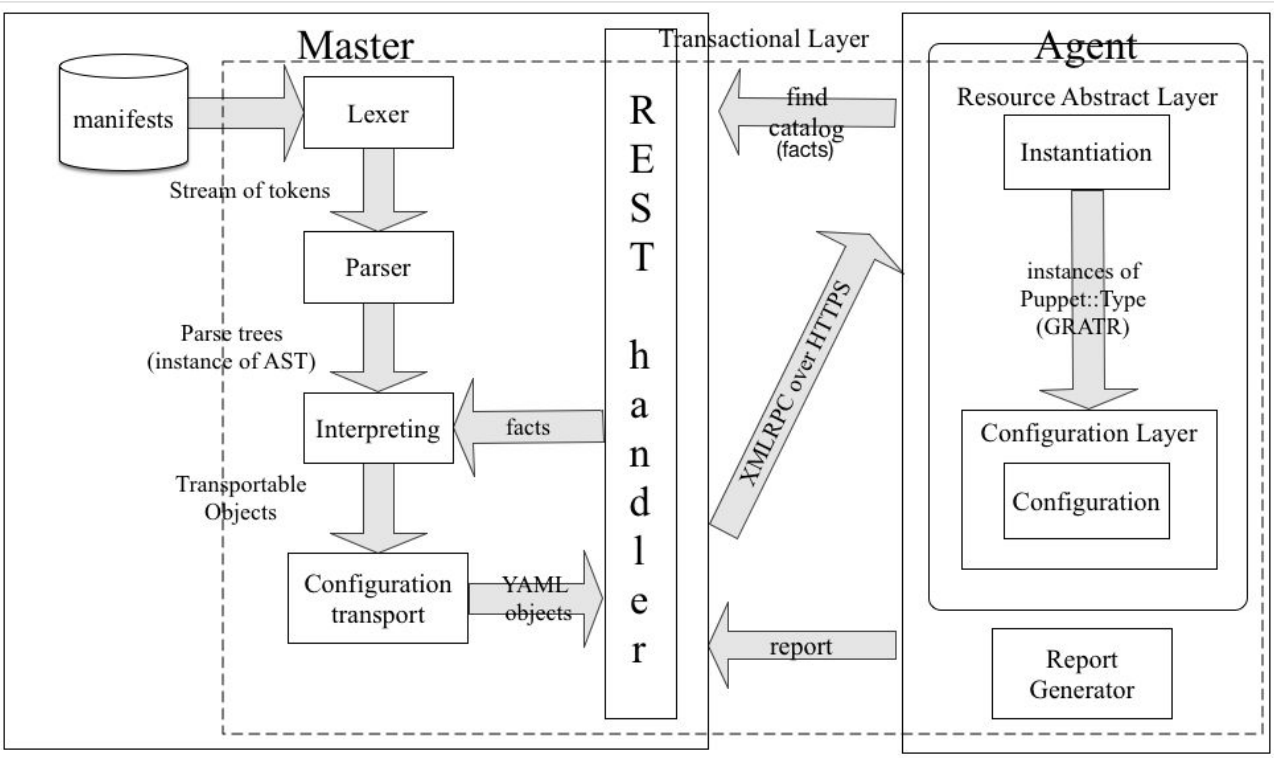


Il processo nel master si conclude nel Configuration Transport in cui la struttura ad albero in uscita dall'Interpreter viene convertita in formato YAML (Yet Another Markup Language) e inviata via XMLRPC over HTTPS all'agent.

```
class TransObject
        include Enumerable
        attr_accessor :type, :name, :file, :line
        attr_writer :tags

         ....

        def to_yaml_properties
            instance_variables
            #%w{ @type @name @file @line @tags }.find_all { |v|
            #}
        end
```



L' agent, tramite il Resource Abstraction Layer, converte nuovamente gli oggetti in YAML in Puppet Library objects per poi fornirlo al Configuration Layer. Il processo vero di configurazione e applicazione delle modifiche è trattato come una transazione il cui risultato verrà inviato al master.



Ansible

Ansible è un IT automation tool sviluppato in Python avendo come obiettivo la semplicità e la massima facilità d'uso. Questi due obiettivi sono in parte soddisfatti sfruttando due standard defacto: SSH come protocollo di comunicazione e lo YAML tramite cui definire le configurazioni.

Nel corso della trattazione di Puppet è stato affrontato immediatamente il layer Development analizzando come Puppet si interconnetta alle macchine remote. A differenza di Puppet, Ansible di default, è una tecnologia push che sfrutta il protocollo SSH per interagire con gli host remoti. Ansible a tempo di esecuzione delle configurazioni dette Playbooks, instaura n

connessioni SSH per numero di *plays,* pagando il tempo di negoziazione previsto dal protocollo TCP.

Ansible sfrutta una funzionalità di OpenSSH, detta SSH Multiplexing o Control Persist.

Il processo nell'instaurazione di una connessione SSH in modalità multiplexing può essere riassunto in una serie di passi:

- apertura di una connessione TCP tra client e server
- lato server viene creata una socket unix detta *control socket* associata al client
- nel tentativo di instaurare una nuova connessione, dallo stesso client viene utilizzata la control socket evitando di instaurare una nuova connessione TCP. [22]

Componenti

- **Inventory**, nella trattazione dei processi push, è stata evidenziata la mancanza di full automation e di auto discovery dei nodi agent, Ansible usa di default un documento statico INI-like detto Inventory in cui vengono inseriti staticamente gli host verso cui lanciare le configurazioni. [23]

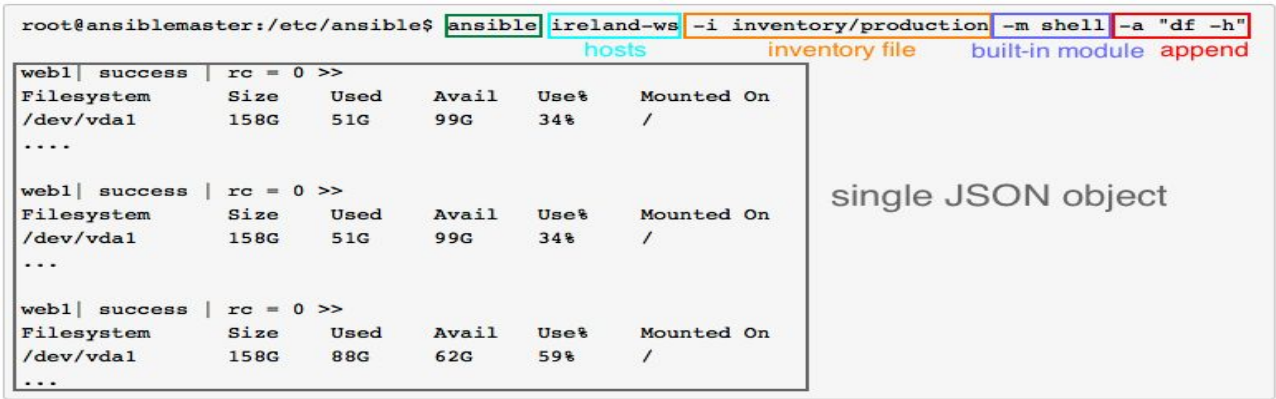
```
mail ansible_ssh_host=mail.myapp.com

web1 ansible_ssh_host=w1.myapp.com
web2 ansible_ssh_host=w2.myapp.com

web4 ansible_ssh_host=w4.myapp.com
web5 ansible_ssh_host=w5.myapp.com

db1 ansible_ssh_host=d1.myapp.com
db2 ansible_ssh_host=d2.myapp.com

[Ireland-webservers]
web[1:3]

[NewYork-webservers]
web[4:6]

[dbs]
db1
db2

[mailserver]
mail
```

- **Built-In Modules**, Ansible fornisce una batteria di moduli pre-installati per la configurazione dei classici servizi (ntp, firewalld, ufw, apache2..) e risorse utilizzabili sia tramite ad-Hoc commands sia tramite Playbooks.

```
- name: sysctl swappiness
  sysctl: name=fs.file-max value=64000 state=present
```

  Ansible non possiede un layer a livello di dichiarazione delle risorse quale poteva essere il RAL in Puppet, l'utente deve essere a conoscienza dei moduli applicabili alla propria distribuzione. [24]

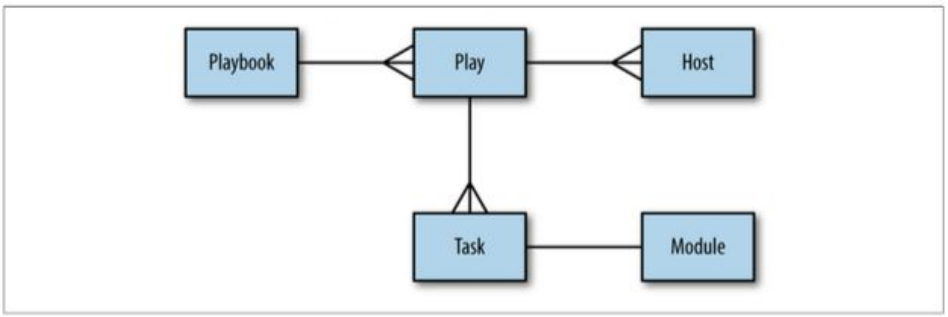
- **Ad-Hoc Commands**, sono comandi da poter lanciare parallelamente sugli host utilizzando i moduli o semplicemente in append alla shell. [25]



- **Playbooks,** come gli Ad-Hoc commands, sono entrambe due modalità per sfruttare le potenzialità di Ansible. Tuttavia, i Playbook non sono solo dei comandi lanciati verso host ben definiti, rappresentano bensì un sistema di configuration management, orchestration e deployment. I Playbook sono composti da *plays*, quali hanno l' obiettivo di mappare un gruppo di host in ruoli ben definiti implementando complessi flussi di orchestration. A livello base, un task non è altro che una chiamata a un modulo ansible.
    - tag *hosts* definisce verso quale host o gruppo di host verrà applicato il playbook, che dovrà coincidere con quello riportato nell'Inventory.

- *tag vars,* è possibile definire in Ansible, variabili a vari livelli, ognuna con un diverso scope:
  - extra vars, inserite in append ai comandi ansible
  - inventory vars
  - playbook, roles vars
  - defaults vars
- tag *task* raggruppa tutti i *plays* del playbook
- tag *handlers*, in aggiunta al modulo *notify*, è un sistema ad eventi per rispondere al cambiamento dello stato di una particolare risorsa (i.e l'aggiunta di un virtualhost, deve provocare il reload del webserver in questione). [26]

- **Roles,** per configurazioni complesse il playbook non è lo strumento adatto.



Il diagramma evidenzia la composizione e la verbosità dei playbooks, ad aggravare la situazione vi è anche la topologia o il set più o meno ampio degli host considerati. Ansible ha sfruttato un classico principio della programmazione, quale l' incapsulazione, in cui i Roles in Ansible hanno l'obiettivo di realizzare configurazioni complesse in una struttura ben definita. Si notano gli elementi caratteristici già introdotti nei Playbooks, che avranno una nuova forma:

```
roles/
    webserver/
        tasks/
            main.yml
        handlers/
            main.yml
        templates/
            ntp.conf.j2
        files/
            bar.txt
            foo.sh
        vars/
            main.yml
        defaults/
            main.yml
        meta/
            main.yml
```

```
- hosts: webservers
  roles:
    - common
    - webserver
```

Dopo aver trattato a fondo i due strumenti è possibile riassumerne i vantaggi e le criticità valutandoli in base ad una serie di parametri tra i quali: effort del setup iniziale, grado di orchestration e possibile workflow. [27]

Setup Iniziale: visto la sua modalità push Ansible non richiede alcun software di Configuration Management lato client. Ansible predilige infatti il protocollo SSH, standard defacto nell'accesso ad host remoti, e Python entrambi di default nelle recenti distribuzioni sia Debian che RedHat, mentre Puppet usando un'architettura master-agent richiede oltre ad una installazione del master anche di una versione agent entrambi richiedenti Ruby.

