ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

PANORAMICA DELL'APPROCCIO ARCHITETTURALE ORIENTATO AI MICROSERVIZI E ANALISI DELL'APPLICABILITÀ NEL CONTESTO TRAUMATRACKER

Elaborato in
SISTEMI EMBEDDED E
INTERNET OF THINGS

Relatore
Prof. ALESSANDRO RICCI

Presentata da
MANUEL BONARRIGO

Co-relatori
Ing. ANGELO CROATTI
Dott.ssa SARA MONTAGNA

Seconda Sessione di Laurea
Anno Accademico 2017 – 2018

PAROLE CHIAVE

TraumaTracker

Microservizi

Virtualizzazione

Healthcare

Docker

Alla mia famiglia.

Indice

Introduzione

Negli ultimi anni, il concetto di *service orientation* è stato arricchito da una nuova accezione, basata sul rigore del processo produttivo e sulla resilienza del sistema in produzione al maggior numero possibile di fattori esterni. Quello che è stato chiamato approccio ai 'microservizi' rispecchia in realtà una filosofia nell'ambito dei sistemi distribuiti, presente fin dai primi anni dell'esplosione del Web, che prevede come il fine ultimo di un'applicazione venga suddiviso in una moltitudine di piccole entità, autonome e distribuite. L'aumento in importanza degli ultimi anni è data dall'esperienza acquisita dal primo decennio di questo secolo, in cui molte organizzazioni hanno tentato di seguire un approccio a servizi '*vendor-driven*', con vari middleware che prendevano il carico delle difficoltà gestionali e comunicative. Il prezzo pagato è stato quello di perdere il contatto con il proprio dominio applicativo, legandosi a tecnologie invasive, difficili da utilizzare e da sostituire.

L'approccio a microservizi è costruito sulla conoscenza e sull'esperienza ottenuta dall'analisi delle difficoltà passate, e indirizza direttamente le problematiche incontrate nelle applicazioni di inizi anni '00. L'obiettivo è quello di garantire uno sviluppo rapido delle funzionalità nel sistema in cui viene applicato, in maniera autonoma e indipendente, ma mantenendo un approccio allo sviluppo controllato e testato automaticamente, dalla compilazione alla produzione. L'alta coesione fra le funzionalità all'interno di un microservizio, unita all'accoppiamento lasco che intercorre fra questi, compongono l'aspetto fondamentale della libertà che gli sviluppatori possiedono nel creare le soluzioni richieste dalle necessità del dominio applicativo, al punto tale da poter utilizzare stack tecnologici completamente diversi all'interno di ogni microservizio, o fra una versione del microservizio e l'altro.

Tale libertà, unita all'approccio automatizzato e ai livelli di resilienza raggiunti dai migliori sistemi attuali basati su microservizi, hanno fatto sì che venisse considerata l'applicazione dei principi cardine di questa filosofia architetturale al sistema TraumaTracker, un progetto svolto in collaborazione con il TraumaCenter dell'ospedale Maurizio Bufalini di Cesena, che mira a fornire uno strumento di supporto alle operazioni di primo soccorso dei pazienti traumatizzati. Trattandosi di un dominio medico-ospedaliero, si è sentita forte la

necessità della garanzia di poter sia rispondere in maniera veloce ed efficiente a qualsiasi cambiamento, che di ottenere la massima stabilità del sistema, facendo fronte a qualsiasi eventualità, anche catastrofica, in linea con le necessità ospedaliere.

All'interno di questa tesi sono state analizzate le caratteristiche dell'approccio architetturale ai microservizi, considerando il costo in termini di lavoro da effettuare sulla *codebase* esistente del TraumaTracker allo scopo di migrarne l'infrastruttura a rispettare i principi di questa nuova concretizzazione della *service orientation*.

Background Il primo capitolo introduce brevemente le caratteristiche del TraumaTracker, e fornisce un *insight* maggiore sulle motivazioni che hanno portato alla necessità di considerare una struttura *back-end* più solida ed opinionata.

Microservizi Il secondo capitolo espone con particolarità quali sono i principi che caratterizzano l'approccio architetturale a microservizi entrando nel dettaglio di ognuno, segnalandone vantaggi e difficoltà nell'applicazione, oltre che alle *bad practice* note.

TraumaTracker orientato ai microservizi Il terzo capitolo esprime i cambiamenti necessari nell'infrastruttura del TraumaTracker perchè questo possa godere dei vantaggi dei microservizi, e vengono espressi gli obiettivi raggiunti ed i lavori futuri in relazione all'argomento di tesi.

Capitolo 1

Background: il progetto TraumaTracker

La ricerca, e il conseguente progresso della tecnologia, sono sempre state rivolte verso un unico obiettivo: il miglioramento della condizione umana. La medicina è forse il campo in cui questo principio prende la sua forma più concreta. Gli sviluppi tecnologici dell'ultimo secolo hanno contribuito a prevenire e curare malattie finora ritenute mortali[5], e scoprirne l'esistenza di alcune prima che lo divenissero[4]. Nella medicina moderna, la disponibilità, l'affidabilità e la precisione degli strumenti utilizzati da personale altamente qualificato sono i punti cardine dell'intera disciplina.

L'informatica è stata subordinata per molti anni a fornire il proprio contributo tramite il software che rende operative le apparecchiature necessarie ai medici per svolgere la propria attività, più che un supporto attivo alla professione. Tuttavia, la quarta rivoluzione industriale ha dimostrato che entità software di complessità adeguata possono sfruttare la quantità massiva di informazione generata dall'interconnessione di tali apparecchiature, al punto tale da essere in grado di effettuare decisioni in autonomia. Per quanto questo sia un'utopia nello scenario medico odierno, la strada che si è palesata è quella di un ambiente sanitario in cui al medico vengono consegnati in tempo reale i risultati di complesse elaborazioni dei dati raccolti dal paziente sul quale si sta effettuando un intervento, incrociati con eventuali conoscenze pregresse del sistema. Egli può quindi prendere decisioni basate su uno spettro di conoscenza espanso rispetto alla sola propria memoria.

1.1 Obiettivi

Il TraumaTracker si propone come uno strumento nel supporto organizzativo e amministrativo alla *trauma resuscitation*, ponendo particolare attenzione

alla *trauma documentation*.

Con *trauma resuscitation* si intende la stabilizzazione dei parametri vitali nei pazienti sottoposti a trauma ad una soglia che consenta loro di restare in vita. Questa pratica è caratterizzata sia da tempi di esecuzione stringenti che da numerose operazioni effettuate sul paziente (somministrazione di farmaci, primi interventi chirurgici, manovre di stabilizzazione), spesso simultanee.

La *trauma documentation* consiste invece nel report che il medico responsabile della cura del paziente traumatizzato (definito come TraumaLeader) deve compilare una volta terminata la stabilizzazione. L'accuratezza di tali report è di fondamentale importanza sotto molti punti di vista[6]:

- Diventa possibile valutare sia le operazioni del TraumaTeam in termini di performances, che la capacità dell'organizzazione logistica dell'ospedale nel permettere le operazioni necessarie in risposta al trauma;

- Le analisi a posteriori sui dati riportati permettono lo studio dell'evoluzione del trattamento e la correlazione fra le azioni intraprese e la percentuale di successo della resuscitazione;

- Le assicurazioni mediche e i responsabili della medicina legale necessitano della totalità degli interventi sul paziente traumatizzato per portare a termini i propri compiti.

Quando il TraumaLeader è direttamente coinvolto nelle operazioni di resuscitazione piuttosto che nella sola coordinazione del TraumaTeam, il *tracking* degli interventi, delle diagnosi e dei farmaci somministrati al paziente e la successiva compilazione dei report relativi sono completamente basati sulla memoria del Team e del Leader, con la responsabilità ultima direttamente di quest'ultimo.

Il compito di dover memorizzare qualsiasi operazione intrapresa diventa un fardello che appesantisce la capacità organizzativa del TraumaLeader. Inoltre, pur trattandosi di professionisti altamente specializzati, la memoria e l'attenzione di un essere umano, o di un gruppo di esseri umani, sono facilmente fallimentari. Per i motivi sopracitati, il TraumaTracker si pone come strumento cardine per la raccolta organizzata e programmatica dei dati relativi al paziente traumatizzato e sottoposto a resuscitazione.

1.2 Infrastruttura generale

Il compito di creare report relativi alle operazioni del TraumaTeam viene svolto dalla collaborazione fra tre sottosistemi software, visibili in figura 1.1.

Il componente denominato TLAAgent è l'interfaccia operativa messa a disposizione del TraumaTeam, nonchè la parte destinata all'utilizzo durante la resuscitazione. Per tale motivo è principalmente costituita da feature *wearable* e *hands-free*, come smart-glasses per la visualizzazione di dati in tempo reale, e capacità di speech recognition; un tablet permette la gestione capillare dei farmaci somministrati, delle procedure e delle diagnosi effettuate, del cambiamento nello stato vitale del paziente e dell'acquisizione di dati multimediali (registrazioni vocali, foto, video) necessari alle analisi successive.

Il tutto viene corredato dalle informazioni temporali e logistiche raccolte dalla componente *back-end* del sistema: il `TTLocationService`, tramite un'infrastruttura di beacon Bluetooth distribuiti per l'ospedale, tiene traccia della localizzazione geografica delle operazioni del TraumaTeam rilevando gli spostamenti dei device del TLAAgent.

Contestualmente, il `TTGatewayService` mette a disposizione i parametri vitali del paziente traumatizzato per tutto il tempo in cui egli rimane collegato agli strumenti di monitoraggio dell'ospedale. L'insieme di tutte queste informazioni, accuratamente composte, forma la *trauma documentation* richiesta al sistema tramite il `TTService`: tale servizio associa qualsiasi intervento medico al luogo in cui è stato eseguito, proiettando una serie di grafici temporali che evidenziano le conseguenze di tali operazioni sui parametri vitali del paziente traumatizzato.

La `TTDashboard` è una web-app che fornisce al TraumaLeader le funzionalità necessarie per gestire i report creati dal sistema, oltre che permettere l'elaborazione di ulteriori statistiche tramite i dati raccolti.

1.3 Benefici raggiunti

Il sistema è stato sviluppato e messo in esecuzione con la collaborazione del TraumaTeam dell'ospedale Bufalini di Cesena, che ospita uno dei più grandi centri traumatologici italiani. I benefici apportati dal TraumaTracker sono stati verificati tramite questionari posti agli stessi medici che ne hanno testato le funzionalità: l'unanimità ha concordato sul fatto che il sistema previene una ricostruzione errata del trattamento del trauma, oltre che presentare una forma standardizzata e pulita dei report.

Il cambiamento avvertito nella quantità di informazione raccolta, su una base generale di 50 parametri, è sempre stato positivo, con il 17% dei medici che trova che il sistema permetta di tenere traccia di 10 parametri in più rispetto alla stesura mnemonica dei report, il 33% concorda su 15, mentre il 50% concorda sul fatto che sia stato possibile registrare fino a 30 parametri in più.

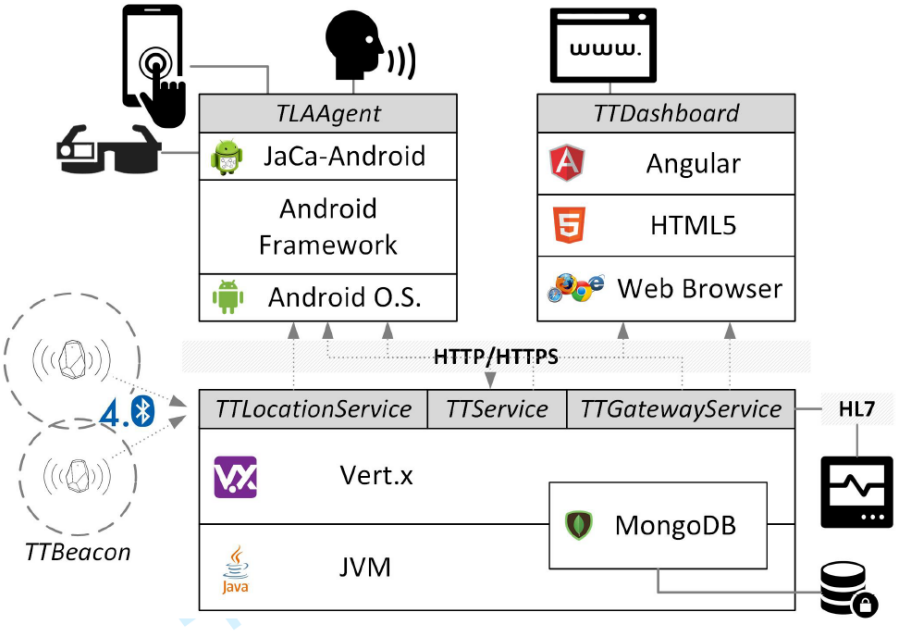

Figura 1.1: Infratruttura di massima del sistema TraumaTracker. Fonte originale [6]

Negli stessi questionari, metà dei medici ha affermato che la documentazione automatica permette di risparmiare 15 minuti nella stesura dei report. L'altra metà concorda su un risparmio temporale di 30 minuti netti.

