Second way: enables fast and constant flows of feedback from right to left at all stages of our value stream, amplify feedback to prevent problems from happening again or enable faster detection and recovery

Third way: creation of a generative, high trust culture that supports a dynamic, disciplined and scientific approach to experimentation and risk taking

FIRST WAY: PRINCIPLES OF FLOW

Requires the fast and smooth flow of work from Development to Operations, to deliver value to customers quickly. We increase flow by:

* Make our work visible: differenza rispetto processi fisici perché nel technology vaue stream our work is invisible. To help us see where work is flowing well and where work is queued or stalled we need to make our work visible, one of the best methods of doing this is using visual work boards such as kanban boards, work originates to the left (pulled from a backlog) and then is pulled from work center to work center (utilizzo di cartellini). Work is completed when it reaches the right side of the board, stakeholders can more easily prioritize work in the context of global goals (figura)
* Limitare il work in process (WIP): in ambito industriale, la produzione giornaliera è solitamente stabilita da una schedulazione che viene pianificata a cadenze regolari (giornalmente, settimanalmente); in ambito tecnologico però il ritmo di lavoro è molto più dinamico. Una limitazione del WIP si rivela necessaria per incrementare la produttività a livello di singolo e di gruppo, rende più facile la stima del lead time per un determinato ticket e soprattutto è utile per individuare eventuali colli di bottiglia. Una delle principali soluzioni proposte dai principali leaders DevOps è quella di limitare il numero di cartellini in una certa area della kanban board per evitare il sovraccarico per una certa area di lavoro.
* Ridurre le dimensioni del carico di lavoro: è tra le più importanti prerogative del Lean Thinking. In un contesto manifatturiero lotti di grandi dimensione portano a un considerevole aumento del WIP e soprattutto a un alto livello di variabilità nella gestione del flusso di lavoro. Tali considerazioni possono essere traslate e applicate allo stesso modo nel Mondo IT e per questo è importante tenere a mente una delle lezioni più importanti che il Lean Thinking ci ha insegnato, ovvero che per ottenere brevi lead time e prodotti di qualità è necessario ridurre il più possibile il carico di lavoro avvicinandosi il più possibile al miraggio del single-piece-flow (figura con esempio). Nel contesto software ciò che può assumere il ruolo di un lotto (e quindi carico di lavoro) in una tipica catena di montaggio sono le righe di codice e quindi l’equivalente del single-piece-flow in ambito tecnologico si raggiunge attraverso il cosiddetto “continuos deployment”, effettuando commit frequenti e di piccole dimensioni verso un version control system (non prima di aver testato e integrato le righe di codice con il resto del lavoro)

Quattro aree da ottimizzare per attuare una trasformazione del nostro apparato IT secondo i principi DevOps:

* Creazione dell’ambiente (environment creation)
* Sviluppo e deployment del codice
* Impostazione ed esecuzione dei test
* Architettura di lavoro

The First Way: The Technical Practices of Flow

In ambito di produzione software mettere in pratica questi concetti di produzione snella è delicato. Bisogna infatti evitare una disruption troppo aggressiva che avrebbe l’effetto opposto. Il mondo DevOps viene incontro a questa necessità proponendo e implementando una serie di soluzioni che vanno a inserirsi nella macro-categoria del continuos delivery. Ho scelto di presentare le quattro pratiche che ho ritenuto più importanti e complementari al fine di avere un risultato soddisfacente.