Curva di apprendimento: Nonostante anche Ansible abbia un linguaggio domain specific, combina i moduli pre installati allo standard YAML per agevolare la scrittura dei file di configurazione.
Dalle recenti versioni di Puppet è stato reso deprecato il linguaggio domain specific in favore di Ruby, rendendo ancor più complesso lo studio da parte del personale operativo.

Gestione del sistema di CM: Puppet basato su Ruby, deve aggiornarsi alle versioni di quest ultimo. Realizzato in un architettura master-agent, le versioni installate sia su master che su gli agent devono coincidere, creando un notevole effort nella gestione di questo sistema.

Ansible non richiedendo alcun software lato client, non ha necessità di aggiornare alcun componente aggiuntivo. Gli unici aggiornamenti che i client subiranno sarà legato ad OpenSSH e a Python facenti parte già dei classici aggiornamenti di sistema.

Cross Platform: Puppet fornisce supporto per un ben più ampio set di piattaforme enterprise e non, da Unix a OSX a Windows Server ed Oracle. Ansible, progetto rilasciato solo da qualche anno, richiede OpenSSH limitandone per ora l' uso.

Resource Ordering: Puppet relaziona le risorse con speciali tag *require* o *after*, realizzando come visto un grafo AST. Il flusso nell'esecuzione dei playbook su Ansible ha un approccio top-down sequenziale senza alcuna dipendenza.

Livello di automazione: Ansible essendo una push tecnology, richiede una maggior azione da parte dell'utente nell'applicare la configurazione desiderata. Puppet, lato provisioning può fornire tutti gli strumenti necessari: agent e certificato SSL automatizzando il processo di provisioning.

Livello di orchestrazione: Puppet Enterprise, combinando un'interfaccia matura e un middleware message oriented chiamato Apache ActiveMq riesce a fornire un servizio di orchestration ad alto livello.

Valutazione finale

La tesi, volgerà alla definizione degli ambienti di sviluppo trattatando alcune delle metodologie alla base dello sviluppo software che maggiormente si legheranno ad uno scenario DevOps. La scelta di Ansible come strumento per la prosecuzione della tesi è valutata secondo ben definiti parametri oggettivi:

- **scenario aziendale**, in azienda la realizzazione degli ambienti di sviluppo ha subito una drastica virata da Puppet ad Ansible, nell'ordine di:
    - diminuire l'effort nella risoluzione dei bug causati dalla combinazione Puppet e Ruby.
    - diminuire il tempo speso nell'istruire i nuovi componenti dei team optando per una learning curve meno ripida associata ad Ansible.

- **livello di orchestrazione richiesto**, prevedendo il corso naturale della tesi, l'autore non richiede di disporre un livello di orchestrazione elevato e di resource ordering previsto tra le feature aggiuntive apportate dalla versione enterprise.
- **tempo di setup iniziale**, visto la caratteristica di tesi triennale e il tempo limite di 200 ore, l'autore ha preferito optare per un setup veloce soddisfacendo comunque i requisiti richiesti.
- **esperienza personale,** dopo circa sei mesi di studio e di applicazione di Puppet, l'autore in accordo con l'azienda, ha optato per Ansible, considerando come primo fattore le dimensioni dell'azienda e l'eterogeneità dei sistemi realizzando un sistema di applicazione delle configurazioni descritto nel paragrafo di Ansible.

CAPITOLO 3:
PRATICHE DI INGEGNERIA DEL SOFTWARE APPLICATE ALLE INFRASTRUTTURE

Acquisite le conoscenze relative alla definizione degli ambienti e alla loro effettiva implementazione, visto l'Infrastructure as Code come fosse un sistema software, in questo capitolo verranno trattate alcune delle metodologie di sviluppo e produzione del software consone al pattern.

Continuous Integration

Facente parte dell'Extreme Programming XP, la Continuous Integration nasce dall'esigenza di risolvere una problematica comune detta **Integration Hell**.
L'Integration Hell è quello stato di difficoltà e disagio dovuto all'integrazione di vaste porzioni di codice sviluppate indipendentemente da diversi soggetti o team durante lunghi intervalli temporali che potrebbero essere tra loro divergenti.
Scenari comuni di Integration Hell:

1. Lo sviluppo di un'applicazione software che coinvolge un ampio numero di persone o team con competenze differenti, cui le integrazioni di vaste porzioni di codice avvengono durante lunghi intervalli temporali che potrebbero essere divergenti tra loro.

2. Il settore commerciale necessita di presentare le nuove feature ai clienti.
   La realizzazione di una build richiede un significativo sforzo da parte dello sviluppatore.

3. L'applicazione è stata consegnata al cliente ed è nello stage di User Acceptance Test.
   Il cliente richiede la risoluzione di bug o di migliorie che devono essere velocemente implementate saltando la fase di test. Questo potrebbe compromettere la stabilità dell' applicazione.

Più del software sviluppato da grandi gruppi spende una parte significativa del suo tempo di sviluppo in uno stato inutilizzabile, nessuno è interessato a lanciarlo o testarlo fino alla deadline.

Nasce l'esigenza di sviluppare software in maniera differente, la Continuous Integration:
"A software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to a multiple integration per day . Each integration is verified by an automated build to detect integration errors as quickly as possible."
(Paul M. Duvall, Steve Matyas, Andrew Glover, Continuous Integration, Addition-Wesley, 2007, pref.xxi) [28]

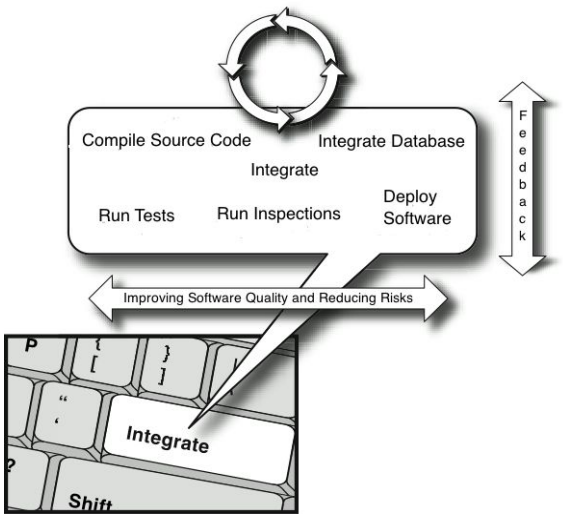
La Continuous Integration, per essere efficace e produttiva, deve essere una pratica condivisa da tutte le persone facenti parte del team, impattando sul loro sviluppo day-by-day:

- **Commit code frequently**, uno degli aspetti centrali della CI è integrare frequentemente al meno una volta al giorno. Si possono sfruttare varie tecniche quali :
    - *Make small changes*, non modificare più componenti alla volta ma scegliere un task minore scrivendo i relativi test integrando nel version control repository.
    - *Commit after each task*.
- **Build software at any change,** la CI richiede ad ogni integrazione che venga eseguita una nuova "build" dell'intera applicazione.
  Per build non si intende solo la compilazione, ma il processo che unisce il codice sorgente verificandone il funzionamento come fosse un unico componente.
- **Creating a comprehensive automated test suite,** avere delle build con risultato positivo, senza avere dei test chiari, significa solo che l'applicazione può essere compilata ed assemblata. I test devono garantirne l'effettiva funzionalità.
- **Don't commit broken code**, si raccomanda l'esecuzione degli script di build in locale prima di integrare in produzione.
- **Don't check in on a broken build,** se una build fallisce, gli sviluppatori devono identificare la causa risolvendola il più velocemente possibile.
  Integrare su una build rotta, richiederà un maggior sforzo per renderla nuovamente funzionante.
- **Always run all commit tests locally before commiting**
- **Never go home on a broken build**

Realizzare un sistema di Continuos Integration può essere complicato, comportando un effort aggiuntivo ai componenti del team di sviluppo, tuttavia si notano numerosi benefici in termini di:

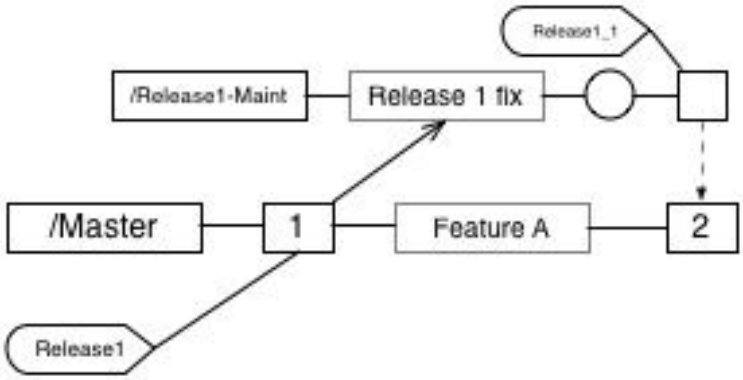
- **Reduce Risks**
  si riducono fortemente i rischi apportando numerosi benefici:
  - Detection & Fix dei bug in tempi brevi
  - Stato di salute del software misurabile
  - Riduzione delle assunzioni
- **Reduce Repetitive Processes**
  Automatizzandoli in un sistema di Continuous Integration, si guadagna in termini di tempo e costi.
- **Enable Better Project Visibility**
- **Estabilish Greater Product Confidence**
  Grazie ad un sistema di Continuous Integration pienamente funzionamente, si può avere lo storico delle build.
- **Generate Deployable Software**
  Il processo di Continuous Integration insieme alle pratiche di Continuous Deployment permette di avere software funzionante a qualunque step di sviluppo. Questo risolve gli scenari di Integration Hell, spiegati precedentemente: il settore commerciale ha sempre a disposizione una versione dell'applicazione funzionante al passo con lo sviluppo, il team di sviluppo ha una versione precedente in cui poter integrare le minors o in caso di bug aver comunque possibilità di effettuare rollback. [30]

Un sistema di Continuous Integration è rappresentato dal concetto di *Integrate Button*.
L'Integrate Button è una astrazione di una generazione di build pienamente funzionante ed automatizzata, che rende l'integrazione un "non evento". [31]



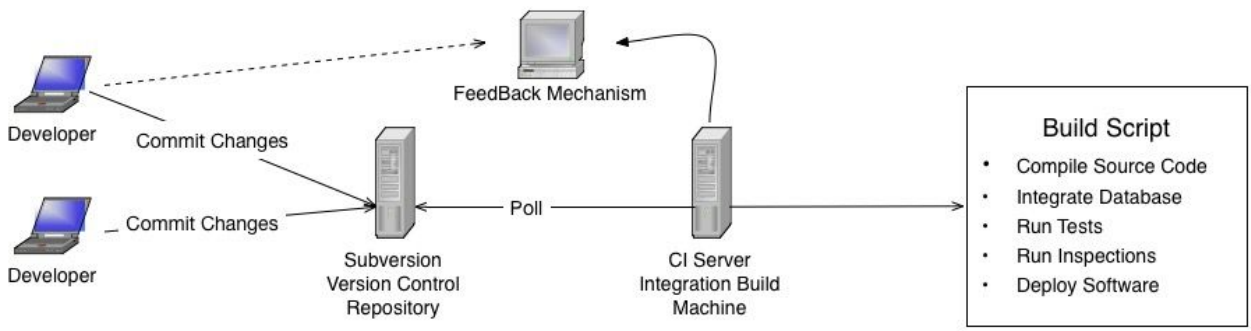
Tipicamente un deployment, a dispetto della tecnologia e dell'applicazione è composto da sei macro step:

1. **Assegnare una specifica etichetta alla repository**
   Assegnare un'etichetta a una specifica repository facilita l'individuazione e delinea l'appartenenza ad un gruppo di files.