Inoltre, è stato possibile aumentare la quantità media di informazione raccolta sullo stato di salute del paziente traumatizzato da 6,75 a 18 nelle fasi precedenti all'attivazione del TraumaTeam, cosicchè questo possa rispondere in maniera cosciente fin dal primo momento delle operazioni.

La differenza principale fra report mnemonico e automatico si evidenzia quindi nella precisione del secondo. I valori rilevati non vengono mai approssimati e sono sempre associati ad un momento temporale ben definito, permettendo analisi successive molto più accurate.

Velocità, precisione, facilità d'uso e affidabilità sono tutti fattori estremamente importanti nella disciplina medica. È stato dimostrato come la mortalità dei pazienti sia direttamente correlabile ad una bassa qualità della *trauma documentation*, poichè questa si pone come esplicitazione della coordinazione fra i professionisti sanitari e la loro capacità decisionale [6]. Inoltre, un ulteriore studio relativo alla documentazione medica prova come una migliore qualità di quest'ultima renda possibile ridurre la degenza ospedaliera dei pazienti traumatizzati [7]. Il sistema TraumaTracker ha quindi le potenzialità per

diventare uno strumento di fondamentale importanza, a servizio di qualsiasi team medico.

1.4 Problematiche emerse

Da un punto di vista tecnologico, gli ostacoli consistono nel portare la resilienza dell'infrastruttura del TraumaTracker agli standard che ci si aspetta da un ospedale. Nulla dovrebbe poter permettere un'interruzione del servizio, nella stessa maniera in cui un black-out non intacca l'operatività della struttura sanitaria. In maniera sicuramente troppo approssimativa rispetto alla realtà, il lavoro necessario si può riassumere nell'espandere il comportamento del sistema fino all'aver coperto tutte le eventualità possibili, dall'incrinazione di uno degli schermi dei tablet del TLAAgent alla catastrofe naturale. Seppure estremamente vasto, si tratta comunque di un insieme finito e prevedibile di scenari. Ciò che sia la disciplina ospedaliera sia quella informatica hanno in comune, è la prevenzione del fallimento tramite il rigore nei processi: la commistione fra le due prevarrà sicuramente su tutte le possibili interruzioni del servizio che si verranno a riscontrare.

Un evento al di fuori di questo insieme, estremamente noto per la propria capacità dirompente di interrompere l'operatività del software in produzione, è il cambiamento del dominio applicativo di riferimento, e non è possibile evitare di considerare come il dominio medico sia forse uno dei più complessi e dinamici a cui l'informatica si possa affacciare: per questo motivo è fortemente auspicabile orientare l'architettura dell'infrastruttura *back-end* verso una forma che abbracci l'evoluzione e garantisca una facile integrazione di nuove funzionalità di back-end, oltre che la possibilità che tutte quelle esistenti possano essere composte in nuove applicazioni, dalla logica ancora sconosciuta, soprattutto considerando la calda accoglienza che il sistema TraumaTracker ha ricevuto dal personale medico che ne ha testato le funzionalità.

Nonostante l'infrastruttura generale sia già stata progettata tenendo conto di molte di queste problematiche, la quantità di scenari fallimentari è estremamente vasta, così come la quantità di scenari evolutivi possibili. Per affrontare questo problema, e per prevenirne molti altri che diventeranno chiari nel seguito della dissertazione, si è deciso di effettuare uno *shift* nei confronti dell'approccio architetturale che maggiormente si è fatta notare per la sua resilienza nei confronti dell'evoluzione del dominio a cui è applicata e del fallimento di pressocchè qualsiasi entità con la quale interagisca, mantenendo comunque una flessibilità tale da permettere la massima eterogeneità nelle tecnologie e soluzioni adottate, ed un riuso ed una componibilità eccezionale: i microservizi.

Date le caratteristiche di una tale infrastruttura (evidenziate nel prossimo capitolo), è possibile affrontare una problematica che trascende il livello tecnologico. Esistono infatti delle opposizioni all'idea che dei sistemi software affrontino autonomamente delle problematiche all'interno delle aziende ospedaliere [6]. L'utilizzo di un tale approccio architetturale, basato su principi molto forti di modularità e testing intensivo, permette un controllo estremamente disciplinato del corretto funzionamento del software prodotto nelle sue parti critiche, permettendo la consegna di una garanzia particolarmente valida sull'operatività delle stesse.

Capitolo 2

Microservizi

Il termine microservizi descrive uno stile architetturale in cui molte piccole componenti software distribuite collaborano nel creare sistemi di informazione. Martin Fowler e James Lewis descrivono i microservizi come "[...] *un approccio nello sviluppare una singola applicazione nella forma di una suite di servizi di dimensioni ridotte, ognuno dei quali possiede il proprio flusso di esecuzione e comunica con gli altri tramite meccanismi a basso overhead. Tali servizi vengono modellati in relazione ad una parte atomica del dominio, e possono essere messi in funzione tramite strumenti automatizzati di controllo software. [...] possono essere sviluppati utilizzando linguaggi di programmazione eterogenei ed utilizzare tecnologie di persistenza diverse l'uno dall'altro*" [8].

In un periodo in cui la parola 'microservizi' non esisteva nemmeno, Werner Vogels definiva come la service orientation rappresentasse per Amazon "[...] *l'incapsulamento dei dati insieme alla business logic che opera su tali dati, consentendo come unico punto di accesso un'interfaccia nota del servizio. Non sono permessi accessi diretti al database dall'esterno del servizio che lo gestisce, e non esistono dati condivisi fra i servizi*" [9].

Sam Newman li definisce in maniera più ermetica come "[...] *piccoli servizi autonomi che cooperano*", nell'incipit di un intero libro dedicato all'esposizione di tale forma architetturale [10].

La mancanza di una definizione formale scaturisce dal fatto che i microservizi non sono tanto un pattern architetturale, quanto più una libera interpretazione del concetto di service orientation indirizzata alla scalabilità, con esempi di estremo successo quali Amazon e Netflix dai quali prendere spunto.

Fattorizzando tutte le descrizioni precedenti, l'architettura a microservizi può essere riassunta come la decomposizione di un dominio applicativo complesso nelle sue sottoparti atomiche, delle quali si isola la responsabilità all'interno di componenti distribuite in cui ognuna ha il completo controllo

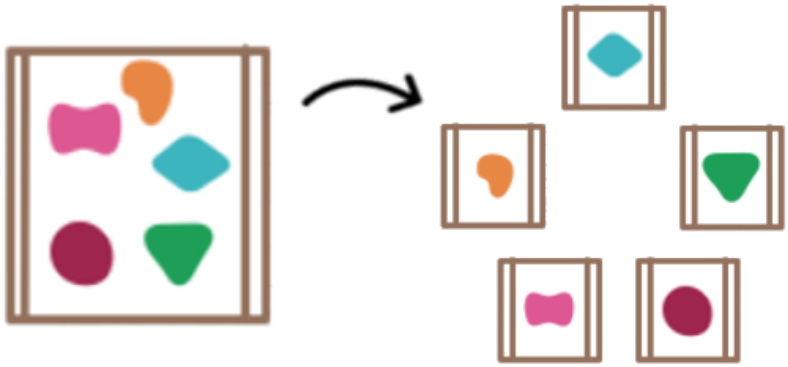

Figura 2.1: Differenza strutturale fra l'approccio monolitico ed un approccio a microservizi

delle proprie risorse, che non vengono condivise con nessun altro componente. La comunicazione fra queste entità autonome avviene tramite meccanismi agnostici all'informazione che sta venendo trasportata, sulla base di interfacce pubblicate.

Una rappresentazione visiva particolarmente significativa per catturare il concetto dietro i microservizi è in figura 2.1: le forme colorate rappresentano le diverse funzionalità del sistema, e sono circondate dal confine che rappresenta l'unità applicativa. Un esempio immediato di come questa separazione risulti vantaggiosa si ha nel momento in cui ci si trova davanti alla necessità di scalare le capacità del sistema (figura 2.2): con l'approccio a microservizi ci si può permettere di replicare solo ciò che è necessario, nei punti in cui tale bisogno si manifesta.

Ciò che va attivamente ricercato per determinare l'appartenenza o meno all'attuale corrente architetturale è l'adesione ad una serie di principi cardine che garantiscono la scalabilità e la resilienza richiesta da questa particolare concretizzazione della service orientation.

2.1 Principi

Concependo tutti i microservizi di un sistema, diventa palese come organizzare la coesione di una moltitudine di applicazione autonome distribuite aumenti notevolmente la complessità gestionale rispetto ad un unico processo dal flusso di controllo ben definito. Diventa quindi necessario che ogni micro-

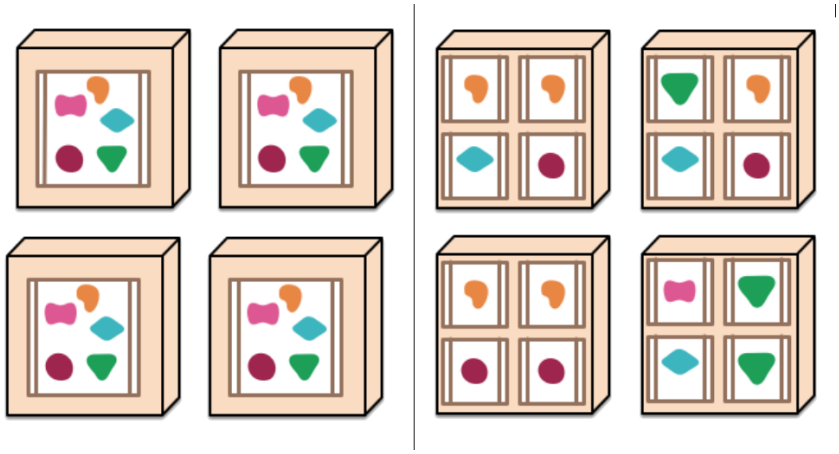

Figura 2.2: Differenza concettuale nello scalare un'applicazione monolitica e un'applicazione organizzata intorno ai microservizi

servizio venga sviluppato seguendo dei principi che permettano all'insieme di singoli di costituire un sistema dalle proprietà omogenee.

Nonostante tali principi possano differire in base alle necessità del sistema che sta venendo sviluppato, in [11] ne vengono identificati sette che si ritengono essere basilari per il successo nell'approccio architetturale basato sui microservizi. Ognuno di questi risponde ad una problematica ben definita nel campo del controllo della coordinazione di processi distribuiti, come alle difficoltà note durante lo sviluppo di un sistema dall'architettura così particolare.

Una prima, minimale, definizione di ogni principio consiste in:

- **Suddivisione atomica del dominio** – Ogni microservizio è addetto a svolgere una singola, indivisibile, sottoparte del dominio applicativo. Le funzionalità espresse da questo sottodominio costituiranno l'interfaccia pubblicata del microservizio.

- **Cultura dell'automazione** – Testing, deployment ed esecuzione devono essere ripensati in un'ottica nella quale quante più problematiche vengano assorbite da una logica superiore, permettendo di indirizzare gli sforzi dello sviluppo altrove.