1. Creare le fondamenta della cosiddetta deployment pipeline:  
   L’obiettivo di partenza è quello di ottenere un production-like environments in ogni stage del processo, creati automaticamente tramite script o informazioni di configurazione presenti in qualche repository del version control system utilizzato e senza quindi la necessità di operazioni manuali che rallenterebbero un lavoro di per sé non eccessivamente complicato e ripetitivo. Fornire a uno sviluppatore tutti gli ambienti di cui ha bisogno in tempo breve permette al developer di lavorare a 360 gradi sul prodotto anche a livello giornaliero: se l’ambiente per il testing è a disposizione degli sviluppatori dall’inizio è chiaro che essi potranno testare giornalmente il loro lavoro e risolvere sul nascere eventuali problemi, evitando di trovarsi con grosse parti non funzionanti in là con il tempo. Ciò che si vuole fare è quindi creare un build mechanism che crei tutti gli ambienti (sviluppo, test, produzione) velocemente e sin da subito, senza la necessità di doverli richiedere aprendo un ticket che può far perdere anche settimane di lavoro.  
   Queste richieste non possono essere soddisfatte mediante operazioni manuali di sistemisti, in tal caso sarebbe impossibile garantire il pacchetto completo sin da subito e il rischio di errore sarebbe alto, oltre a occupare del personale in operazioni ripetitive e non stimolanti. Bisogna ricorrere a un processo automatizzato per la creazione dei vari ambienti. Attraverso tools che verranno presentati nel dettaglio nel Capitolo 3 è possibile automatizzare una serie di procedimenti sistemistici velocizzando notevolmente le operazioni:  
   - Copiare un ambiente virtualizzato (tramite VMware o lanciando uno script Vagrant)  
   - Applicare il paradigma “infrastructure as a code” tramite i configuration management tools Puppet, Chef, Ansible  
   - Creare ambienti virtuali e farne il provisioning secondo le nostre necessità (tramite Vagrant)   
   Questo tipo di soluzione permette anche di sperimentare soluzioni differenti nell’infrastruttura alla base degli ambienti: infatti, una volta che è quasi tutto controllato tramite script, ogni modifica e sperimentazione risulta piuttosto veloce.  
   E’ importante precisare che questo nuovo modo di operare si riflette anche su altri elementi del progetto, come per esempio la questione del version control system: sarà infatti fondamentale versionare e conservare in un repository centrale non solo il codice applicativo, ma anche gli script che servono a costruire gli ambienti, in tal modo rendiamo la configurazione di cui sopra parlato facilmente reperibile, controllabile e riutilizzabile nel caso in cui ci fosse la necessità di recuperare una vecchia versione del codice o della configurazione degli ambienti.  
   Applicando queste best-practice è naturale che uno sviluppatore possa impostare diversamente il lavoro sin dall’inizio: molte delle moderne tecniche di sviluppo software si basano su un approccio iterativo applicato a una ridotta mole di lavoro rispetto al datato approccio a cascata visto precedentemente (*o presento prima il modello a cascata o lo presento ora, probabilmente meglio ora con slide Agile/Scrum*).  
   L’obiettivo è assicurare che la parte di sviluppo e di QA (quality assurance) siano continuamente a contatto: il lavoro prodotto a fine di ogni iterazione deve essere interamente funzionante in un production-like environment (*valuta di dare prima una definizione di production-like environment*), non è più sufficiente che lo sviluppatore si limiti alla mera operazione di lanciare con successo il codice sul suo computer. Alla fine del progetto ci saranno state migliaia di “deploy” del codice in un production-like-environment e si ha quindi che la maggior parte dei problemi siano già stati individuati e risolti.
2. Costruire un sistema veloce e affidabile di “Automated Testing”  
   Senza un sistema automatizzato di testing, più codice viene scritto e più tempo e soldi sono richiesti per testare il codice, e considerando la mole delle applicazioni che vengono sviluppate oggigiorno è chiaro che un approccio simile rappresenti per l’azienda un “unscalable business model” destinato a collassare.  
   (*valutare esempio Google, molto esplicativo*). Il primo obiettivo è quello di permettere agli sviluppatori di poter usufruire di test automatici giornalmente in modo da avere un immediato feedback del loro lavoro e poter risolvere immediatamente eventuali problemi. Per raggiungere invece un risultato a 360 gradi bisogna creare una “test suite” che incrementa la frequenza dei test da periodica a continua procedendo nella costruzione di una deployment pipeline che assicuri che ogni cambiamento inserito in version control venga lanciato e testato nel production-like environment (*figura di una possibile deployment pipeline con varie tipologie di test*). Tra i più conosciuti e utilizzati tools per creare una deployment pipeline c’è Jenkins, che verrà ampiamente descritto nei prossimi capitoli. Poter usufruire di una deployment pipeline funzionale è fondamentale per tutti i membri del team in quanto fornisce una visione d’insieme che altrimenti sarebbe impossibile avere e permette di controllare facilmente come procede il lavoro individuando facilmente, nel caso in cui ci siano dei problemi, il punto critico all’interno della catena. È evidente quanto tutto ciò semplifichi il lavoro e lo renda incredibilmente più produttivo.  
   (*esempio differenza frequenza periodica vs frequenza continua*).  
   La macro categoria degli “automated tests” si suddivide, all’interno della catena, in diverse sottocategorie, dalla più veloce alla più lenta da eseguire:  
   - Unit tests (faster running automated tests): sono i test per un singolo metodo, classe o funzione e servono agli sviluppatori per controllare la buona riuscita del codice scritto. Per non rallentare troppo le operazioni non si collegano ai database. Quando una build passa gli unit test di riferimento viaggia all’interno della deployment pipeline e raggiunge la fase degli acceptance test.  
   - Acceptance tests (slower running automated tests): testano l’applicazione nel suo insieme assicurandone il funzionamento secondo i criteri e le funzionalità designate in fase di progetto e richieste dal cliente. Inoltre verificano che non siano stati introdotti errori di regressione (ovvero riguardanti funzionalità che precedentemente funzionavano). Possono essere richieste anche un paio d’ore per l’esecuzione.  
   - Integration tests: …  
     
   - Manual tests: sono gli ultimi a essere eseguiti (exploratory testing, UI testing). E’ importante capire quali casi necessitano di test manuali, applicare una mera automatizzazione di tutti i test manuali può provocare risultati indesiderati come i falsi positivi (es: i test automatici passano perché il codice è formalmente corretto, ma non ci rendiamo conto di un scarsa performance nel funzionamento di una feature).   
     
   Come si è notato, il tempo per l’esecuzione delle varie tipologie di test cresce procedendo lungo la deployment pipeline, per questo motivo è importante rilevare i problemi prima possibile, il che risulterebbe più gestibile rispetto al presentarsi di un errore durante i test di integrazione siccome, in tal caso, lo sviluppatore ne verrebbe a conoscenza solo dopo diverse ore e, oltre a un rallentamento del lavoro, ne complica la risoluzione. Per tale motivo è importante che gli Unit Tests siano completi e setaccino il più a fondo possibile il codice per scovare dei bug. Martin Fowler (*chi è*) esprime questo concetto attraverso la nozione di “ideal testing automation pyramid” in cui si vede che gli Unit Test sono alla base in quanto devono coprire la quasi totalità delle casistiche (*figura*).  
   Oltre a sottoporre alle varie fasi di test il codice, risulta necessario testare altri attributi del sistema, i cosiddetti “non-functional requirements” come la scalabilità, la capacità del sistema, la sua sicurezza ecc…