2. **Produrre un ambiente privo di assunzioni**
   E' fondamentale supporre che non vi siano configurazioni nascoste che potrebbero far fallire il software. Tipicamente si suddivide l'ambiente in layer, classificandoli in base alla loro criticità (configurazioni del sistema operativo, configurazioni dei servizi, strumenti di terze parti) aggiungendoli o rimuovendoli in base alle necessità.
3. **Generare una build, associarla ad una specifica etichetta alla build ed installarla nell'ambiente**
   Assegnare un etichetta ad una repository significa mettere in relazione un certo numero di files, assegnare un'etichetta ad una build significa invece identificare univocamente un output binario.
4. **Eseguire tutti i test con esito positivo in un simil ambiente di produzione**
5. **Generare feedback relativi alla build**
6. **Se necessario, effettuare rollback alla versione precedente**

Continuous Integration stack



Version Control Repository

E' un componente fondamentale per realizzare un processo di Continuous Integration.
Lo scopo del Version Control Repository è di gestire le differenze nel codice, garantendo un "single point source code" da cui ottenere diverse versioni del codice.
Per codice si intende qualunque tipo di codice (source code, database scripts, tests, build e deployment script) necessario per creare, installare e testare l'intera applicazione.

Automated Build

E' un insieme composto da uno o più script utilizzati per compilare, testare e realizzare l'applicazione.
A seguito è proposto un esempio di un Ant script per realizzare una build.

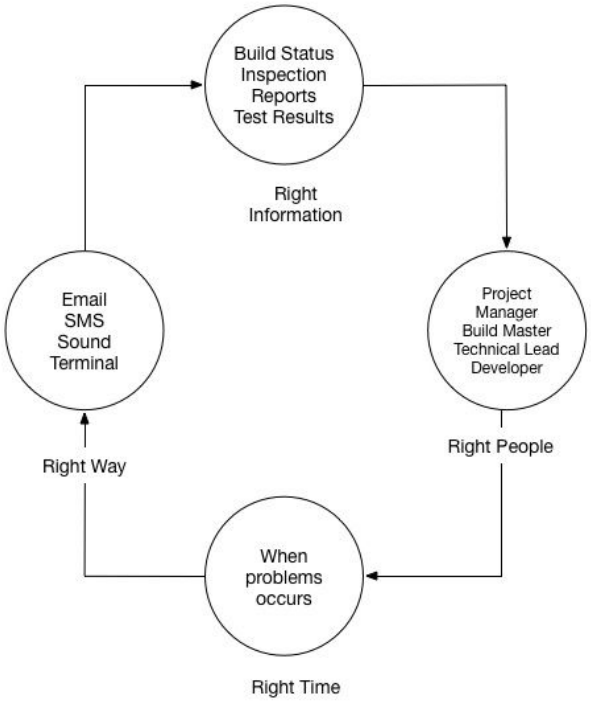
```
<?xml version="1.0" encoding="iso-8859-1"?>
<project name="infrastrure as a code" default="all" basedir=".">
        <target name="clean" />
        <target name="svn-update" />
        <target name="compile-src" />
        <target name="compile-tests" />
        <target name="integrate-database" />
        <target name="run-tests" />
        <target name="package" />
        <target name="deploy" />
</project>
```

Continuous Integration Server

Un Continuos Integration server esegue una build ogni volta che vi sono modifiche al codice applicate alla version control repository.
Il server CI ottiene il codice, esegue lo script di build e infine pubblica il risultato su una opportuna dashboard.
E' fondamentale che il server di Continuous Integration effettui le build appena si verifichino modifiche sul version control repository, riducendo sensibilmente i tempi di error detection.

Continuous Feedback Mechanism

La Continuous Integration richiede prontezza nel correggere gli errori dovuti all'integrazione, il team deve essere in grado di ricevere feedback dal sistema di build.
Ottenere velocemente feedback risulta quindi uno dei nodi centrali: è inutile avere build performanti senza avere un sistema di feedback che avvisi lo sviluppatore all'istante.
Si parla di "right information to the right people at the right time and in the right way". [32]

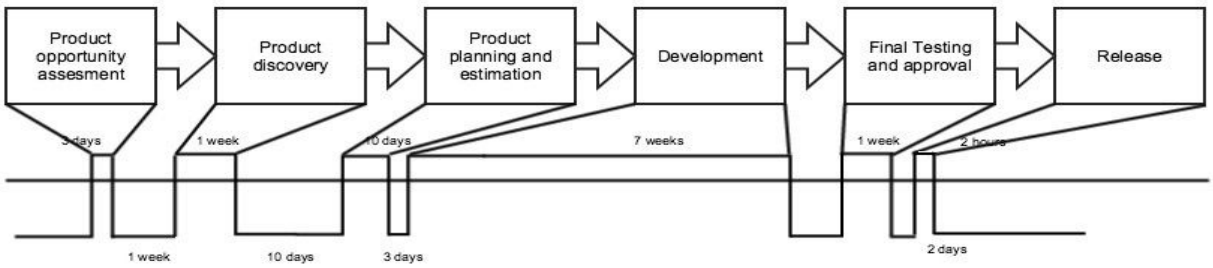


Continuous Delivery

La Continuous Integration è un enorme passo in avanti in termini di produttività e di qualità del software, la Continuous Delivery invece, assicura che l'intero sistema sia *production-ready* ad ogni release del codice riducendo gli sprechi e i tempi di attesa nel rilascio del software.
Gli scenari più comuni sono:

- il team sistemistico richiede documentazione e fix dal team di sviluppo
- il team di testing richiede versioni sempre più aggiornate del software
- il team di sviluppo riceve richieste di bugfix mentre è già al lavoro su una nuova funzionalità
- allo step di release l'applicazione non supporta i requisiti non funzionalità

Queste situazioni portano il software a non essere sviluppato o a subire evidenti ritardi.
Nella sottostante value stream map è rappresentato un ipotetico processo di release del software quale impiega tre mesi per giungere agli utenti ma cui l'effettivo valore aggiunto (campo value added time) è di soli due.

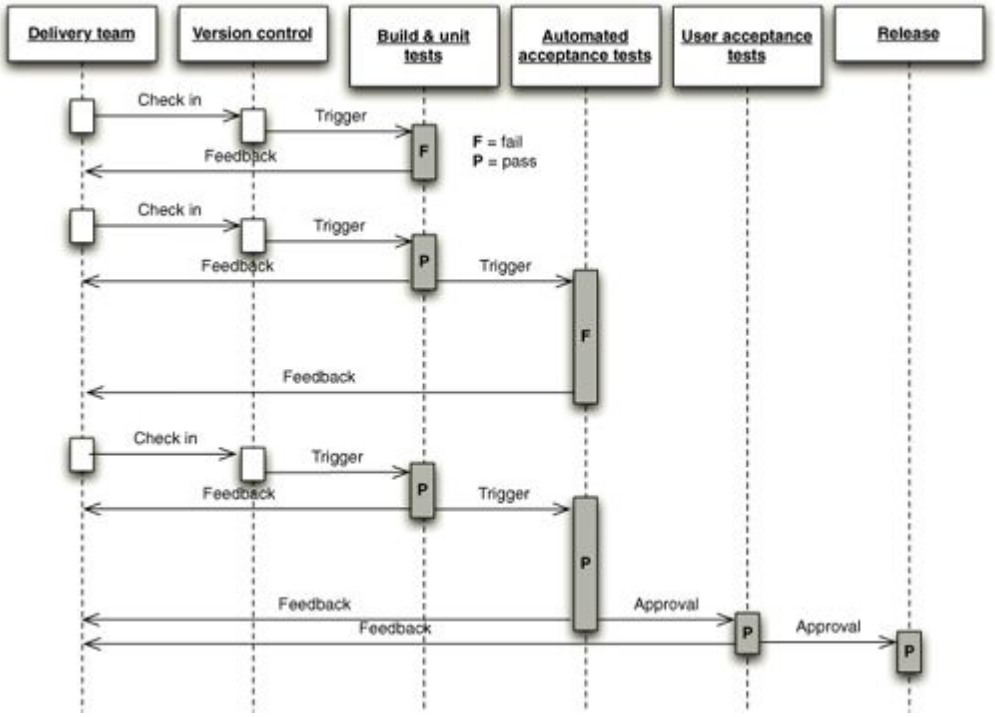


Il concetto fondamentale nella Continuous Delivery è la deployment pipeline, una rappresentazione del flusso di sviluppo dalla commit alla release di una particolare feature, basata proprio sulla Continuous Integration.
La deployment pipeline ha due benefici sostanziali:

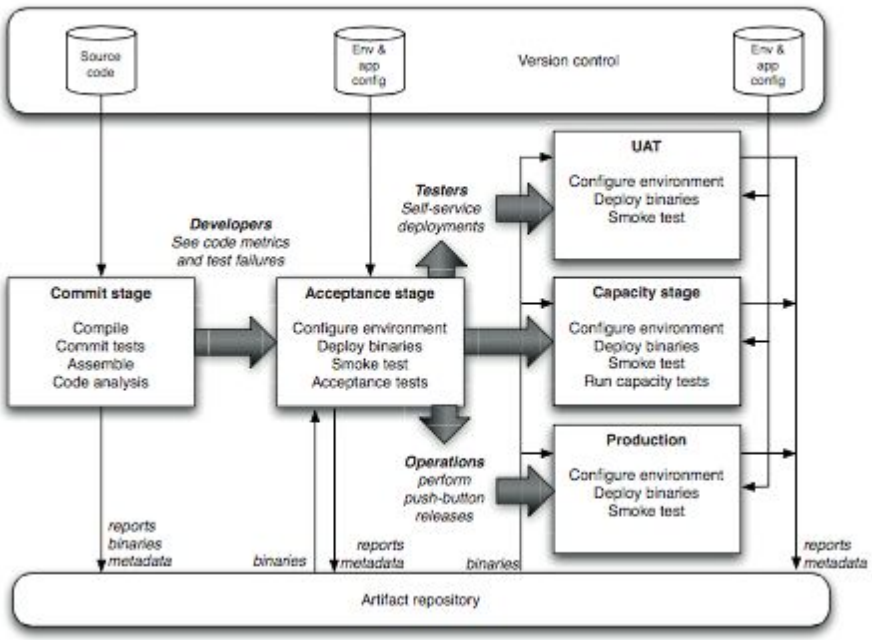
- esclude build che non portino valore giunghino in produzione
- automatizza il rilascio del software nei vari step, il processo acquisce ripetitibilità e affidabilità

La causa dei ritardi può essere implicabile ad una mancanza di Configuration Management e di automatizzazione.

L'obiettivo ora non è più l'*Integrate Button* bensì il *Deploy Button.* Automatizzando il deploy, rendendolo un non evento, si crea un pull system nella quale ciascun team all'esigenza recupera il "manufatto" realizzato nello step precedente. Per non evento significa non dover predisporre azioni a riguardo come la creazione di una finestra di manutenzione o uno stop forzato al team di sviluppo.
La deployment pipeline è un processo automatizzato per condurre il software dal version control all' utente finale. Ogni modifica al software percorre una serie di step visibili in figura sottostante coinvolgendo diversi ruoli all'interno dell'azienda.



Queste ultime frasi, fanno intuire che la deployment pipeline incarna tutto il concetto di Continuous Integration e di Configuration Management, in cui ad ogni modifica del software si scaturiscono una serie di azioni quali la generazione della build o il lancio della suite di testing. L'immagine sottostante descrive una tipica deployment pipeline, partendo dallo sviluppatore che applica la prima modifica (azione di commit) nel Version Control System.