- **Segregazione implementativa** – L'unico modo per ottenere le informazioni che un microservizio elabora deve consistere nell'utilizzare l'interfaccia che questo pubblica. Le banche dati sono da considerarsi inaccessibili a priori, e di proprietà del microservizio che le definisce. Tramite l'adesione a questo principio, ogni microservizio può utilizzare liberamente lo stack tecnologico che più si confà alle proprie necessità e sostituirlo in qualsiasi momento, in quanto nulla potrà creare dipendenze a riguardo.

- **Decentralizzazione pervasiva** – Promuovere una logica completamente contenuta all'interno del microservizio, senza fare assunzioni forti nè sui metodi di comunicazione nè sulla struttura degli altri microservizi.

- **Deployment indipendente** – Rendere ogni microservizio deployabile in maniera assolutamente indipendente l'uno dall'altro. Soprattutto, la modifica del funzionamento interno di un microservizio non dovrà intaccare in alcun modo quell già esistenti.

- **Resilienza agli errori** – Ogni microservizio, e di conseguenza l'intero sistema, deve essere progettato considerando sia il mezzo di comunicazione, che gli altri servizi, che la macchina host potrebbero fallire senza alcun preavviso, in qualsiasi momento. Ci si aspetta che il microservizio preveda queste occasioni e sappia come comportarsi nelle diverse casistiche.

- **Monitoring pervasivo** – È necessario intraprendendere una forma di logging che permetta di individuare e risolvere ogni forma di disservizio in una qualsiasi delle parti del sistema. Per tale motivo il monitoring non dovrà essere effettuato internamente ai microservizi, ma in un eventuale microservizio dedicato.

Molti dei principi elencati ricalcano norme di buona programmazione (in particolare del paradigma ad oggetti) estremamente note, senza le quali l'intera disciplina informatica farebbe parecchia fatica ad ottenere i risultati e gli sviluppi a cui sta abituando il mondo. Esiste tuttavia una differenza notevole nella misura in cui queste norme vengono viste nel mondo dei microservizi: non seguire in maniera precisa una linea pulita e programmatica di design e sviluppo rende gli interventi manutentivi estremamente difficili in una fase iniziale del sistema, assolutamente impossibili quando il numero di microservizi comincia a crescere. Di recente, l'azienda di analisi informatica Segment ha dismesso completamente i microservizi, ritornando di fatto ad un architettura con server monolitici. L'unica regola che mancava alla loro visione del sistema era la decentralizzazione pervasiva applicata ad ogni microservizio, e il risultato è stato la completa impossibilità nel progredire con lo sviluppo [12].

Ognuno di questi principi verrà analizzato in maniera approfondita in un paragrafo dedicato. Contestualmente, verranno presentati esempi architetturali e tecnologie utili per risolvere i problemi noti del principio in analisi.

Figura 2.3: I sette principi fondamentali dei microservizi identificati da Sam Newman [11]

2.2 Suddivisione atomica del dominio

Concepire unità di programmazione che svolgano un solo compito è una norma di design nota in qualsiasi ambito della disciplina informatica. Tale principio è conosciuto con molti nomi in diversi ambiti, con uno dei più diffusi essere il *Single Responsibility Principle* [13] nel paradigma di programmazione ad oggetti. Nell'architettura a microservizi questo viene allargato a coprire un'intera sottoparte del dominio applicativo, di cui si assume la totale responsabilità. In letteratura, il sottodominio in questione prende il nome di Bounded Context [2].

Nonostante le differenze sostanziali nel punto di applicazione dei due principi, le qualità desiderate dall'utilizzo rimangono le stesse: ottenere *loose coupling* fra microservizi differenti e *high cohesion* all'interno dello stesso. Ciò che si cerca di ottenere è un ambiente dalla logica isolata, che non venga influenzato dai cambiamenti nel resto dell'ecosistema di microservizi. Allo stesso tempo, se un qualsiasi requisito funzionale dovesse cambiare rendendo la logica obsoleta, tutte le modifiche andrebbero effettuate in un unico luogo.

I benefici raggiungibili tramite una separazione netta nei confini fra microservizi sono numerosi.

Da un punto di vista tecnico, si vengono a creare delle interfacce di ser-

vizio molto forti, che fungono da barriera nei confronti del resto del sistema. Data la natura distribuita, risulta impossibile creare delle dipendenze da moduli software dei quali non si ha la paternità, così come è impossibile usarli in maniera impropria: tutta la comunicazione inter-processo avviene tramite chiamate remote asincrone, senza che nulla venga reso pubblico riguardo alle implementazioni scelte per assolvere la chiamata. Considerando di dover applicare un qualche tipo di manutenzione ad uno dei microservizi, l'alta coesione delle operazioni che contiene facilita la comprensione di quale sia la logica interna, il che aumenta la possibilità di comprendere cosa cambiare, ed in che punto, perchè la manutenzione abbia il successo desiderato [8].

Da un punto di vista umano, uno dei principali vantaggi risiede nello *shift* di responsabilità del team che lavora al microservizio. Concentrandosi principalmente sulle capacità offerte dal Bounded Context a cui lavorano, gli sviluppatori diventano esperti del sotto-dominio di cui è richiesta l'implementazione nel sistema, prima che delle specializzazioni tecnologiche necessarie a portare a termine tale compito. Ricorsivamente, una migliore comprensione del dominio da parte di chi lo gestisce consente una migliore qualità dei servizi offerti dalle interfacce [3].

2.2.1 Bounded Context

La definizione formale di Bounded Context consiste in "una parte definita del software in cui un insieme di termini, definizioni e regole vengono seguite in maniera consistente"[2], ovvero il confine della parte di dominio applicativo oltre il quale le motivazioni che portano ad una particolare struttura della logica interna di comportamento, se conosciute, non vengono condivise nè rispettate. Aumentando la complessità del dominio, tali confini diventano sempre più difficili da tracciare, e separare completamente le parti logiche che sono in grado di svolgere una funzionalità autonoma diventa progressivamente più complesso, se non impossibile.

Parafrasandone l'adagio, la legge di Conway esprime come, nel modellare un sistema, si sia vincolati nell'imitare le modalità comunicativo-organizzazionali di tale sistema. Effettuare un tale tipo di analisi, e verificare che i confini tracciati fra i Bounded Context separino zone del modello coerenti ed autonome che interagiscono in maniera nota e modellata, è una prima indicazione della bontà del proprio operato. Unificando la definizione di Evans, i Bounded Context vanno quindi ricercati come confini tracciati dai processi comunicativi in atto fra i modelli delle parti del dominio, e devono contenere responsabilità fortementente coese, il più possibile indipendenti dal resto del sistema. Una pratica più empirica per determinare i confini fra le parti di un sistema è data dall'*ownership* dei dati che vengono utilizzati nell'esecuzione delle diverse

operazioni offerte: come si vedrà al paragrafo 2.4, ogni microservizio dovrà conservare le proprie informazioni in maniera trasparente al sistema. Creare un Bounded Context intorno alle parti che godono di una grande coesione e un proprietario ben definito è quindi una buona, ma non totale, garanzia su come, a patto di evoluzioni del dominio applicativo, tali confini resteranno saldi.

Questo é anche uno dei problemi principali che si presenta nel momento in cui tale principio si applica ai microservizi. I Bounded Context rappresentano il confine, l'interfaccia pubblicata con il quale si proporranno funzionalità agli altri microservizi: dal momento dell'entrata in produzione, rappresenteranno tutto ciò che viene offerto al sistema. Sbagliare la progettazione di tali confini, per quanto sia un errore al quale si può rimediare (paragrafo 2.6.2), toglie risorse allo stato di evoluzione e miglioramento costante a cui l'approccio dei microservizi vorrebbe arrivare, ed è una problematica da tenere in considerazione durante le fasi di analisi.

2.2.2 Nanoservizi

I nanoservizi consistono nell'applicazione del concetto di singola responsabilità ad una riduzione sostanziale del Bounded Context, in pratica a una singola operazione dello stesso. Sulla base di questa pletora di nanoservizi, vengono creati gli effettivi servizi offerti al codice cliente.

È notoriamente considerato un antipattern [15, 16], ma esistono aziende che affermano di utilizzare questo stesso approccio nella loro fase di produzione con notevoli vantaggi [17].

Il problema è che la responsabilità del servizio è l'unico fattore ad essere ridotto: tutto l'overhead dell'architettura resta presente, e la critica che viene posta resta valida nel momento in cui i benefici ottenuti tramite una tale, minuziosa, decomposizione non coprono il costo della messa in piedi del sistema di gestione di microservizi.

2.3 Cultura dell'automazione

La suddivisione di un dominio applicativo in sottoparti atomiche presenta un problema: la quantità di microservizi che si vengono a creare scala linearmente con la dimensione e complessità del dominio in analisi. L'idea di gestire le fasi di sviluppo, testing, deploy e debug tramite lavoro manuale diventa in maniera estremamente rapida un collo di bottiglia molto stretto per la natura in costante evoluzione del sistema. Tuttavia, la pratica denominata *Continuos Delivery* mira esattamente a risolvere questo problema, permettendo non solo di avere un'evoluzione rapida del sistema, ma di avere un'evoluzione rapida e sicura.

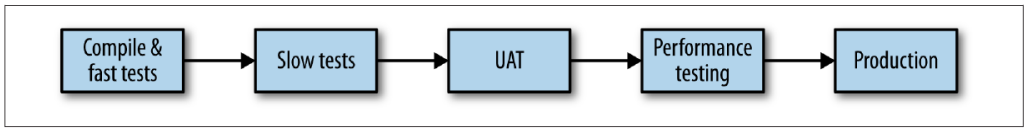

Figura 2.4: Un esempio di pipeline di Continuous Delivery

L'essenza della tecnica consiste nell'avere una serie di step automatizzati che certifichino la conformità del prodotto finale con la linea in produzione. Tuttavia è fondamentale notare come, alla base della Continuous Delivery (CD), sia presente la Continuous Integration (CI): la CI è un approccio alla fase di *check-in* del codice che deve essere introiettato dai team di sviluppo. Consiste nell'integrazione costante (da cui il nome) del codice prodotto da ogni sviluppatore con il branch di sviluppo principale, eliminando alla base tutti i problemi dovuti all'integrazione di grosse quantità di codice con diversi giorni (o settimane) di sviluppo alle spalle.

La CD rappresenta l'utilizzo di strumenti automatizzati finalizzati a verificare che tali cambiamenti non impatteranno in maniera negativa la *mainline* del software, per far sì che il tempo e le energie degli sviluppatori possano essere dedicati al miglioramento delle qualità del prodotto.

In figura 2.4 è presente un esempio della pipeline di sviluppo seguita dalla compilazione all'integrazione. È importante notare come anche il processo di compilazione venga standardizzato ed automatizzato, in quanto esistono linguaggi (per esempio C++ e Scala) nei quali è possibile comunicare dei *flag-value* al compilatore in maniera tale da alterare il risultato della compilazione, e questo può introdurre dei comportamenti imprevisti in produzione.

Avendo ottenuto una corretta compilazione, la fase successiva consiste nell'esecuzione di tutti i test ritenuti veloci, secondo aspettative soggettive, come l'Unit Testing. Subito dopo vengono effettuati i test più lenti, che generalmente includono comunicazione reale tramite la rete. Questa separazione avviene per la diversa gravità del fallimento nelle due fasi, oltre che per la velocità di individuazione e correzione di un errore. Supponendo che le due fasi di testing vengano effettuate contemporaneamente, i test lenti potrebbero ritardare di parecchio la scoperta e la correzione di un errore in un Unit Test. Soprattutto, non ha senso fare partire i test lenti se quelli veloci non vanno a buon fine.

L'User Acceptance Testing (UAT) è la certificazione dell'effettiva consegna delle funzionalità richieste da parte degli utenti finali. Questo assume una connotazione particolare nel caso dei microservizi, affrontata con maggior dettaglio in seguito (paragrafo 2.6.1). I test sulle performance attestano che, dopo le eventuali modifiche, il sistema rispetti ancora i vincoli fisici con i quali è stato progettato: tempo di latenza delle risposte, cicli di cpu utilizzati, etc.