In questa fase detta *commit stage,* il sistema di Continuous Integration interroga il CVS sull'esistenza di nuove commit, in caso positivo crea una nuova istanza della deployment pipeline.

Lo stage di commit ha il compito di escludere ogni possibile build inadeguata ad essere portata in produzione, fornendo subito feedback.
Nello stage di commit:

- viene compilato il codice
- vengono eseguiti gli unit tests
- vengono effettuate analisi sul codice
- vengono creati gli installer

Gli unit test, sono test applicati alle singole funzionalità o ai componenti base dell'applicazione. Nel caso gli unit tests diano esito positivo, il sistema di Continuous Integration, tipicamente in automatico, passa allo step successivo di acceptance. In caso negativo l'istanza della deployment pipeline viene semplicemente scartata in favore di una nuova build.
Gli unit tests non sono in grado di assicurare che la build sia funzionante in produzione o che soddisfi i requisiti richiesti dal cliente.

L'obiettivo dello step di acceptance è di assicurare che il prodotto soddisfi i requisiti del cliente, portando effettivamente valore. Un criterio di acceptance può essere funzionale o non funzionale. Un requisito non funzionale coinvolge tutti gli addetti responsabili: dai developer agli analisti fino agli operators, richiedendo non solo una conoscenza tecnica di tutta l'infrastruttura ma anche conoscenze di business.
Eseguiti i test di acceptance con esito positivo il sistema di deployment crea flussi separati in base al numero di ambienti in cui si deve eseguire il codice.
In questo stage è consigliabile avere un sistema di deployment che permetta sia l'esecuzione in automatico come prevede la *Continuous Deployment* o manuale a discrezione dello sviluppatore.

Rispetto alla Continuous Deployment, la Continuous Delivery permette di aver facoltà decisionale sul rilascio del software ricendo feedback anche da altri settori come quello tecnico, dirigenziale o di marketing per citarne alcuni.[32]

Osservate le varie fasi della deployment pipeline, per attenuare i tempi di attesa e ridurre i rischi di software non sviluppabile vi sono alcune best practices da seguire:

- **Stessa modalità di deployment in ogni ambiente**, i vari ambienti di sviluppo seguendo l'approccio DTAP, differiranno per alcuni fattori quali:
    - politiche di sicurezza (password strength, user jail...)
    - ottimizzazioni

    - caratteristiche fisiche (spazio disco, RAM, CPU)
    - caratteristiche di networking (IP, MAC)

  Il processo di deployment dovrebbe permettere l'impostazione di questi parametri ai vari ambienti attraverso strumenti di version control e di gestione credenziali (LDAP, via CLI, ..).

- **Deploy in ambienti simil produzione**, uno dei problemi che causano maggior problemi durante il processo di sviluppo software è lo scostamento tra ambiente di test/staging rispetto quello di produzione. Per far sì che gli sviluppatori abbiano un discreto livello di confidenza, i test che verranno eseguiti durante la deployment pipeline, dovranno essere rivolti verso ambienti simil produzione o production-like. Ottenere copie esatte dell'ambiente di produzione può risultare complesso, bisogna considerare un numero elevato di componenti e variabili:
    - Infrastruttura (network, firewall..)
    - sistema operativo (distribution, release, kernel version)
    - Stack applicativo (webserver, database, network file system)

  Strumenti, già introdotti quali CVS, virtualizzation, disk imaging e strumenti di Configuration Management tra cui Ansible, sono in grado insieme di gestire questo parrticolare requisito.

- **se uno step fallisce tutta la pipeline fallisce**, il processo di Continuous Delivery deve essere veloce, ripetibile e affidabile e fornire un feedback adatto per un'immediata risoluzione, la pipeline deve fallire in qualsiasi step si trovi.

Acquisito i vari step e le best practices, nel capitolo successivo verrà realizzato un prototipo di una deployment pipeline applicando i concetti e i strumenti appresi nei capitoli precedenti.

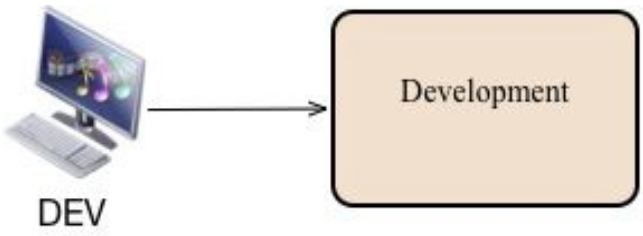
CAPITOLO 4:
PROTOTIPO DI INFRASTRUCTURE AS CODE

Il capitolo finale consiste nell'applicazione dei principi cardine dell'Infrastructure as Code, sfruttando alcune delle metodologie legate allo sviluppo software, strumenti di version e di Configuration Management ad un progetto Symfony in corso.

Annunciato più volte ma mai introdotto, il contesto aziendale è quello di *ideato*, una software house nata nel 2008. Attraverso lo sviluppo realizza siti web e software, progettando strategie e implementando servizi online. Basando tutta l'operatività su un approccio Lean minimizza gli sprechi e persegue gli obiettivi attraverso il circolo virtuoso del software, misurando i risultati, capendo quello che emerge per poter ricominciare. [34]

L'obiettivo di questo capitolo sarà legarsi all'approccio Lean minimizzando gli sprechi rilevati lungo il processo di produzione del software dalla fase di sviluppo fino al cliente finale riprendendo i concetti della Continuous Delivery.

Nel capitolo IT Configuration Management, era stato definito il concetto di DTAP come separazione concettuale degli ambienti di sviluppo. Osservando il flusso di sviluppo del software interno è stato possibile sintetizzare un set di ambienti isolati: development, testing, staging e production.



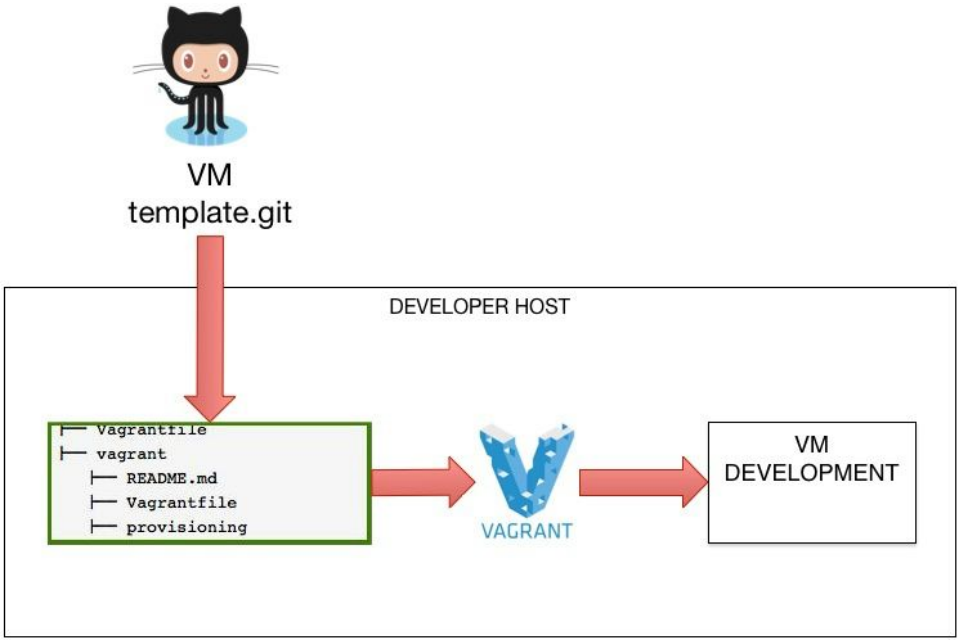
Come primo passo vi è quindi la creazione dell'ambiente di sviluppo o di development.
Precedentemente si era analizzato la value stream map individuando in quali step si verifichino i tempi di attesa e gli sprechi durante il rilascio del software: uno dei luoghi è proprio lo step di development in particolare durante il quale il developer deve realizzare l'ambiente ideale per lo sviluppo dell'applicazione.

La fase di development in Ideato si realizza in macchine virtuali presenti in locale utilizzando uno strumento chiamato Vagrant. [35] Creato al top di alcuni software di virtualizzazione tra i quali VirtualBox [36] e VMware [37], automatizza i classici step manuali come il setup del networking o dello storage acquisendo le configurazioni da un unico file statico detto Vagrantfile. Vagrant inoltre rende possibile l'utilizzo dei più recenti strumenti di Configuration Management quali Ansible, Puppet o Chef per definire risorse e servizi all'interno dell'ambiente virtualizzato.

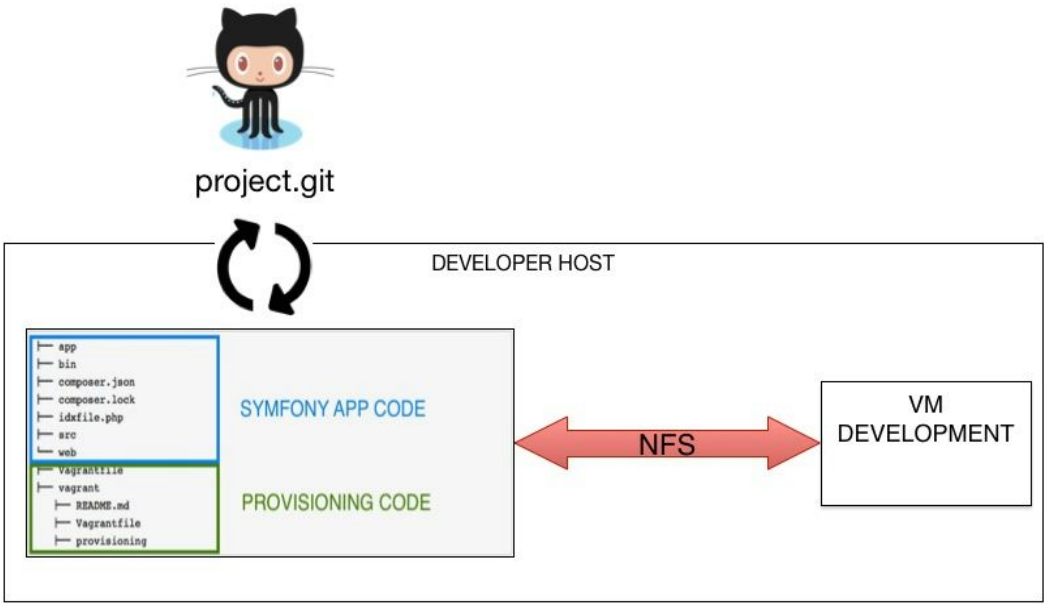
Abbinando quindi ai classici sistemi di virtualizzazione, i sistemi di Configuration Managment, al tempo di boot verrà sia lanciato il provisioning del sistema operativo sia la configurazione dei servizi necessari allo sviluppo dell'applicazione.
Nella figura sottostante vi è un classico Vagrantfile dove sono inserite alcune configurazioni legate alla virtual machine e all'utilizzo di Ansible.

Automatizzando la creazione dell'ambiente di sviluppo, non si garantisce comunque l' esatta riproducibilità dell' ambiente fra i vari componenti facenti parte di un progetto.
L'importanza di rendere ai sviluppatori l'esatta copia dell'ambiente, priva di bug, difetti o di task manuali, si lega proprio come già anticipato al concetto di Lean Production, nella quale ogni azione che non porti effettivo valore al cliente finale venga considerata spreco e quindi da eliminare.

La soluzione già adottata è stata quella di realizzare un template custom, ideale per lo sviluppo web mantenuto sotto controllo di versione combinando Vagrant ed Ansible.



Symfony è un noto framework per lo sviluppo di applicazioni web creato nativamente per PHP che sfrutta una serie di librerie PHP riusabili dette Symfony Components. [38]
Considerando Symfony quale PHP web framework, l'ambiente per questo progetto sarà composto nient'altro che da un classico stack LAMP (Linux Apache2 MySQL PHP). Realizzato l'ambiente, il secondo passo è quello di configurare un repository per il progetto su Github, un software as a service con alla base Git.

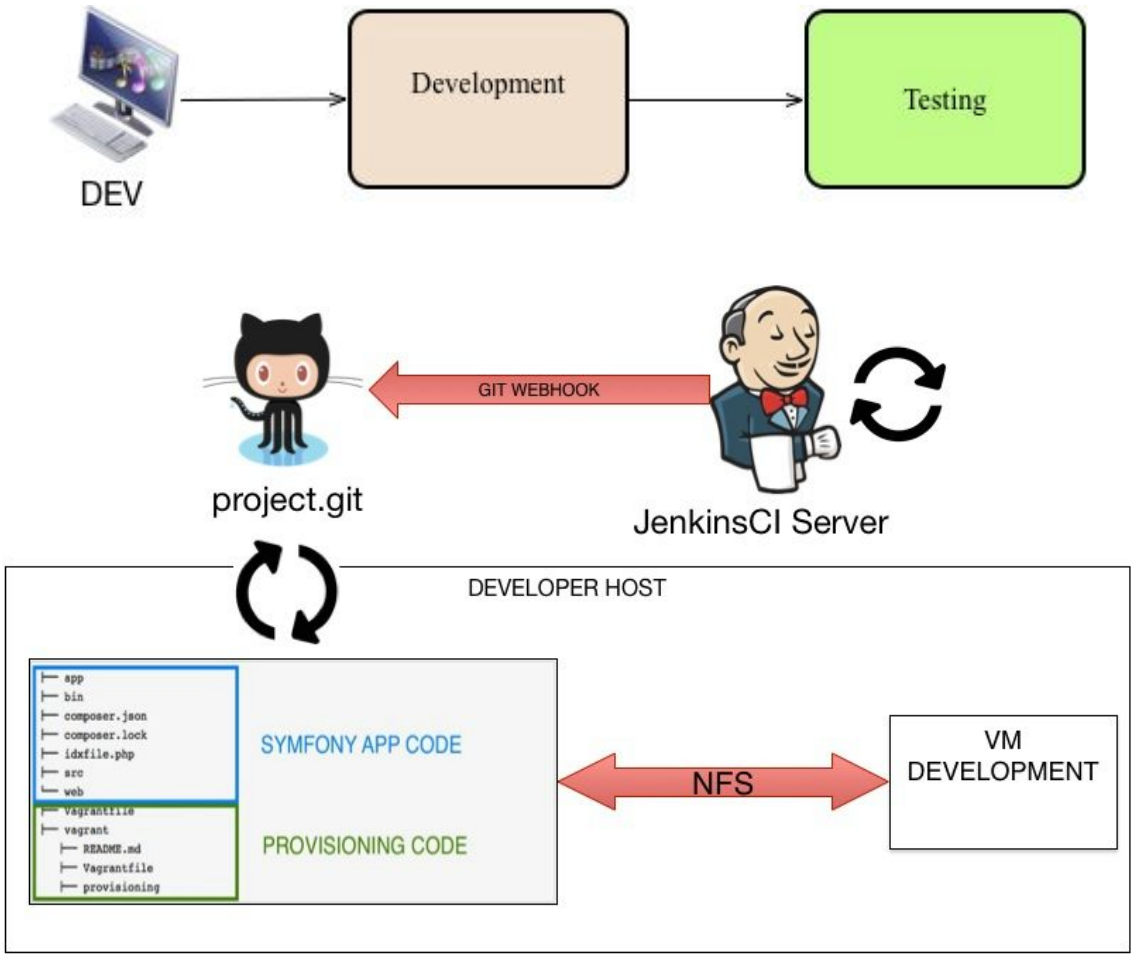


Aggiungendo il CVS come già visto nei precedenti capitoli, si abilitano una serie di feature alla base dello sviluppo software odierno:

- centralizzazione codice
- mantenimento e versionamento del codice
- branch feauture based
- experimental branch
- continuous integration

La Continuous Integration trattata ampiamente nel capitolo legato alle pratiche software, è una pratica già adottata in Ideato, ad ogni modifica applicata al CVS, il server di Continuous Integration lancia una corrispettiva build.

L'avanzamento del DTAP non deve risultare come uno step senza ritorno, ripensando alla deployment pipeline, questo processo verrà eseguito svariate volte durante il ciclo di vita del software.

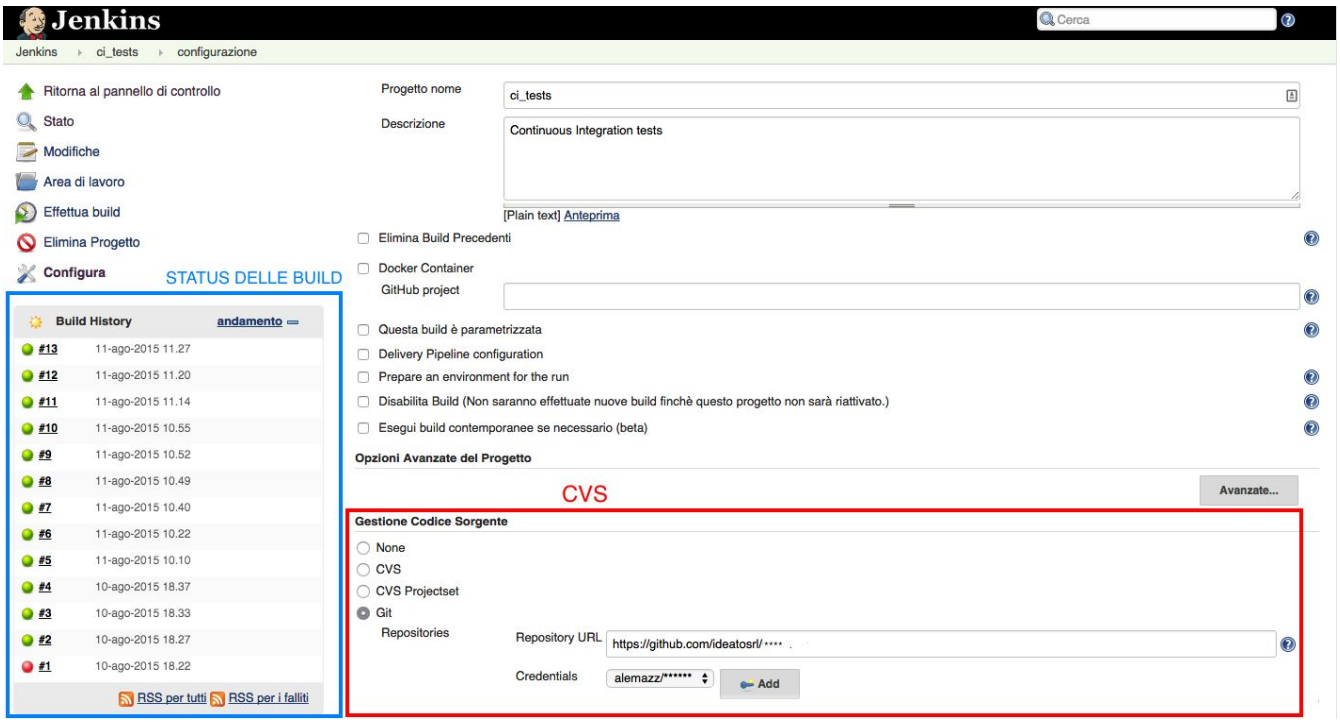


JenkinsCI [39] è uno strumento open source di Continuous Integration, scritto in Java. Jenkins ha principalmente due ruoli:

- **Eseguire script di build o di testing**
  Jenkins fornisce un sistema di integrazione continua, facilitando l'esecuzione di suite di tests e di generazione di build.
- **Monitoraggio di cron jobs**

Essendo il progetto in PHP, verrà utilizzato PHPUnit [40] per lanciare suite di tests, un porting di XUnit per PHP incluso in Idephix. Idephix è uno strumento di automazione PHP utile per eseguire operazioni remote e locali. Può essere utilizzato per distribuire le applicazioni, ruotare i registri e sincronizzare le repository. [41]

Nell'immagine sottostante è rappresentata la fase di testing con un job di Jenkins.
E' facile notare nel riquadro rosso la sorgente del codice e lo storico delle build associate da cui possibile ricavarne il report. Nei due riquadri sottostanti viene mostrato il tipo di evento tale da scatenare l'esecuzione della build, nel prototipo sarà introdotto il web hook, e l'effettivo comando per eseguire la build. Come già detto, Ideato usa Idephix, per incapsulare l'esecuzione della suite di test.



Ideato presenta un'ampia gamma di tecnologie all'interno dei suoi progetti quindi è presumibile realizzare un sistema di Continuous Integration non dedicato solamente al singolo tipo di progetto.

Analizzando Jenkins e la modalità in cui vengono effettuate le suite di tests, vi è un vincolo stretto tra la suite di test e i componenti software necessari da dover rendere disponibili sul server di Continuous Integration. Infatti se un progetto richiede PHP5.5, anche le suite di test relative in esecuzione nel server di Continuous Integration richiederanno la stessa versione.

Una prima soluzione per ovviare all'eterogeneità dei componenti software potrebbe essere quella di rendere disponibili varie versioni pacchettizzate a scapito di mantenibilità.

Una delle modalità alternative, rilasciata insieme al kernel linux sono i containers.

In opposizione alle soluzioni di para virtualizzazione quali XEN o di virtualizzazione hardware come KVM, che permettono l'esecuzione di diverse famiglie di sistemi operativi da Windows a Linux, la virtualizzazione tramite container o containerization crea solamente istanze multiple ed isolate di user space dello stesso kernel.

I containers sono un prodotto maggiormente snello o *lightweight* rispetto alla full virtualization, creati a livello del kernel evitano di dover virtualizzare la parte hardware.

Ricordando la sostanziale differenza tra kernel space e user space nei sistemi operativi UNIX, i containers saranno isolati sia tra loro e in particolare verso l'host.

Tramite i control group introdotti dal kernel 2.6.4 i containers possono essere configurati o limitati in base all'utilizzo delle risorse hardware tra cui CPU, disk I/O e RAM.

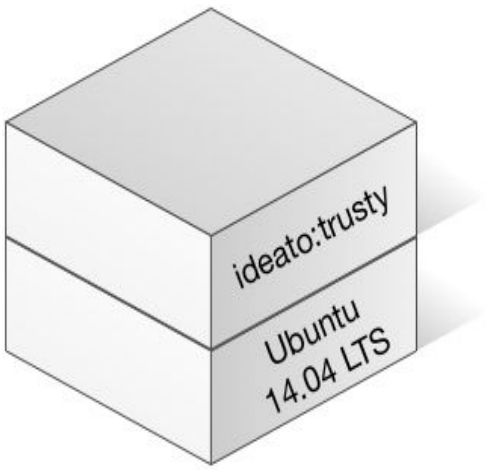
Visto la versalità e il livello di isolamento, rappresentano un caso d'uso perfetto in cui lanciare le suite di tests.