Risultano particolarmente importanti lavorando nello scenario di coesistenza di diversi microservizi sullo stesso host ipotizzato in 2.8, in quanto si vuole sapere se si sta per sovraccaricare una macchina con delle richieste che non riuscirà a soddisfare.

Il concetto di integrazione continua deriva in maniera imprescindibile dalla risoluzione pressocchè immediata di qualsiasi errore venga palesato tramite questi test.

2.4 Segregazione implementativa

La segregazione implementativa è un concetto estremamente noto nella disciplina informatica: si basa sull'idea che, per il codice cliente, le tecnologie utilizzate nel realizzare il modulo software di cui sono richieste le operazioni siano del tutto trasparenti. Un paragone semplicistico con il paradigma ad oggetti si ha nel *Dependency inversion principle*, in cui si attesta che bisogna dipendere dalle astrazioni date dalle interfacce, e non dai dettagli implementativi con cui tali interfacce vengono realizzate [14]. Nello scenario distribuito dei microservizi, tale impossibilità nella dipendenza dai dettagli deve essere garantita dall'interno: le operazioni esposte non devono rispecchiare la forma delle tecnologie interne, ma la forma delle funzionalità semantiche che il Bounded Context attorno al quale si costruisce il microservizio offre.

I vantaggi si palesano nel momento in cui diventa possibile sviluppare ogni microservizio usando le tecnologie più appropriate per il caso e – molto più importante – risulta possibile cambiare tale stack tecnologico anche mentre il microservizio è già in operatività, poichè il codice cliente potrà effettuare le proprie richieste tramite un'interfaccia trasparente ai cambiamenti avvenuti.

Uno dei dettagli implementativi più importanti da nascondere utilizzando l'architettura a microservizi è la scelta della tecnologia di persistenza. Dopo quello che è stato definito il 'Disgelo dei database' [18] è possibile scegliere soluzioni diverse che rispondano alle diverse necessità nell'aggregare i propri dati, con l'obiettivo di aumentare le performance di querying. Qualunque sia la scelta, è molto importante che nulla dall'esterno del microservizio possa effettuare comunicazioni dirette al database in questione. Questo infatti, essendo considerato un dettaglio implementativo, potrebbe venire sostituito da una nuova tecnologia di persistenza che si basi sullo stesso schema, e se l'accesso diretto fosse possibile, qualsiasi consumatore si troverebbe danneggiato. L'unica maniera per ottenere informazioni da questa banca dati è che esista un'operazione apposita nell'interfaccia pubblicata dal microservizio, così come per il resto delle funzionalità.

Figura 2.5: Esempi di eterogeneità delle tecnologie utilizzabili in un sistema a microservizi

2.4.1 Database integration

Questa pratica, che consiste nel trasferire la coordinazione dei dati creati da una moltitudine di sorgenti in un unico database centralizzato, viene identificata come pura e semplice causa di fallimento nell'ambito dei microservizi, e non dovrebbe mai essere presa in considerazione. Il principale problema che si viene a creare è la dipendenza dei clienti della banca dati dalla tecnologia e dallo schema che al momento vengono utilizzati. Questo rende impossibile la modifica di entrambi senza che i clienti ne siano impattati negativamente.

Una prima patch a questo problema consiste nel trasformare l'architettura in figura 2.6 in quella mostrata in figura 2.7, dove viene rimossa la possibilità di accedere direttamente alla banca dati costruendo un microservizio che incapsuli il database, ed esponga un'interfaccia che permetta una forma di accesso basato sulla semantica di quello che si desidera, piuttosto che una query nuda e cruda. In una seconda fase della risoluzione del problema della *database integration*, si possono usare una serie di tecniche basate all'incirca sugli stessi principi presentati nel paragrafo 2.2.1 per suddividere le responsabilità manageriali del database, e consegnarle ai microservizi di competenza. Essendo la banca dati mascherata da un microservizio che ne regola l'accesso questa operazione può essere eseguita in relativa tranquillità, poichè sarà sua responsabilità reindirizzare le richieste che riceve ai nuovi curatori delle parti del database. Nell'eventualità, futura rispetto a questo scenario, in cui i vari clienti effettuino le richieste direttamente ai nuovi responsabili e che il traffico verso il microservizio-maschera sia nullo, questo può venire dismesso.

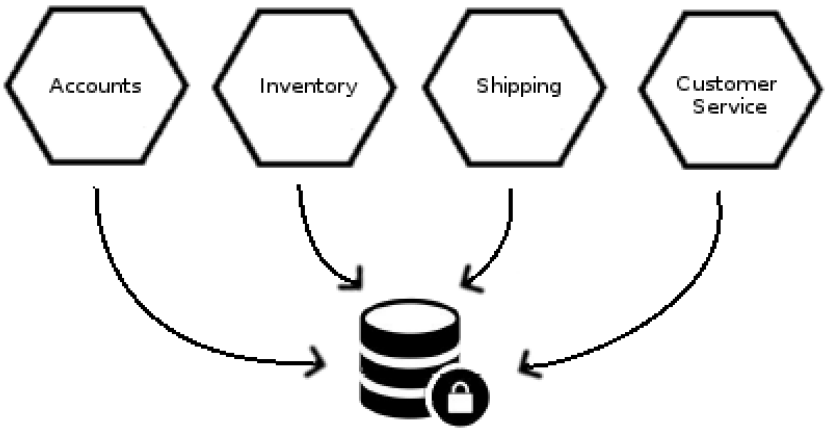

Figura 2.6: Database integration

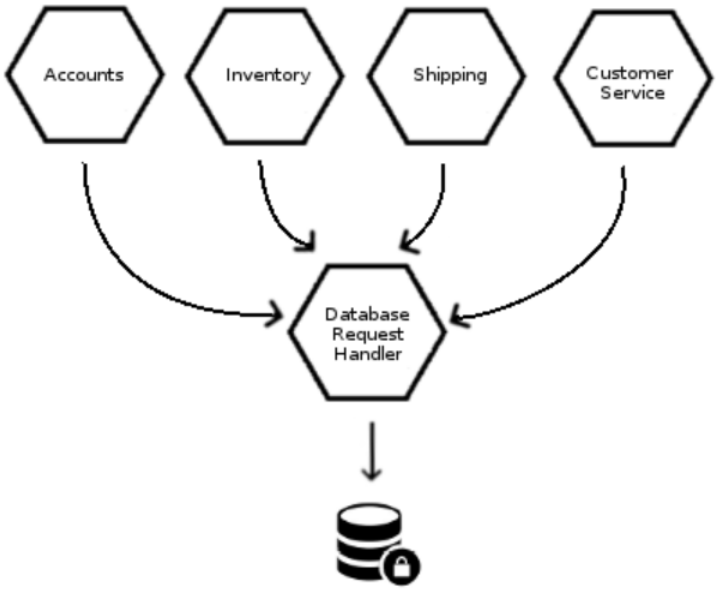

Figura 2.7: Un database il cui accesso diretto viene intercesso per mezzo di un microservizio

2.5 Decentralizzazione pervasiva

La decentralizzazione pervasiva è uno dei principi tramite i quali si concretizzano le condizioni necessarie affinchè sia possibile applicare il principio del deployment indipendente. Consiste nell'isolamento di ogni forma di tecnologia necessaria per concretizzare il Bounded Context di riferimento, in maniera tale che non esista alcuna dipendenza tecnologica nei confronti di altre parti del sistema. Data la complessità del contesto di applicazione, effettuare un tale taglio verticale è un problema che va affrontato da numerosi punti di vista.

Sharing is not caring Framework che orchestrino il funzionamento del sistema nel suo complesso, o l'utilizzo di librerie comuni per svolgere funzionalità simili, così come il riuso di strutture dati appartenti ad altri microservizi, sono visti come *bad practices* in quanto forzano in qualche maniera diverse parti del sistema ad essere accoppiate. L'accoppiamento di qualsiasi microservizio significa un ostacolo insormontabile per la sopravvivenza dello sviluppo indipendente. Bug nella libreria, diverso versioning della stessa, differente filosofia nell'utilizzarla, sono tutte problematiche che derivano dall'accoppiamento ad una stessa libreria. Rinunciare alla comodità nominale di un framework è un prezzo sufficientemente basso da pagare per rinunciare anche alla sola evenienza di complicazioni così subdole e impredicibili fra microservizi.

GUI Realizzare tutte le funzionalità necessarie all'autosostenimento di un intero microservizio va dalla realizzazione dell'interfaccia grafica al mantenimento degli host sui quali questi poi andranno in esecuzione. Per adempiere al primo requisito esistono due differenti percorsi a seconda della complessità, data dal numero di microservizi a cui ci si collega per ottenere le informazioni da mostrare: nel caso in cui siano veramente tanti, cooncentrare tutta la logica di presentazione in un unico posto risulterebbe troppo oneroso, oltre alla possibilità in cui particolari informazioni potrebbero avere la necessità di essere visualizzate con un struttura particolare, responsabilità del microservizio che le ha generate. In questi casi, sfruttando un sistema basato sui componenti, è possibile ricevere direttamente uno di questi già inizializzato a contenere le risposte, ed avere solo la responsabilità di inserirli nello scheletro dell'interfaccia grafica. In questa maniera è possibile essere più dinamici nella realizzazione dell'UI. È necessario prestare parecchia attenzione, in quanto questa è un'eccezione alla regola precedente.

DevOps La cultura DevOps risponde alla necessità dei team di sviluppo di dover gestire anche le macchine su cui verrà effettuato il *deploying*: esistono troppe variabili a livello operazionale perchè non abbiamo peso nel processo di

modellazione e realizzazione del software. La cultura DevOps prevede che la messa in piedi di un sistema (o di un modulo) non venga considerata divisa fra sviluppo e deploy, ma che vada seguita da entrambi i punti di vista sin dalle prime fasi della modellazione, sia dagli sviluppatori che dagli 'operazionali'. La convivenza di questi due aspetti permette agli sviluppatori di ottenere un *insight* aumentato rispetto alle necessità fisiche del prodotto, oltre a quali punti necessitano di una maggiore copertura da un punto di vista del logging. Per gli operazionali diventa più chiaro quali siano i punti critici nella *mission* dell'applicazione, capendone gli obiettivi, i problemi, e le necessità a partira dalle fasi iniziali della definizione del software.

Formato È necessario che il formato di comunicazione resti completamente agnostico rispetto al sistema di comunicazione che si sceglie. Vincolare le due parti significherebbe diffondere per l'intero sistema un accoppiamento estremamente difficile da eradicare in futuro, contraddicendo completamente il primo elemento di questo listato. La scelta ricade facilmente nell'insieme dei formati testuali *mainstream* come JSon, XML, HAL o addirittura HTML. L'XML è decisamente il più espressivo e compatto, definendo idiomaticamente la capacità di riferirsi ad altri sistemi tramite un sistema di hyperlinking, rendendo facile localizzare il punto del sistema che fornirà la risorsa necessaria. Il JSon è estremamente più leggero nei confronti del sistema di comuicazione e di facile fruibilità da parte dell'occhio umano, pur perdendo in capacità espressive. HAL copre esattamente il caso in cui si desideri utilizzare JSon insieme alla funzionalità di hyperlinking già presente in XML. La forza dell'HTML sta nel fornire sia funzionalità messaggistiche che di presentazione, e il formato è molto più conosciuto di tutti quelli già presentati. Affidare questa doppia responsabilità al formato di comunicazione, tuttavia, espone più facilmente a problematiche dovute al differente tipo di interazione che esiste con l'HTML da parte di un'applicazione ed un essere umano.