Docker è un progetto open-source basato su container linux che utilizza le stesse funzionalità di isolamento delle risorse del kernel Linux date dai *cgroups* e *namespaces*. Un container in Docker è creato partendo da una immagine read only che definisce cosa il container gestisce,

i dati in esso incapsulato o condiviso e i processi relativi.
Docker sfrutta AUFS (Advanced Union File System), un layered file system che tramite la copy on write rende il processo di creazione estremamente performante. La pecurità di essere a layer abilita anche il poter creare un'immagine a partire da un layer di una precedente immagine. [42]
La soluzione all'eterogeneità dei componenti software è proprio data dalla capacità di Docker di realizzare nuove immagini a partire dalle esistenti.

```
FROM ubuntu:trusty
RUN apt-get update && apt-get install -y --no-install-recommends \
    software-properties-common \
    python-software-properties \
    git \
    vim \
    curl \
    wget

--- dockerfile_trusty_ideato
```

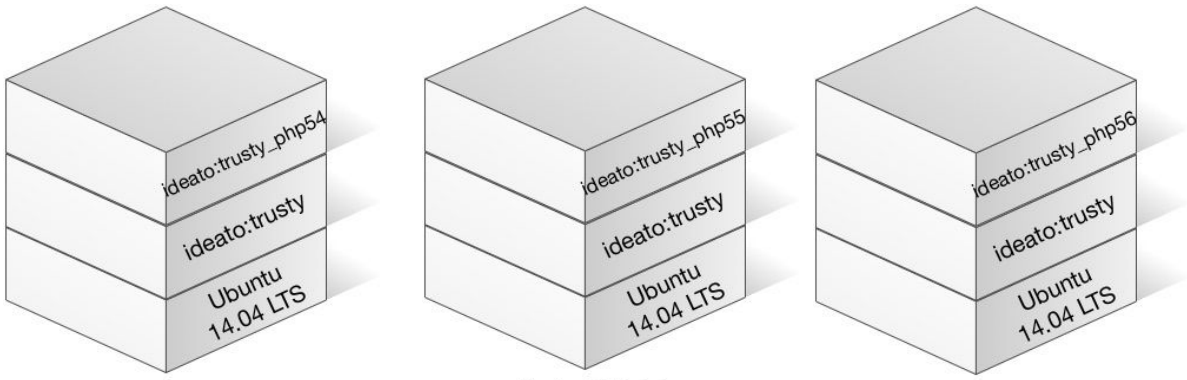
```
docker build -t ideato:trusty -f dockerfile_trusty_ideato .
```



Ideato Base Image
Ubuntu Trusty Jahr 14.04 LTS

```
FROM ideato:trusty
ENV DEBIAN_FRONTEND noninteractive
RUN add-apt-repository -y ppa:ondrej/php5
RUN apt-get install -y  \
        php5 \
        php5-cli \
        php5-dev \
        php5-common \
        php5-mysql \
        php-pear \
        php5-gd \
        php5-curl \
        php5-xdebug \
        php5-mcrypt \
        php5-imagick \
        php5-sqlite \
        php5-xmlrpc \
        php5-intl \
        php5-xsl \
        php5-readline \
        php5-json \
        php5-json \
        libapache2-mod-php5

-- -dockerfile_trusty_ideato_php55 ---
```

Il riquadro evidenziato, mostra l'inserimento di una repository nel package manager di Apt. Le repository sono archivi web nei quali vengono raggruppati i pacchetti software installabili su distribuzioni Debian. Una delle modalità con cui installare le varie versioni di PHP è proprio tramite repository, i vari Dockerfile, file text plain da cui creare le immagini, si distingueranno, nel caso di PHP, solamente dalla repository relativa alla versione.



```
docker build -t ideato:trusty_php54 -f dockerfile_trusty_ideato_php54 .
```

```
docker build -t ideato:trusty_php55 -f dockerfile_trusty_ideato_php55 .
```

```
docker build -t ideato:trusty_php56 -f dockerfile_trusty_ideato_php56 .
```

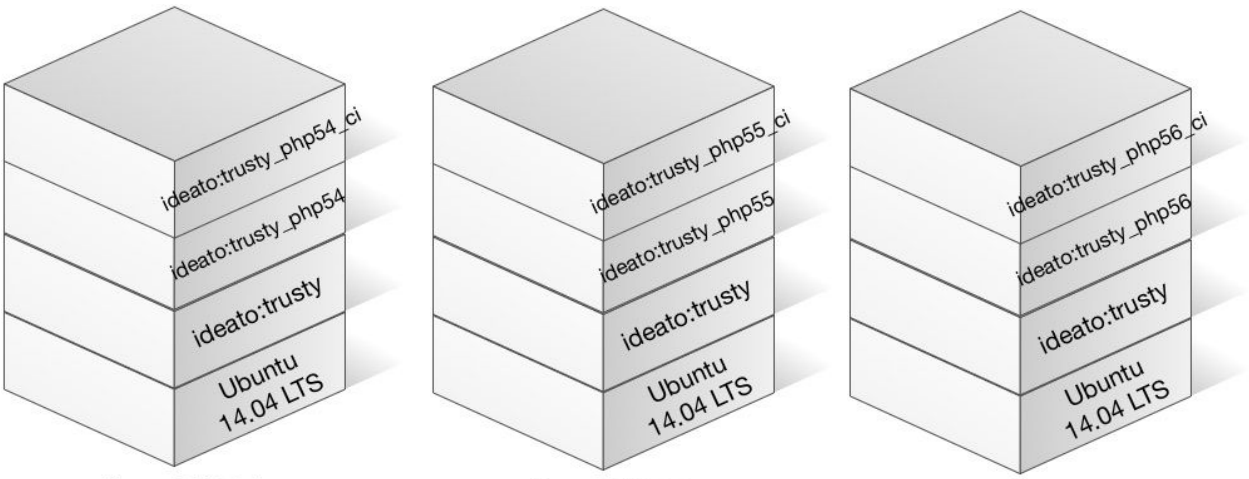
```
FROM ideato:trusty_php55
RUN apt-get install -yq \
        python-dev \
        python-pip \
        openssh-client \
        mysql-server-5.5
RUN pip install ansible
RUN mkdir -p /etc/ansible
ADD vault.txt /etc/ansible/
RUN wget https://phar.phpunit.de/phpunit.phar
RUN chmod +x phpunit.phar
RUN mv phpunit.phar /usr/local/bin/phpunit

--- dockerfile_trusty_ideato_php55_ci_testing ---
```

```
docker build -t ideato:trusty_php54_ci -f dockerfile_trusty_ideato_php54_ci_testing .
```

```
docker build -t ideato:trusty_php55_ci -f dockerfile_trusty_ideato_php55_ci_testing .
```

```
docker build -t ideato:trusty_php56_ci -f dockerfile_trusty_ideato_php56_ci_testing .
```

Queste immagini aggiungono funzionalità e software aggiuntivi per eseguire le suite di tests all'interno del server di Continuous Integration.



Ideato PHP 5.4
Continuous Integration Image

Ideato PHP 5.5
Continuous Integration Image

Ideato PHP 5.6
Continuous Integration Image

Il possibile workflow per poter sfruttare appieno il server di Continuous Integration è quello di definire i software aggiuntivi all'interno di un Dockerfile specifico per progetto realizzato al top delle immagini specifiche per il server di Continuous Integration.
Il Dockerfile per questo specifico progetto, oltre a PHP5.6 richiederà anche Elasticsearch, [43] un motore di ricerca con capacità full text per applicazioni distribuite basato su Apache Lucene e NodeJS. [44]
Senza le funzionalità di Docker, il responsabile operativo avrebbe dovuto installare manualmente i due componenti nel server di Continuous Integration e valutarne i conflitti con le versioni pre installate necessari ad altri progetti.
Introducendo i containers, i componenti aggiuntivi in questo caso Elasticsearch e NodeJS, vengono installati in un ambiente isolato, senza alcun intervento dei tecnici e senza comprometterne la stabilità dell'host.

Il Dockerfile, come ultima azione, esegue un script aggiuntivo che:

- metterà in esecuzione i servizi richiesti
- effettuerà dei test sintattici sui i playbook Ansible che in seguito verranno utilizzati per il provisioning verso gli altri ambienti
- eseguirà le suite di tests relativi all'applicazione.

```
FROM ideato:trusty_php56_ci
ENV DB default
RUN apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 4F4EA0AAE5267A6C
RUN apt-get clean && apt-get update && apt-get install -yq \
    npm \
    openjdk-7-jre \
    openjdk-7-jdk \
    nodejs
RUN wget https://download.elastic.co/elasticsearch/elasticsearch/elasticsearch-1.4.2.deb
RUN dpkg -i elasticsearch-1.4.2.deb
ADD bin/run_tests /
ENTRYPOINT ["./run_tests"]

--- Dockerfile ---

#!/bin/bash
cd /ehoreca/workspace
service mysql start
mysql -uroot -e "create database $DB"
service elasticsearch start
ansible-playbook -i vagrant/provisioning/hosts.ini \
--vault-password-file /etc/ansible/vault.txt \
--extra-vars "target=development target_user=root target_vars=host_vars/development.yml" \
$vagrant/provisioning/playbooks.yml --syntax-check
bin/idx build

--- run_tests ---
```

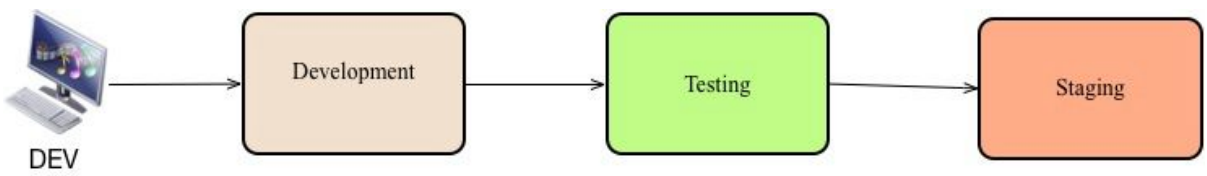
ANSIBLE SYNTAX TEST

IDEPHIX BUILD

Nell'immagine sottostante, presa da Jenkins, sono visualizzati i tre comandi che rappresenteranno la build dell'applicazione:

- installazione dei componenti necessari all'applicazione tramite Composer
- creazione dell'immagine tramite dockerfile incluso nel CVS
- lancio del container

Ad ogni esecuzione della build verrà realizzata l'immagine relativa del progetto.
Questo passo, in scenari di multijobs concorrenti e lancio frequente di tests, non risulta comunque oneroso, Docker infatti applicando politiche di caching ad ogni layer dell' immagine, non esegue ex-novo la creazione dell'immagine.
Al termine della suite di tests il container viene eliminato, liberando le risorse occupate.



La transizione dallo step di Development & Testing a Staging risulta una delle parti maggiormente complesse nella progettazione di questo prototipo.

La Continuous Delivery che è parte integrante e alla base del prototipo di Infrastructure as Code, si concentra sulla riduzione degli sforzi per portare, da questo "deliver", il software in produzione.
Riprendendo due delle best practices per realizzare una deployment pipeline funzionale:

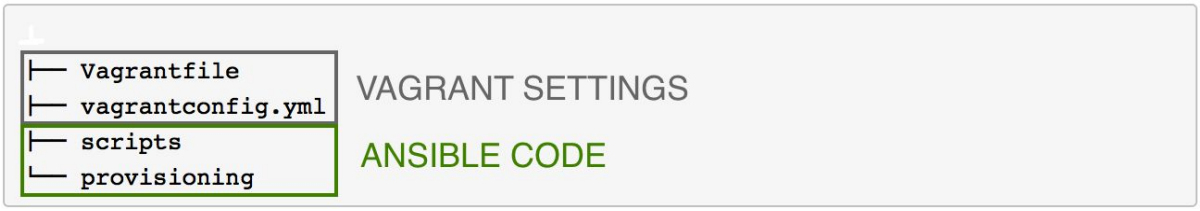
- **Stessa modalità di deployment in ogni ambiente**, i vari ambienti di sviluppo seguendo l'approccio DTAP, differiscono per alcuni fattori quali:
    - politiche di sicurezza (password strength, user jail...)
    - ottimizzazioni
    - caratteristiche fisiche (spazio disco, RAM, CPU)
    - caratteristiche di networking (IP, MAC)

  Il processo di deployment dovrebbe permettere l'impostazione di questi parametri ai vari ambienti attraverso strumenti di version control e di gestione credenziali(LDAP, ENC o via CLI).