Dumb pipes, smart endpoints La concezione di service orientation del decennio scorso è stata negativamente influenzata dalla comparsa sul mercato di middleware di comunicazione che promettevano di privare gli sviluppatori del fardello di gestire tutti i vari aspetti della distribuzione. Forti della garanzia, questo tipo di tecnologia (con esempio più nefasto il concetto di Enterprise Serial Bus) è stato installato al centro di molti sistemi orientati ai servizi. L'errore concettuale è stato rendere coeso il proprio sistema tramite un middleware che col tempo, promettendo fasi di sviluppo sempre più semplificate, offriva sempre più funzionalità in cambio della responsabilità di parti del dominio, diventando sempre più legato al sistema. Se la libreria di comunicazione avesse subito un aggiornamento, l'intero sistema ne avrebbe dovuto gestire le conse-

guenze. Il primo punto di questa lista veniva completamente negato, con le insostenibili conseguenze nei confronti dello sviluppo. Cercando di sovvertire questa tendenza, viene proposta l'idea che nulla della logica interna di un microservizio debba essere conosciuta dal medium di comunicazione, e che l'unica funzionalità di questo sia trasportare informazioni dal punto A al punto B – *Dumb pipes, smart endpoints* [11]. Nonostante sia possibile implementare le proprie soluzioni, lo stile REST basato su HTTP fornisce l'implementazione delle capacità richieste, rispettando l'agnosticismo che permette il disaccoppiamento fra l'infrastruttura di comunicazione e le componenti operative del sistema. Tutto ciò che viene scambiato è semplicemente una 'richiesta' o una 'risorsa', senza che nulla dei dati contenuti traspaia.

2.6 Deployment indipendente

Da molti autori e professionisti questo è definito come il principio più importante per definire l'architettura a microservizi. È il concetto che esprime la vera natura autonoma del software creato, e prevede che, nel voler *deployare* un nuovo microservizio o la nuova versione di uno già esistente, tale operazione non alteri in alcun modo lo stato del sistema. I concetti di Bounded Context, segregazione implementativa e versioning effettuano già un servizio di protezione abbastanza efficace nei confronti della manutenzione evolutiva a fronte di cambiamenti del dominio: tuttavia, la preservazione del sistema avviene soltanto da un punto di vista software.

Nell'ottica dei microservizi esistono svariate forme di deploy su una macchina host, con le due principali raffigurate in figura 2.8. La forma di deploy rappresentata nella parte destra potrebbe risultare la più comoda in una situazione logistica in cui non si hanno macchine a sufficienza, oppure ottenere una macchina virtuale include il sollevamento di talmente tante richieste al punto di impedire la progressione dello sviluppo. Tuttavia, con la coesistenza sullo stesso host esiste la possibilità che un nuovo microservizio con una elevata necessità di risorse hardware (o una poco oculata gestione delle stesse) porti allo stallo del sistema. In una tale situazione, si perderebbero quattro servizi, insieme al rispetto del principio del deploy indipendente. Nella situazione a sinistra questo scenario è semplicemente impossibile, in quanto uno stallo dell'host non intaccherebbe il resto del sistema; tuttavia questa configurazione risulta molto più costosa in termini di organizzazione e coordinazione, oltre che a richiedere ovviamente più macchine e risorse per essere portato a compimento. Il livello di resilienza che si riesce ad ottenere giustifica questo costo, ma in condizioni

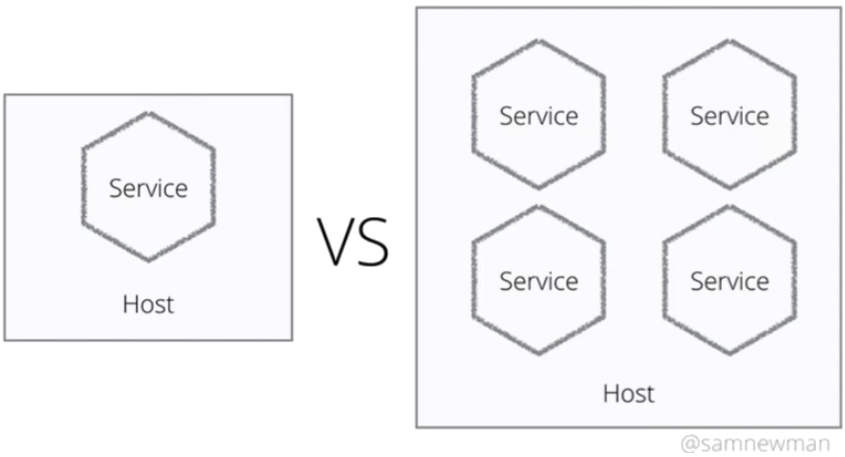

Figura 2.8: Un microservizio in esecuzione su un unico host sulla sinistra, multipli microservizi sullo stesso nella destra. Fonte originale [25]

di basse necessità di scaling, o in un momento di prototipazione del sistema, la configurazione di destra riesce ad essere tollerata.

Il termine 'Deployment indipendente' trasporta con sè un'altra accezione semantica, orientata alla gestione dei team di sviluppo e delle loro tempistiche di *delivery*. Uno dei vantaggi di possedere un'infrastruttura in cui una moltitudine di parti collabora in maniera agnostica l'una rispetto all'altra per fornire un servizio più complesso si manifesta nella possibilità di assegnare priorità diverse a microservizi diversi. Sulla base delle analisi effettuate sul dominio applicativo, infatti, sarà possibile individuare dei microservizi la cui operatività è critica per la *mission* del sistema, o dai quali ci si aspetta di ricevere un profitto maggiore rispetto agli altri, sia in termini di utilità intrinseca che, banalmente, denaro. Sulla base di questa consapevolezza è possibile calibrare la dimensione ed esperienza dei relativi team di sviluppo. I diversi microservizi possono quindi venire realizzati e rilasciati seguendo tempistiche differenti.

2.6.1 Testing

Pre-Production

Uno dei metodi per garantire che il nuovo microservizio sia capace di inserirsi in maniera sicura ed indipendente all'interno del sistema è affidarsi ad una serie di test che ne provi l'affidabilità in tutta una serie di contesti. Tuttavia, considerato l'ambiente autonomo e distribuito dei microservizi ci sono delle casistiche particolari da considerare.

L'Unit Testing è la forma di base per garantire che la qualità del microservizio rispetti la parte di dominio che sta modellando. Il testing si basa sulla verifica delle funzionalità interne del microservizio, scrivendo test ad hoc che indichino se un qualche metodo, tale è la profondità dell'unit testing, non risponda in maniera coerente con le aspettative degli sviluppatori. È il più sempice da utilizzare, ed è capace di fare emergere bug particolarmente insidiosi dalla *codebase* sulla quale si ha lavorato.

Il Service Testing sposta l'attenzione del collaudo un gradino più in alto: ciò di cui si attesta la qualità è il microservizio in sè e la capacità di interagire con altri produttori e consumatori. Il punto centrale del testing, tuttavia, è ancora il nuovo microservizio, quindi i collaboratori esistono sotto forma di *mock*. Le funzionalità che vengono prese in esame sono la capacità di interrogare i microservizi produttori ed utilizzare propriamente le informazioni ricevute, oltre che fornire i dati richiesti ai consumatori. Mentre avere un mock per i produttori risulta abbastanza semplice, in quanto si tratta di componenti passivi che devono preoccuparsi di restituire informazioni consistenti (o non consistenti, in base agli aspetti che si stanno testando) sul canale prefissato per le comunicazioni nel momento in cui il microservizio sotto analisi decide di effettuare delle richieste, un consumatore deve essere un componente in grado di effettuare attivamente le richieste desiderate in qualsiasi momento. Un mock di consumatore risulta quindi più difficile da implementare, tuttavia il suo ruolo all'interno del Service Testing è fondamentale per garantire le funzionalità del microservizio in analisi.

Il livello più alto di testing pre-produzione è rappresentato dal Consumer-Driven Testing. Questo risponde alla necessita, nel rispettare il principio di deployment indipendente, di non danneggiare alcuna parte del sistema in produzione con il rilascio di un nuovo microservizio. La responsabilità di aderire alle caratteristiche dei produttori di informazione ricade sul microservizio stesso, motivo per il quale questi sono esclusi da questo processo di testing. Tuttavia, non è possibile testare le informazioni create ad uso dei consumatori direttamente in produzione, in quanto queste potrebbero essere fallaci, ed alterare il sistema. La soluzione ricade nei Consumer-Driven Contract (CDC): questi funzionano esprimendo le aspettative dei consumatori sotto forma di un linguaggio testabile contro le capacità del microservizio produttore del quale si stanno testando le funzionalità.

Pact[21] e Pacto [22] sono due strumenti che si occupano di interpretare il codice che genera le aspettative dei consumer ed applicare i CDC generati al produttore in questione. Dei trigger vengono generati quando le risposte dal microservizio produttore non sono conformi alle aspettative del consumatore, segnalando la presenza di operazioni fallaci nei confronti del sistema.

Post-Production

Applicare tutti i concetti appena descritti ad un microservizio che si affaccia al sistema per la prima volta potrebbe risultare sufficiente per garantirne le funzionalità: le dipendenze nei confronti dei consumatori sono soltanto teorizzate, e quelle nei confronti dei produttori sono potute essere analizzate con cura. Viene comunque applicata una *smoke test suite*, una serie di test indirizzati a confermare che il deployment del microservizio abbia avuto esito positivo. Una volta superati i test della suite, il microservizio è pronto per la produzione, e viene reso di pubblico dominio.

La *smoke test suite* non è sufficiente nei casi in cui stia venendo introdotta una nuova versione di un microservizio già in funzione nel sistema, in quanto si deve riuscire a dimostrare di non aver introdotto una subottimalità rispetto allo stato precedente del sistema. Le due tecniche principali per testare questa eventualità sono il *Blue/Green Deployment* ed il *Canary Releasing*. Entrambe si basano sul mantenere attivi contemporaneamente sia il nuovo microservizio che quello che verrà eventualmente dismesso, ma le attività performate durante la coesistenza varia leggermente. Nel *Blue/Green*, tutto il traffico viene immediatamente rediretto al nuovo microservizio non appena si siano conlusi i processi della smoke test suite. Il vecchio microservizio resta comunque presente, ma nascosto agli occhi del sistema. Se ad un certo punto la nuova forma si dimostra meno produttiva, per esempio dimostrandosi incapace di sorreggere la stessa quantità di lavoro, gli indirizzi di rete vengono immediatamente ricablati per puntare al vecchio. Nel canary invece, il carico produttivo viene trasportato gradualmente verso il nuovo microservizio, con il vantaggio di riuscire a calibrare meglio i paramtetri di funzionamento. Con il canary è possibile valutare più fattori sulla base del carico dirottato, come la validità di eventuali nuovi algoritmi o che il nuovo stack tecnologico abbia portato i benefici di performances desiderati. Dopo un'analisi completa dei risultati, le versioni vecchie possono essere dismesse.

2.6.2 Versionamento

Il versionamento è la tecnica che permette di affrontare evoluzioni nel dominio applicativo o errori nell'interpretazione dello stesso. L'unica possibilità all'accadere di questi eventi è cambiare il comportamento dei microservizi che gestiscono la logica delle parti oramai non aggiornate, o fallaci, per abbracciarne uno che rispecchi veramente un modello della realtà. Nella pratica, questo si traduce in un cambiamento nell'interfaccia esposta, con la rimozione di operazioni riconosciute come appartenenti ad un diverso Bounded Context e quindi

spostate sotto la responsabilità di un altro microservizio, o con un cambiamento nella semantica esposta. Il problema che si presenta è che un cambiamento netto e repentino danneggerebbe tutti i consumatori del microservizio. Allo stesso tempo, sarebbe possibile riuscire a tracciare tutte le dipendenze rotte per risolverle soltanto finchè il numero di microservizi rimane basso, ma l'esperienza dimostra come questo caso sarebbe più un'eccezione, che la regola: il numero di microservizi attivi in un sistema complesso cresce molto velocemente, arrivando a sfondare l'ordine delle migliaia [11]; inoltre, questo violerebbe il principio del deploy indipendente.

La soluzione offerta dal versionamento consiste nel creare una nuova versione del microservizio, assegnarle un nuovo numero di release basandosi sulle regole del versioning semantico[1] e nel caso di versioni incompatibili con i fruitori attuali si mantengono in attività entrambe le versioni per un periodo di tempo abbastanza lungo perchè i microservizi clienti abbiano la possibilità di migrare verso la versione successiva.

Questa soluzione trasporta con sè una problematica particolarmente insidiosa: vengono a coesistere due microservizi dalla logica interna molto simile. Fino al momento della destituzione della versione più vecchia, qualsiasi cambiamento dovrà essere replicato su entrambe le versioni. Questo mina leggermente il concetto di deploy indipendente, in quanto un cambiamento in un microservizio obbliga al cambiamento nella versione precedente corrispettiva, ma è un compromesso accettabile per la corretta fruizione delle funzionalità offerte ai consumatori.

Dal punto di vista di un utilizzatore del microservizio che vuole proteggersi dalla necessità di doversi preoccupare del versionamento, esiste una tecnica definita Tolerant Reader [19]: prevede che di fronte ad una serie di valori presentati al microservizio cliente, questo debba dipendere soltanto da quelli che effettivamente vengono usati all'interno della computazione. Supponendo i campi A, B, e C come risultato dell'invocazione dell'operazione X su di un microservizio produttore, e considerando necessari per il microservizio cliente solo A e C, andrebbe evitato sia il parsing dell'intero messaggio come un unico blocco, sia l'ammissione di un qualche tipo di conoscenza nei confronti del campo B. Se in una versione futura il campo B dovesse essere eliminato, o il campo D aggiunto, entrambi gli scenari darebbero luogo ad un microservizio cliente malfunzionante, senza che la nuova versione del microservizio produt-

[1] Il versioning semantico prevede che il numero di versione sia assegnato considerando la forma `MAJOR.MINOR.PATCH`, dove un incremento in `MAJOR` indica l'incombatibilità con i fruitori della versione precedente, in `MINOR` indica l'aggiunta di funzionalità che possono essere integrate con versioni precedenti mentre incrementando `PATCH` si indica un nuovo deploy con il quale si applicano correzioni a bug per la versione corrente

tore possa fare niente per evitare il disguido. Leggendo soltanto A e C come campi fini a loro stessi, il microservizio si scherma contro possibili cambiamenti dirompenti nei confronti delle proprie funzionalità offerte.

2.7 Resilienza agli errori

I gradi di libertà concessi ad un team di sviluppo per la realizzazione di un microservizio sono direttamente proporzionali al numero di librerie con le quali ci si approccia per risolvere le problematiche di ogni strato del software. Aggiungendo a tutto questo la componente distribuita, ciò che si viene a costituire è un sistema in cui la possibilità di errori diventa talmente alta da non poter più lavorare con l'ottica di eliminarli alla radice, quanto con la loro accettazione ed integrazione all'interno del microservizio. È importante notare che con errore si intendono principalmente i malfunzionamenti a livello di comunicazione, in quanto questa è l'attività principale che viene svolta a livello di sistema: un errore all'interno di un singolo microservizio avrebbe importanza solo nel momento in cui tale errore venisse propagato verso gli altri, così come il fallimento di una componentistica hardware viene vista come un problema dell'intero sistema solo nel momento in cui generi informazioni sbagliate in risposta alle richieste effettuate. La rete, infatti, è il substrato che fa da collante fra tutti i microservizi del sistema, e proprio la rete sarà la causa delle problematiche più ricorrenti e complesse da risolvere.

L'insieme delle "Otto assunzioni sbagliate della computazione distribuita" fornisce un ottimo punto di partenza per regolare la comunicazione fra i microservizi, specialmente cercando di ottenere la massima autonomia: vengono elencati errori di cui è talmente famoso il potenziale distruttivo da non potere fare a meno di non considerare cosa succederebe allo stato di resilienza del microservizio senza i controlli adeguati nei confronti della rete.

Riassumendoli:

1. **La rete è infallibile** – tutti i microservizi devono considerare l'evenienza che le richieste effettuate potrebbero non venire evase, o essere danneggiate. L'inaffidabilità della rete è uno dei motivi per cui nella fase di modellazione si cerca di rendere i microservizi il meno dipendenti possibile dalla comunicazione intensa con altri: tali conversazioni potrebbero essere interrotte in un momento qualsiasi, per un motivo qualsiasi, e ogni risultato raggiunto andrebbe perso.

2. **La latenza è nulla** – la latenza è il prezzo imposto dalla fisica per ottenere la possibilità di distribuirsi nello spazio. Oltre ad essere decisa-

mente non nulla, è anche variabile, e solo vagamente calcolabile a priori. La possibilità che il suo valore sia troppo alto (e che la richiesta inviata sia da considerare come persa) è da tenere sempre in considerazione, sia nell'effettuare chiamate sincrone che asincrone.

3. **La banda è infinita** – corollario dell'assunzione precedente. La banda rappresenta la massima quantità di dati presenti sulla rete in un determinato istante di tempo. Se si cerca di utilizzare in maniera costante più banda di quella sostenibile dall'infrastruttura di rete, la congestione è tale da causarne un blocco temporaneo, con la conseguente impossibilità di comunicare.

4. **La rete è sicura** – la rete è strutturata secondo lo stesso principio di mezzo di comunicazione agnostico con il quale è consigliato di sviluppare i microservizi. Nella rete, tale principio viene spinto al punto tale da richiedere che la sicurezza delle informazioni trasmesse vange gestita fra i partecipanti allo scambio: qualsiasi trasmissione effettuata senza adottare misure di precauzione è da considerarsi di pubblico dominio. Nel momento in cui viene effettuato il *deploying* di un microservizio, questo è da considerarsi accessibile a tutti, e il livello di sicurezza deve essere adeguato all'importanza del servizio offerto.

5. **La topologia di rete non cambia** – è impossibile basare un comportamento programmatico sulla struttura della rete sulla quale si vuole effettuare della computazione. La struttura della rete è variabile per sua stessa natura, e molti dei protocolli di routing in atto fra i nodi comunicativi cambiano attivamente la struttura percepibile della rete.

6. **Esiste un solo amministratore** – uno dei concetti alla base di Internet è che molte reti e sottoreti collaborino per fornire la più ampia connettività possibile. Ognuna di queste reti è gestita in maniera autonoma ed eterogenea l'una dall'altra, con le proprie regole, i propri limiti e i propri divieti.

7. **Il costo di trasporto è zero** – perchè ci sia comunicazione, è necessario che esista un'infrastruttura fisica, ed è necessario che venga pagata l'energia elettrica affinchè tale infrastruttura rimanga operativa. Concependo un'applicazione distribuita è necessario tenere in considerazione che potrebbe doversi ritenere necessario acquistare l'accesso alla rete.

8. **La rete è omogenea** – i nodi che costituiscono la rete sono sistemi completamente diversi l'uno dall'altro, e nel momento in cui ci si trova nella necessità di dovercisi interfacciare, bisognaa concepire soluzioni

che si leghino il meno possibile a tecnologie particolari, ma si possano applicare in qualsiasi contesto.

Nonostante tutti siano fondamentali, alcuni diventano rilevanti solo con l'aumentare delle responsabilità dell'applicazione; i primi due coprono il ciclo di vita di qualsiasi sistema distribuito fin dalle prime fasi della progettazione.

Considerando il caso in cui i messaggi fra microservizi diventino corrotti (e aggiungendo che tali messaggi potrebbero in realtà già partire corrotti, a causa di problemi interni al microservizio mittente, o il cambiamento nella logica del formato con cui le informazioni vengono trasmesse) una soluzione è offerta dal concetto di AntiCorruption Layer[1], con il quale si intende, nell'ambito dei microservizi, un modulo che sia il primo ricevente di ogni informazione entrante, occupandosi di garantirne la correttezza nei confronti del Bounded Context interno al microservizio. Le richieste che giungono con un payload malformato, o che non rispettano la forma richiesta, possono quindi essere individuate e scartate, o eventualmente venire trasformate in una forma comprensibile, anche se meno espressiva, al microservizio ricevente.

Nel caso in cui le richieste effettuate non ricevano risposta per un lasso di tempo abbastanza lungo perchè queste si considerino perse, è possibile utilizzare tre tecniche, ognuna delle quali copre un aspetto specifico del problema della latenza. La gestione dei timeout di tali richieste, nella sua semplicità, è il primo: aspettare troppo poco vorrebbe dire ripetere una richiesta che possibilmente era andata a buon fine, aspettare troppo tempo per avere una risposta negativa significherebbe rallentare il sistema inutilmente, non impostare un limite significa potenzialmente fermare l'intero sistema. La *best practice* consiste quindi nell'impostare un timeout default a qualsiasi richiesta che oltrepassi i limiti del microservizio, e regolarne la durata basandosi sui log di comportamento.

Nel caso di un microservizio destinatario che utilizzi costantemente tutto il timeout senza rispondere, è possibile utilizzare il pattern noto come *Circuit Breaker*[20] (interruttore), con il quale si disabilita lo svolgimento effettivo della richiesta e si risponde direttamente in maniera negativa, permettendo all'esecuzione del chiamante di riprendere più velocemente. Lo stesso modulo che funge da interruttore si occuperà di effettuare delle chiamate concrete, di tanto in tanto, e non appena il microservizio destinatario risponderà nuovamente, ristabilirà il flusso di controllo corretto. La terza tecnica consiste nel prevenire che i problemi che hanno portato alle prime due possano avvenire: il pattern *Bulkhead* [20] (paratia) consegna ad ogni modulo che necessiti di comunicare con l'esterno il proprio *pool* di connessioni, in maniera tale che dei destinatari poco performanti non possano portare alla congestione totale delle risorse del microservizio mittente, ma solo della parte con la quale comunicano.

2.8 Monitoring pervasivo

Un riassunto semplicistico del sistema delineato finora descrive una moltitudine di applicazioni mirate a compiere una singola attività, realizzate tramite gli stack tecnologici più disparati, ognuna in esecuzione su un host fisico o virtuale ad hoc, soggette ad evoluzioni continue, che comunicano tramite chiamate sincrone e asincrone tramite la rete.

È indispensabile riuscire ad avere una visione d'insieme tale da permettere l'individuazione immediata della causa di un problema, poichè data la quantità di interconnessioni fra microservizi e la complessità generale del sistema, effettuare un controllo dei log microservizio per microservizio, host per host, applicazione per applicazione, magari contemporaneamente, diventa semplicemente impossibile da gestire.

Le metriche da considerare sono molteplici, e principalmente suddivise in tre livelli di astrazione crescente dal punto di vista dell'osservatore. Da un punto di vista tecnico, è necessario sapere quale sia lo stato corrente degli host sui quali stanno girando i microservizi, e di quali e quante risorse questi hanno bisogno. Questo diventa vitale nel momento in cui si abbia più di un microservizio in funzione sullo stesso host o si possano dedicare risorse centellinate per massimizzare i settori di virtualizzazione di una macchina fisica, ma è sempre utile per conoscere i veri bisogni del microservizio contenuto in tutte le sue fasce di utilizzo. Le metriche di servizio raccontano la storia delle capacità comunicative, di come e quante volte si abbiano avuti problemi nello scambio di messaggi, e cosa e quanto spesso è stato contattato in maniera priva di errori. Queste danno un'indicazione precisa sull'impatto che l'aggiornamento di un microservizio possa avere nei confronti dell'operatività del sistema, oltre che a far emergere i punti in cui è richiesta più attenzione nello sviluppo (magari per l'alto tasso di errori, magari per il cresciuto numero di utenti che desidera un particolare servizio) e quali è possibile lasciare ad un secondo momento.

La metrica più importante al quale il sistema di monitoring deve essere in grado di accedere consiste in: *sta funzionando tutto?* [23] Il comportamento del sistema in funzione è analizzabile attraverso una tecnica chiamata semantic monitoring. Periodicamente, vengono inseriti insiemi di richieste create ad-hoc: l'obiettivo è coprire tutto il range di responsabilità del sistema in una volta, per assicurarsi che eventuali nuove versioni non abbiano tralasciato sottigliezze nella gestione di alcuni casi limite. I risultati forniscono la metrica di funzionamento richiesta.