- **Deploy in ambienti simil produzione**, uno dei problemi che causano maggior problemi durante il processo di sviluppo software è lo scostamento tra ambiente di test/staging rispetto quello di produzione. Per far sì che gli sviluppatori abbiano un discreto livello di confidenza, i test che verranno eseguiti durante la deployment pipeline, dovranno essere rivolti verso ambienti simil produzione o production-like. Ottenere copie esatte dell'ambiente di produzione può risultare complesso, infatti bisogna considerare un numero elevato di componenti e variabili:
    - Infrastruttura (network, firewall..)
    - sistema operativo (distribution, release, kernel version)
    - Stack applicativo (webserver, database, file shared versione e configurazione)

Per assolverle, il codice di provisioning che crea l'ambiente di development deve differire a meno di variabili, politiche di security ed ottimizzazioni dall'ambiente di produzione. Durante lo step di development è stato analizzato lo stack Vagrant e il template custom, tuttavia quest'ultimo è adatto per ora solo ad un ambiente di sviluppo.

La prima parte di attività legate alla realizzazione del prototipo si concentrano sulla ridefinizione del template in modo da creare un flusso fra diversi step della deployment pipeline. Il template è un progetto già avviato in Ideato con lo scopo di automatizzare la creazione di uno stack LAMP volto allo sviluppo web sfruttando Vagrant e Ansible.



Al comando "vagrant up", Vagrant:

1. effettua il parsing del Vagrantfile contenenti le configurazioni
2. interagendo con VirtualBox:
    a. recupera dalla rete o dal filesystem locale un immagine pacchettizzata del SO
    b. configura RAM, spazio disco e networking
    c. condivide via NFS la cartella locale di sviluppo con la macchina virtuale
3. eseguerà gli script relativi ad Ansible
    a. installa Ansible all'interno della macchina virtuale
    b. lancia internamente alla VM il playbook contenente la definizione dei servizi LAMP

Le attività legate a questa prima fase saranno riverse verso il codice Ansible del template nel valutare e abilitare la definizione dei servizi anche verso i step di staging e produzione.

Le due tematiche riscontrate durante l'analisi e la progettazione sono state le seguenti:

- etereogeneità delle distribuzioni software (Debian, RedHat)
- separazione degli ambienti avendo facoltà di apporre regole di security ed ottimizzazione dei servizi

- mantenere coerente e funzionante la creazione dell'ambiente di development per non interrompere l'attuale flusso.

Seppur ultima elencata, mantenere e coerente la versione attuale del template è un'esigenza assoluta; creare disagio o introdurre bug nella versione attuale, porta ad un incremento dei tempi di rilascio e ad un conseguente spreco.
La soluzione è stata sfruttare la potenzialità del CVS per realizzare un branch detto experimental o di testing in cui fosse possibile applicare modifiche senza compromettere la versione corrente nel branch detta di master. Una volta testata e resa funzionante la versione sperimentale è stata applicata a quella corrente tramite il processo di merge.

I file di configurazione di Ansible sono in formato YAML detti Playbooks incapsulati in elementi top level detti Ansible roles.
L'eterogeneità delle distribuzioni può essere gestita in due possibili modi:

- realizzare duplici Ansible roles, ognuno per ciascuna distribuzione
  (es. ideato.common.apache2.debian ideato.common.apache2.redhat)
- strutturare i playbooks per gestire entrambe le distribuzioni

La scelta è ricaduta sulla seconda per varie motivazioni:

- duplicazione del codice: creare un role significa duplicare il codice contenuto nel playbook compreso anche del layout del role stesso
- mantenimento: i roles essendo gestiti all'interno del progetto e non come git submodules il mantenimento per ciascuna distribuzione risulterebbe complicato e renderebbe il progetto estremamente fragile.

Il template avrà quindi solo tre moduli Ansible contenenti sia le specifiche per Debian che RedHat coerenti con lo stack LAMP :

- ideato.common, role generico relativo al sistema operativo
  (pacchetti di sistema, ottimizzazioni, security)
- ideato.webserver, role relativo a PHP e Apache2
- ideato.database.mysql, role relativo a MySQL

Nel codice sottostante, vengono sfruttati i facts, ottenibili da Ansible, riguardanti le caratteristiche specifiche dell'ambiente. Verranno definiti tramite essi, servizi specifici per

ciascuna distribuzione. Come accennato in precedenza Ansible non possiede come Puppet un layer per astrarre dall'effettiva implementazione delle risorse, è presente il modulo apt per distribuzioni Debian e yum per distribuzioni RedHat.

```
- name: include OS-specific variables
  include_vars: "{{ ansible_os_family }}.yml"
-
 name: CentOS Package requirements
 yum: name={{ ideato_common_packages }} state=present update_cache=yes
 when: ansible_distribution == 'CentOS'
-
 name: Ubuntu Package requirements
 apt: name={{ ideato_common_packages }} state=present update_cache=yes
 when: ansible_distribution == 'Ubuntu'
```

facts
variables
conditionals
providers

— prod.yml —

```
---
ideato_webserver_packages:
  - apache2
  - php5
  - libapache2-mod-php5
  - php5-cli
  - php5-dev
  - php5-mysql
  - php-pear
  - php5-mcrypt
  - php5-gd
  - php5-curl
  - php5-xdebug
  - php5-readline
  - php5-sqlite
  - php5-common
  - php5-dev
  - php5-imagick
  - php5-intl
  - php5-xmlrpc

-- Debian.yml --
```

La separazione degli ambienti (development, staging, production) seguirà lo stesso approccio suddividendoli in playbooks differenti: dev.yml e prod.yml. La scelta di non definire anche un playbook per l'ambiente di staging, è dovuta alla best practice di tendere verso ambienti production-like.

```
---
- include: dev.yml
  when: target == "development"
- include: prod.yml
  when: target == "production" or target == "staging"

-- main.yml --
```

```
---
- hosts: '{{ target }}'

  user: '{{ target_user }}'

  vars_files:
    - '{{ target_vars }}'

  gather_facts: yes

  roles:
    - ideato.common
    - ideato.webserver
    - ideato.database.mysql

  sudo: True

-- playbooks.yml --
```

Grazie alla ridefinizione sia del playbook principale sia dei ruoli relativi ai servizi, ora è possibile effettuare il provisioning dei servizi non solo verso l'ambiente di sviluppo ma anche verso i successivi ambienti, fondando le basi per futuri sviluppi legati alla deployment pipeline e alla Continuous Delivery. Il codice sviluppato nel branch sperimentale è stato applicato master dopo un'opportuna fase di testing sfruttando ambienti virtualizzati Vagrant ed è fruibile al seguente URL: https://github.com/ideatosrl/vagrant-php-template.

Una possibile esecuzione:

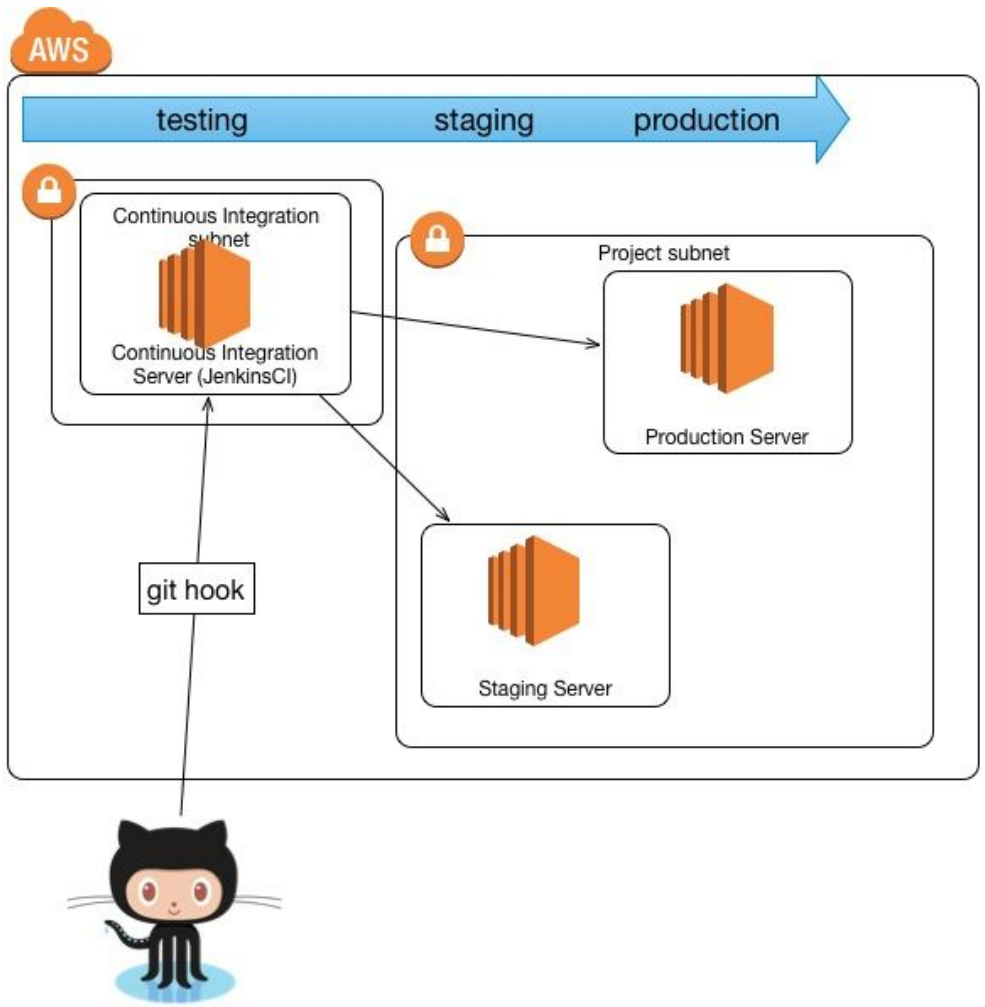
```
ansible-playbook --become-user=root -i vagrant/provisioning/hosts.ini
--vault-password-file /etc/ansible/vault.txt vagrant/provisioning/playbooks.yml
 --extra-vars "target=staging target_user=root target_vars=host_vars/staging.yml"
```

La seconda parte del protipo riguarda la realizzazione di una deployment pipeline fra i vari ambienti, eliminando ogni possible spreco rapportandosi al contesto lavorativo.

La deployment pipeline sarà realizzata tramite Jenkins fornendo una serie di vantaggi:

- facile setup (jobs, notifiche, webhook, plugins)
- aggiunta di trigger in caso di successo o fallimento
- possibilità di lanciare sia build in modalità automatica o manuale
- GUI facilmente consultabile e visualizzabile su altri supporti

Su Jenkins un job rappresenta l'esecuzione di uno o più script che possono essere ad esempio quelli di build o semplici comandi batch ripetivi. Vedremo come un job, nel prototipo rappresenti l'insieme delle procedure che verranno lanciate verso un particolare ambiente.
La relazione tra job e ambiente nel prototipo infatti sarà univoca, la deployment pipeline risulterà una catena di job opportunamente configurati.



Dopo aver configurato il template, è stata realizzata un' ipotetica infrastruttura su Amazon Web Services. [45]
L'infrastruttura composta da:

- un'istanza EC2 c4.xlarge (4 core, 8 GB RAM,) con Jenkins come server di Continuous Integration.
- due istanze EC2 t2.small (1 core, 1GB RAM) per simulare gli ambienti di staging e produzione.

Il pattern DTAP sarà mappato interamente dentro Jenkins sfruttando il plugin "Build Pipeline".

Il flusso può essere sintetizzato nei seguenti passi:

1. il developer effettua una commit su GitHub, scelto come CVS
2. tramite webhook il CVS notifica una modifica avvenuta al server di Continuous Integration
3. il server di Continuos Integration ricevuta la notifica scatena l'evento del job relativo ottenendo la nuova versione della repository e lanciando i comandi della build
4. Jenkins esegue il job per lo step di testing
5. Jenkins notifica il risultato al team di progetto

6. in caso di successo, automaticamente Jenkins esegue il job per lo step di staging notificando il risultato al team di progetto

7. in caso di successo, il team di sviluppo ricevuto la notifica, avvierà manualmente lo step di production

CONCLUSIONI

L'avvento di nuove tecnologie come virtualizzazione, Cloud Computing e containers hanno trasformato le infrastrutture in un agglomerato di software e dati.
Questo ha permesso di poter trattare le infrastrutture come fossero un sistema software adottando alcune delle pratiche di Ingegneria del software per renderla dinamica.

L'Infrastructure as Code attenua il netto divario tra system administrator e developer, incoraggiando il primo ad una maggior comprensione verso metodologie e pratiche software al fine di risolvere alcune problematiche del passato.

Il primo passo verso un'infrastruttura dinamica è stato attraverso la raccolta dei configuration items e lo studio di un'adeguata strategia di Configuration Management in grado di rispondere ad una serie di scenari a cui un'infrastruttura moderna è sottoposta: riproduzione di tutti gli aspetti degli ambienti, flessibilità nel supportare modifiche o ottenere uno storico dei cambiamenti apportati.

La tesi, delineata la strategia ha affrontato i sistemi di Configuration Management ed in particolare Git come strumento tramite cui mantenere e versionare il codice.
Git ha mostrato un'estrema flessibilità grazie al forte supporto allo sviluppo non lineare, volto sia alla delineazione degli ambienti di sviluppo in accordo al pattern DTAP sia al testing di nuove configurazioni tramite i branch cosidetti sperimentali.

Nel capitolo successivo sono stati trattati due strumenti quali Puppet e Ansible in grado di definire e gestire risorse e servizi tramite codice, analizzandone pregi e difetti verso una possibile applicazione al contesto aziendale.

La tesi ha introdotto due tecniche software dell'Extreme Programming: la Continuous Integration e la Continuous Delivery.
La Continuous Integration ha soddisfatto uno dei requisiti previsti dalla strategia di Configuration Management cui sviluppata una modifica possa essere velocemente testata rendendo un corretto feedback agli addetti.
Come spiegato uno dei benefici legati alla Continuous Integration è la riduzione dei rischi con la conseguente acquisizione di una maggior confidenza verso il prodotto software sviluppato.
Uno degli obiettivi dell' Infrastructure as Code è proprio l'instaurazione di un clima di fiducia verso la propria infrastruttura potentovi applicare modifiche incrementali seguiti da tests

attenuando così il rischio di creare imprevedibili effetti a catena.
La Continuous Integration è un enorme passo in termini di produzione e di qualità del software, la Continuous Delivery garantisce che l'intero sistema sia pronto per essere in produzione.
La Continuous Delivery permette l'abolizione o per lo meno la riduzione degli sprechi e dei ritardi durante il rilascio del software. Mostrata una ipotetica value stream map riconducibile al processo di sviluppo software, gli sprechi o ritardi sono di norma generati durante l'interazione con il team operativo riconducibili prevalentemente ad una mancanza di Configuration Management e di automatizzazione.

L'obiettivo finale della tesi consisteva nell'applicazione dei principi cardini dell' Infrastructure as Code ad un progetto Symfony in corso sfruttando le pratiche e metodologie viste durante l'intera trattazione del pattern.

Come per i concetti teorici, la realizzazione del prototipo ha avuto come primo step l'individuazione degli ambienti e le modalità di sviluppo.
La delineazione degli ambienti di sviluppo interni all'azienda ha coinciso in parte con il pattern DTAP individuando ambiente di sviluppo (development), di staging e di production.
Un aspetto critico rilevato all'interno dell'azienda è stata la disomogeneità generata da un'applicazione parziale dei strumenti di Configuration Management.
Gli sforzi per attenuare questa problematica sono stati duplici risolti tramite l'utilizzo di Ansible come un unico strumento di Configuration Management e la possibilità di effettuare il provisioning dei servizi verso anche gli ambienti successivi.
Una parte del prototipo è valsa ad attenuare questa differenza riscrivendo il template di Ansible per apportare la strategia di Configuration Management anche negli step successivi.

La seconda parte del prototipo si è concentrata sull'automatizzazione del rilascio del software tramite l'implementazione di una deployment pipeline nell'ordine di ridurre i tempi e gli sprechi. La deployment pipeline, nodo centrale della Continuous Delivery, ha visto come primo step la realizzazione di un sistema di Continuous Integration tramite JenkinsCI per poi creare una pipeline che relazionasse i vari ambienti.
In fase di progettazione é stato rilevato un secondo set di problematiche tra le quali mancanza di un'unica modalità di deployment ed una etereogeneità dei componenti software.
La prima è stata risolta tramite una combinazione di Idephix ed Ansible e la seconda tramite Docker.

Unendo quindi tecnologie come Cloud Computing, Version Control System e strumenti di Configuration Management è stato possibile ridurre il tempo di setup degli ambienti automatizzandoli in una deployment pipeline.
Il prototipo realizza pienamente i concetti della Continuous Delivery e della Lean Production, rendendo, grazie alla riscrittura del template usato in fase di sviluppo, il deploy verso i vari ambienti un "non evento" attenuando sensibilmente i tempi di attesa e gli sprechi.
Il prototipo garantisce agli sviluppatori la facoltà, dopo aver effettuato tests sia sul codice applicativo che sul codice infrastrutturale, di poter effettuare il setup degli ambienti, il deploy dell'applicazione o di apportare modifiche in completa autonomia senza dover chiedere l'assistenza dal personale operativo riprendendo anche concetti appartententi al fenomeno DevOps.

Il prototipo tuttavia non rende possibile l'abolizione totale degli sprechi legati al setup degli ambienti, il provisioning del sistema operativo, richiede tuttora l'assistenza del personale operativo.
Questa particolare funzionalità è stata esclusa dal prototipo principalmente per mancanza di direttive legate alla locazione degli ambienti e di tempo in vista della prova finale.
Un futuro sviluppo sarà quello di integrare concetti quali di dynamic inventory e di service discovery per aumentare il grado di automatizzazione e di metodologie legate al test coverage dell'infrastruttura quali TDD (Test Driven Development) e TDI (Test Driven Infrastructure).
L'azienda valuterà se il prototipo sarà applicabile al workflow esistente, estendendo o meno le sue funzionalità.

BIBLIOGRAFIA

[1] Definizione virtual machine:
Andrew S.Tanenbaum Herbert Bos, *Modern Operating system 4rd edition,* 2014, Pearson

[2] Definizione di Cloud Computing:
http://www.enterthecloud.it/abc-cloud/nist-ue/

[3] Iron Age e Cloud Era:
Kief Morris, Infrastructure as Code, 2014, O'Reilly, p. 3

[4] Principi cardine di Infrastructure:
Stephen Nelson Smith, *Test-Driven Infrastructure with Chef*, 2013, O'Reilly, p.2

[5] Nuove tecnologie, vecchie abitudini:
Kief Morris, Infrastructure as Code, 2014, O'Reilly, p. 1-2

[6] Snowflake server:
Kief Morris, Infrastructure as Code, 2014, O'Reilly, p. 7

[7] Erosione:
Kief Morris, Infrastructure as Code, 2014, O'Reilly, p. 9
https://www.heroku.com/

[8] Definizione di Infrastructure as Code:
Kief Morris, Infrastructure as Code, 2014, O'Reilly, p. 5

[9] Principi di Infrastructure as Code:
Kief Morris, Infrastructure as Code, 2014, O'Reilly, p. 12-19

[10] ITIL configuration management:
http://www.itil-italia.com/itilconfigmanagement.htm

[11] Strategia di configuration management applicato allo sviluppo web e alle infrastrutture:
Jez Humble, David Farley, Continuous Delivery, Reliable software releases through build, test and deployment automation, 2010, Addison Wesley, p. 31-32

[12] Definizione, cenni storici e tipi di Version Control System:
https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

[13] Introduzione a Git:
https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git

[14] Basi di Git:
https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

[15] Gestione delle dipendenze:
Jez Humble, David Farley, Continuous Delivery, Reliable software releases through build, test and deployment automation, 2010, Addison Wesley, p. 38-39

[16] DTAP:
http://blog.heyunka.com/2012/08/dtap-in-a-lean-and-agile-environment/
http://creativechances.blogspot.it/2013/07/dtap-route-to-application-lifecycle.html
http://www.phparch.com/2009/07/professional-programming-dtap-%E2%80%93-part-1-what-is-dtap/

[17] Push mode vs Pull mode technologies:
Kief Morris, Infrastructure as Code, 2014, O'Reilly, p. 103-104

[18] Introduzione a Puppet:
James Turnbull, Jeffrey McCune, Pro Puppet, 2011, Apress, p. 1

[19] Il modello di Puppet:
James Turnbull, Jeffrey McCune, Pro Puppet, 2011, Apress, p. 2-6
https://docs.puppetlabs.com/guides/introduction.html

[20] Puppet Compiler:
http://www.masterzen.fr/2012/03/17/puppet-internals-the-compiler/
https://docs.puppetlabs.com/guides/puppet_internals.html

[21] Puppet Parser:
http://www.masterzen.fr/2011/12/27/puppet-internals-the-parser/
https://docs.puppetlabs.com/guides/puppet_internals.html

[22] Introduzione ad Ansible:
https://docs.ansible.com/ansible/intro_getting_started.html

[23] Inventory:
https://docs.ansible.com/ansible/intro_inventory.html

[24] Built in modules:
https://docs.ansible.com/ansible/modules.html

[25] Ad-hoc commands:
http://docs.ansible.com/ansible/intro_adhoc.html

[26] Playbooks:
http://docs.ansible.com/ansible/playbooks.html

[27] Roles:
http://docs.ansible.com/ansible/playbooks_best_practices.html

[28] Introduzione alla Continuous Integration e Integration Hell:
https://dzone.com/articles/continuous-integration-how-0

[29] Pratiche di Continuous Integration:
Paul M. Duvall, Steve Matyas, Andrew Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (Addison-Wesley Signature Series)*, 2007, Addison-Wesley, p. 39-43

[30] Valori della Continuous Integration:
Paul M. Duvall, Steve Matyas, Andrew Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (Addison-Wesley Signature Series)*, 2007, Addison-Wesley, p. 29-32

[31] Integrate Button:
Paul M. Duvall, Steve Matyas, Andrew Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (Addison-Wesley Signature Series)*, 2007, Addison-Wesley, p. 13

[32] Continuous Integration stack:
Paul M. Duvall, Steve Matyas, Andrew Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (Addison-Wesley Signature Series)*, 2007, Addison-Wesley, p. 8-20

[33] Continuous Delivery, anatomia e step della Delivery Pipeline
Jez Humble, David Farley, Continuous Delivery, Reliable software releases through build, test and deployment automation, 2010, Addison Wesley, p. 105-112

[34] Ideato:
https://www.ideato.it

[35] Vagrant:
https://www.vagrantup.com/

[36] VirtualBox:
https://www.virtualbox.org/

[37] Vmware:
http://www.vmware.com/

[38] Symfony:
https://symfony.com/

[39] JenkinsCI:
https://jenkins-ci.org/

[40] PHPUnit:
https://phpunit.de/

[41] Idephix:
http://getidephix.com/

[42] Docker:
https://www.docker.com/

[43] Elasticsearch:
https://www.elastic.co/products/elasticsearch

[44] NodeJS:
https://nodejs.org/en/

[45] Amazon Web Services:
https://aws.amazon.com/it/