Avendo uno strumento che tiene traccia dei fallimenti hardware, software e di rete, diventa possibile determinare la posizione del guasto e quali conseguenze siano in atto. Il tassello ancora mancante è la motivazione che a

portato ad un tale guasto: risalire la catena di errori fra chiamate asincrone risulta molto più complesso che leggere lo stacktrace causato da un'eccezione. Una soluzione consiste nell'aumentare l'informatività di ogni log con un ID di correlazione: questo verrebbe posto come un *nonce*[2] all'interno del sistema e correlato ad una richiesta. La propagazione ed il logging degli effetti di questa richiesta trasporterebbe con se l'ID di correlazione, permettendo di consegnare una paternità agli effetti delle chiamate fra i vari microservizi e ricreare l'intero percorso della richiesta, dall'origine al guasto.

[2]Un numero arbitrario usato soltanto una volta nell'intero ciclo di vita del sistema

Capitolo 3

Caso di studio – TraumaTracker orientato ai microservizi

In questo capitolo si cercheranno di integrare tutti i principi discussi finora nell'infrastruttura attuale del progetto TraumaTracker, mettendo in uso le tecnologie necessarie finora discusse e valutandone le qualità.

3.1 TraumaTracker Premium

Un principio minore, non elencato ma comunque riportato da tutte le fonti che si preoccupano di fornire una descrizione introduttiva ai microservizi, consiste in una riflessione estremamente accurata sull'effettiva necessità di stravolgere la propria linea di produzione tramite l'approccio a microservizi. Per ottenerne i benefici diventa obbligatorio farsi carico della gestione di un sistema dall'infrastruttura e logica estremamente complessa che segue una visione basata su principi più che una vera e propria architettura. Bisogna quindi dimostrarsi pronti a pagare quello che è stato definito come il *MicroservicesPremium* [24], ovvero il costo iniziale intrinseco dell'affrontare tutta l'impostazione di una linea di sviluppo e produzione strettamente basata sui principi espressi nel secondo capitolo di questa dissertazione. Considerandoli ordinati per apporto crescente al costo del *premio* nel quale si imbatterà il TraumaTracker, il primo posto viene occupato dal principio della Suddivisione atomica del dominio (paragrafo 2.2.1). La semplicità nell'affrontare tale problematica deriva dal punto di applicazione del concetto di Bounded Context, in quanto ciò che si cerca di isolare è racchiuso all'interno di funzionalità offerte al team medico, e non delle responsabilità del team stesso. Un primo livello di microdominii consiste quindi nella capacità di raccoltà di informazioni di base, con l'aggregazione delle quali riuscire a comporre le funzionalità di un microservizio che gestisca i *report* di resuscitazione.

La Decentralizzazione pervasiva (paragrafo 2.5) potrebbe già essere integralmente rispettata, in quanto tutti gli sviluppatori che hanno contribuito finora al progetto hanno sempre realizzato servizi full stack che rispondono e interagiscono tramite REST, utilizzando Json come formato di comunicazione. Tuttavia, effettuando lo shift verso il concetto dei microservizi, è vitale effettuare una revisione delle librerie incorporate nello sviluppo per assicurarsi che non esistano accoppiamenti impliciti nel portare a termine i compiti richiesti. Per gli stessi motivi risulta già parzialmente soddisfatto anche il principio di Segregazione implementativa (paragrafo 2.4), in quanto l'unico modo per accedere alle informazioni elaborate dai servizi è tramite le richieste pubblicate.

Aderire al principio del Deployment indipendente pone le prime problematiche interessanti nei confronti della struttura attuale del progetto TraumaTracker. Effettuare il deploy di ogni microservizio all'interno di un host dedicato potrebbe risultare troppo costoso in termini di risorse e tempo necessario per ottenerle. Le macchine utilizzate per fornire le funzionalità del sistema sono messe a disposizione dal dipartimento informatico dell'Ospedale Bufalini, che pur sostenendo lo sviluppo del progetto con il massimo impegno, ha come priorità principale il corretto funzionamento dell'infrastruttura informatica dell'ospedale.

La soluzione ricade quindi nell'utilizzare un pool ristretto di macchine fisiche e virtuali sulle quali sfruttare le capacità di gestione di container di Docker, per ottenere dei contesti di esecuzione isolati ed autonomi sui quali deployare i microservizi che compongono la logica del TraumaTracker. Questo riduce leggermente la difficoltà necessaria ad impostare il monitoring globale del sistema, in quanto è possibile realizzare *ad-hoc* le immagini messe in esecuzione nei container affinchè tutte utilizzino la stessa versione del tool di registrazione delle metriche del sistema. Resterebbe comunque necessario gestire la mole di dati inviati da ogni microservizio in maniera tale che si riesca a tracciare ogni azione effettuata in risposta alle richieste ricevute.

Ciò che rivoluzionerebbe completamente il deployment del TraumaTracker consiste nell'adesione alla filosofia Continuous, rispettando di conseguenza il principio di Cultura dell'automazione. La procedura attualmente in uso è totalmente manuale, in particolar modo il testing e l'integrazione con il resto del sistema. Va realizzato, o noleggiato *as-a-service*, un server che gestisca le pipeline di integration, alle quali diventa possibile affidare i Consumer Driven Contract. Questi vanno modellati e integrati in quello che sarebbe il primo refactoring dei servizi attuali, per avvicinarsi all'approccio a microservizi: il compromesso ormai noto consiste nella complessità di realizzazione di un Continuous Server autonomo sia complesso, affittarne uno dispendioso. Nell'idea di realizzarlo autonomamente la scelta ricadrebbe su Jenkins, un tool di Continuous Delivery open source, con un'integrazione eccellente nei con-

fronti di Docker. La vera forza dell'utilizzare un approccio automatizzato si ha nello smorzare il costo del partecipante più imponenente del *Premium*: la resilienza da qualsiasi forma di interruzione del servizio. L'approccio automatizzato, unito al monitoring costante degli host che costituiscono il back-end, permetterebbe la gestione di eventi come il fallimento di una macchina, con *provisioning* immediato di un back-up dormiente o ancora non completamente istanziato. Grossi carichi di lavoro potrebbero portare al *lockdown* delle macchine, e la replicazione dei servizi più importanti dovrebbe essere gestita in maniera coerente. Diventerebbe quindi necessario aggiungere un ulteriore *layer* nella gestione dei container utilizzati per deployare i microservizi, quello dell'orchestrazione: Kubernetes, sviluppato da Google, automatizza il deploy dei container tenendo conto di fattori come la disponibilità degli stessi, il carico che ricevono, e nel caso di *load balancer* già presenti sui microservizi replicati è possibile strategicizzare l'orchestrazione in casistiche particolari.

La quantità di cambiamenti a cui il progetto TraumaTracker dovrebbe far fronte è quindi notevole, e tali cambiamenti riguardano molte parti della sua architettura. Non è neanche detto che questi siano gli unici che ci si debba aspettare, data la forte natura evolutiva che ci si permette con l'approccio ai microservizi, oltre che quella del dominio applicativo medico in sè. Il processo che nasce dalla volontà di adesione a questi principi per ottenere dei risultati nella gestione dello sviluppo e della *fault-tolerance* massivi come quelli di Netflix, ad esempio, è lungo e tutt'altro che immediato.

3.2 Applicazione dei principi

Nella totalità delle procedure che seguono si considera di trovarsi nel caso limite in cui si possiede un'unica macchina sulla quale effettuare il deploy dell'intero sistema. Le funzionalità vengono introdotte in maniera incrementale rispetto alle necessità del TraumaTracker, sia per acquisire le capacità attuali che per aderire all'approccio a microservizi.

3.2.1 Deploy indipendente

La necessità di isolazione fra sistemi e l'autonomia fra i Bounded Context è facilmente raggiungibile anche senza avere a disposizione una macchina dedicata per ogni microservizio in esecuzione. I recenti progressi nello sfruttamento della tecnologia dei container, concretizzati dalla piattaforma Docker permettono il deploy di immagini software *lightweight* in un ambiente che garantisce la massima isolazione e sicurezza. La tecnologia dei container si basa su un'astrazione aumentata rispetto alle macchine virtuali 'standard'. Ciò che viene

```
FROM ubuntu:18.04
RUN apt-get update
   && apt-get install -y openjdk-8-jdk
   && apt-get install -y mongodb
ENV JAVA_HOME /usr/lib/jvm/java-1.8.0-openjdk-amd64
ENV PATH="/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin:${PATH}"
ENV GRADLEOPT --console-plain
```

Listato 3.1: Dockerfile che gestisce la configurazione di un'immagine riutilizzabile che include Java 8 e MongoDB

virtualizzato non è solo la componente hardware di una macchina, sfruttando le risorse del sistema host, quanto le intere capacità di un sistema operativo *NIX. Il vantaggio risiede nell'avere l'intero pool di librerie e funzionalità di un sistema condivise fra i diversi container, che risultano quindi essere soltanto l'insieme di codice, applicazioni di terze parti e configurazioni necessarie all'operatività di runtime: il sistema operativo viene fattorizzato a monte, ed è comune a tutti. I container vengono considerati come processi autonomi sul sistema operativo 'guest', ognuno dei quali possiede il proprio runtime. L'isolamento e la messa in sicurezza vengono certificati dalla piattaforma Docker.

Dockerizzare un microservizio consiste nel ricreare l'ambiente all'interno del quale il sistema si aspetta di trovarsi al momento del deploy racchiudendolo all'interno di un file di configurazione chiamato *Dockerfile*. Una delle funzionalità più importanti dalle quali dipende il TraumaTacker è la lettura di parametri vitali in tempo reale, compito assolto dal TTGatewayService.

Le dipendenze di questo modulo risultano essere:

- Un sistema Linux sul quale runnare.
- Java 7 o superiori.
- MongoDB.

Queste dipendendenze risultano essere condivise anche da altri servizi nel sistema, motivo per il quale verrà costruita un'immagine di base, che poi verrà raffinata con una configurazione *ad-hoc* per ogni modulo. Il Dockerfile, tale è il nome canonico dato allo script dichiarativo tramite il quale Docker viene a conoscenza delle operazioni da effettuare, che definisce l'immagine di base è contenuta nel listato 3.1, la raffinazione che definirà il GatewayService al listato 3.2

Il comando FROM stabilisce quale sia l'immagine di partenza sulla quale effettuare le successive operazioni. È stato scelto Ubuntu in quanto si ha la

```
FROM TraumaTracker/BaseService
COPY ./ /GatewayService/
WORKDIR /GatewayService/
RUN tr -d '\r' < gradlew > new-gradlew
   && mv gradlew gradlew-old
   && mv new-gradlew gradlew
   && chmod +x gradlew
RUN ./gradlew build
EXPOSE 8082:8082 TCP
ENV USERNAME manuel.bonarrigo@studio.unibo.it
CMD ./gradlew run $GRADLEOPT
```

Listato 3.2: Dockerfile che gestisce la configurazione del GatewayService

certezza del successo nell'integrazione, dati i numerosi test svolti a riguardo. I comandi COPY e WORKDIR, rispettivamente, arricchiscono l'immagine con il codice sorgente del progetto e fanno si che al momento dell'avvio di un terminale, la cartella root del codice sorgente sia la working directory. RUN permette di aggiungere funzionalità al sistema base, ed in questo frangente installa le dipendeze richieste tramite *apt-get* e normalizza il contenuto del file gradlew all'esecuzione su UNIX. Il comando EXPOSE segnala a Docker il mapping delle porte del sistema host alle porte del container, attraverso le quali questo dovrà essere raggiungibile: si troveranno già disponibili e configurate al termine del deploy. I comandi ENV definiscono delle variabili di sistema: essendo al momento io il curatore dell'immagine l'USERNAME presenta il mio nom, e GRADLEOPT viene usato semplicemente come contenitore di un parametro. CMD specifica i comandi da lanciare una volta ultimata la fase di boot del container. In questo caso viene lanciato lo script gradle che lancia la compilazione ed avvia l'applicazione utilizzando come parametro GRADLEOPT, in maniera tale da normalizzare la struttura della console.

Le immagini vengono rispettivamente costruite tramite i comandi

```
$ docker build -t TraumaTracker/BaseService .
```

e

```
$ docker build -t TraumaTracker/GatewayService .
```

dove l'unico parametro che differisce è il nome che verrà assegnato all'immagine, e l'assenza di riferimenti a dove prendere le informazioni sottintende di trovarsi già nella directory di cui l'immagine sarà lo specchio.

La piattaforma Docker permette di simulare reti virtuali che connettano i

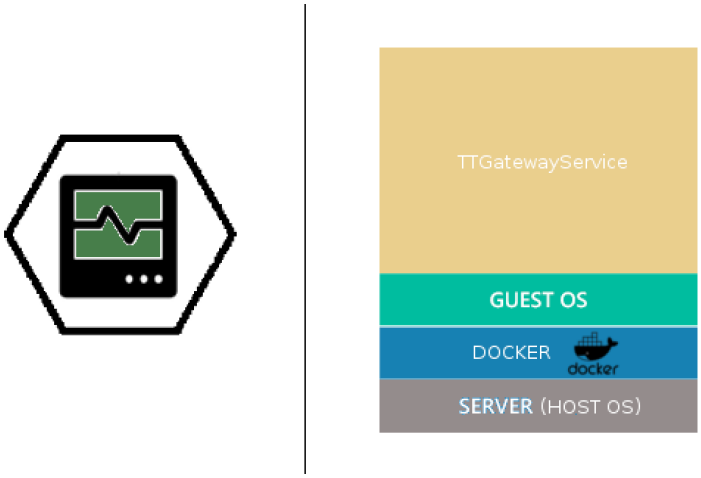

Figura 3.1: 1 - L'ecosistema di microservizi attualmente in esecuzione, con la rispettiva struttura Docker

container in esecuzione su un'unica macchina, usando le quali diventa possibile testare le capacità di comunicazione fra parti diverse di un sistema distribuito senza dover decentralizzare veramente i moduli software

```
$ docker network create traumanet
```

Da questo momento in poi, l'immagine pre-costruita (tramite la quale diventa possibile ottenere le funzionalità del GatewayService su qualsiasi infrastruttura capace di eseguire e gestire container Docker) e l'esistenza di una rete privata tramite la quale effettuare effettivo scambio di messaggi costituiscono una forma iniziale della nuova infrastruttura.

```
$ docker run -d --network=traumanet TraumaTracker/GatewayService
```

Il comando appena mostrato mette in esecuzione (sotto forma di daemon) l'intero GatewayService in un container apposito, isolato dal resto dei sistemi che verranno messi in esecuzione.

L'immagine 3.1 dimostra la struttura iniziale del contesto TraumaTracker orientato ai microservizi, insieme all'infrastruttura finora costruita su Docker.

3.2.2 Cultura dell'automazione

Gli aspetti tecnici della Continuous Delivery consistono nella realizzazione di un server dedicato nel quale confluiscono tutti i commit relativi al software, eventualmente già in produzione. Sul server, un tool appositamente configurato effettua tutti i test che il modulo software espone, ed eventualmente altri dipendenti da caratteristiche indipendenti dell'infrastruttura.

La scelta del software a cui affidare queste responsabilità è ricaduta su Jenkins, in quanto gode di una rinomata integrazione con Docker: in particolare, Jenkins è capace di istanziare container Docker *on-the-fly* per testare particolari aspetti dell'applicazione, rendendo completamente automatizzate e prive di *side-effects* molte fasi di test, come il Service Testing (paragrafo 2.6.1), per i quali è necessaria l'interazione con la rete. Le capacità della piattaforma Docker di istanziare reti virtuali *ad-hoc*, non soggette alle influenze del livello fisico, tornano estremamente utili per evitare aspetti non deterministici nelle fasi di testing.

Istanziare un container Jenkins risulta particolarmente facile, avendo un'immagine disponibile su `https://hub.docker.com`. I comandi da invocare si riassumono in:

```
$ docker run --network traumanet -p 8080:8080 -p 50000:50000
    jenkins/jenkins:lts
```

All'indirizzo `http://localhost:8080` viene pubblicata l'interfaccia web attraverso la quale interagire con Jenkins. Esistono numerose possibilità e combinazioni di funzionalità che permettono di costruire pipeline di Continuous Delivery: quella analizzata in questa dissertazione consiste nell'utilizzo della Declarative Pipeline, unita alla possibilità di effettuare comunicazioni tramite Jenkins e il CMS utilizzato.

La Declarative Pipeline è un'aggiunta recente alle capacità dello strumento, e consiste nella possibilità di specificare il comportamento che si desidera vedere seguito e certificato dalla pipeline in un file chiamato Jenkinsfile (un esempio del quale è al listato 3.3), posizionato all'interno della cartella root del progetto tracciato dal CMS. Questo viene analizzato ad ogni commit, e viene costruita una pipeline che rispecchi le necessità richieste. È costituito da due parti principali – *agent* e *stages*. *Agent* è il componente dello script al quale passare i parametri necessari alla creazione di container 'usa e getta': *any* si riferisce alla possibilità di vedere risolte le proprie richieste in qualsiasi forma di *worker* Jenkins trovi più appropriata; *docker* e *dockerfile* permettono, rispettivamente, di specificare le opzioni con le quali instanziare i container direttamente all'interno del Jenkinsfile, o lasciare che il Dockerfile specificato all'interno del repository CMS venga caricato dinamicamente.

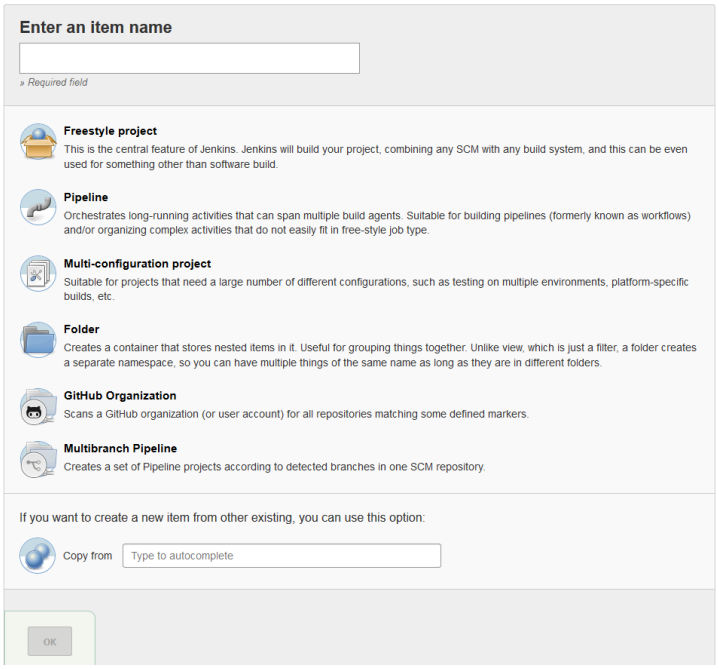

Figura 3.2: Una porzione delle funzionalità offerte da Jenkins in supporto alla Continuous Delivery tramite pipeline

I diversi *stage* permettono di esplicitare le fasi in cui dividere la pipeline: vanno definiti gli *steps* attraverso i quali portare a termine le operazioni. Tools come Gradle e Maven giocano un ruolo chiave in tutto questo, in quanto diventa possibile aggiungere un altro layer di automazione specificando le modalità con le quali il progetto deve essere compilato, quali sono i test di cui si richiede l'esecuzione e in che maniera vada eseguito per testarne il comportamento in esecuzione.

La comunicazione fra Jenkins e il CMS può avvenire principalmente in due modalità. Se il CMS lo consente, è possibile specificare un *web-hook* al quale inviare il codice sorgente ad ogni commit: l'ulteriore porta esposta nella configurazione di Jenkins serve a permettere questo tipo di servizio. È anche possibile lasciare che Jenkins effettui un polling periodico sul CMS, e rilevi automaticamente la presenza di cambiamenti sul repository di codice.

La figura 3.3 mostra gli effetti dell'avere un repository Git linkato a Jenkins tramite *webhook*. Ad ogni commit viene invocato lo script Jenkinsfile definito nelle pagine precedenti, e viene accertato che ogni singolo stage venga portato a termine senza errori. Le pipeline rosse evidenziano come fino al penultimo commit il fallimento sia avvenuto nella fase di Gradle Build. Vengono anche mostrate le tempistiche necessarie al completamento dell'intera pipeline, indi-

```
pipeline {

  agent {
    dockefile {
      filename Dockerfile
      dir /
      label gatewayservice-test-pipeline
    }
  }

  stages {
    stage('Gradle Build') {
      steps {
        script{
          if(isUnix()) {
            sh './gradlew clean build'
          } else {
            bat 'gradlew.bat clean build'
          }
     ..
    }
    stage('Gradle Test') {
      steps{
        script{
          if(isUnix()) {
            sh './gradlew clean test'
          } else {
            bat 'gradlew.bat clean test'
          }
     ..
    }
    stage('Gradle Deploy') {
      steps{
        script{
          if(isUnix()) {
            sh './gradlew clean run'
          } else {
            bat 'gradlew.bat clean run'
          }
     ..
}
```

Listato 3.3: Jenkinsfile per la costruzione di una pipeline dinamica

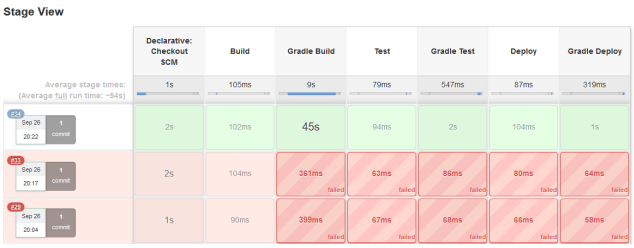

Figura 3.3: Un esempio di pipeline, applicata ad un repository Git

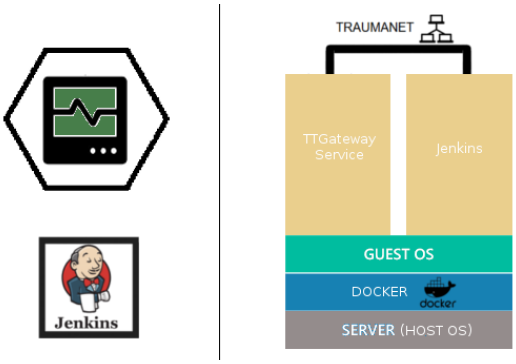

Figura 3.4: 2 - Ogni cambiamento ai microservizi esistenti si considera aver superato tutte gli step della pipeline di Continuous Delivery

pendentemente dal successo o meno. In entrambi i casi, è presente una pletora di metodologie per comunicare agli sviluppatori che hanno invocato la pipeline il risultato, data la possibilità che la quantità di operazioni da svolgere sia troppo lunga per aspettare che siano tutte portate a compimento.

3.2.3 Monitoring pervasivo

L'approccio Continuous pone le basi per per sfruttare le possibilità di innovazione costante desiderata dall'utilizzo dei microservizi. Il comportamento di ognuno di questi può venire alterato ad ogni release, e l'unica maniera di determinare se i cambiamenti apportati siano risultati in un *outcome* positivo è avere un tool di monitoraggio che mantenga uno storico delle informazioni principali, e di come queste cambino nel tempo. Lo strumento scelto per gestire questa responsabilità è Graphite, un tool capace sia di gestire la raccolta, che il plotting, che l'aggregazione di dati, gestendo autonomamente la concorrenza e la coesione delle informazioni ricevute.

Runnare un container che contenga un server Graphite necessita di pochi, semplici, comandi, date le potenzialità della piattaforma Docker. Questa immagine contiene già tutti i parametri di configurazione appropriati perchè il server sia perfettamente operativo non appena termini la fase di 'boot'. Nonostante in futuro sarà necessario avere una prelazione più forte su questo tipo di parametri, per il testing esplorativo in atto in questa dissertazione è sufficiente che il tool sia operativo.

```
$ docker run -d \
  --name graphite \
  --network traumanet \
  --restart=always \
  -p 80:80 \
  -p 2003-2004:2003-2004 \
  -p 2023-2024:2023-2024 \
  -p 8125:8125/udp \
  -p 8126:8126 \
  graphiteapp/graphite-statsd
```

La porta 80 serve l'interfaccia web Carbon, il modulo di Graphite attraverso il quale gestire la sicurezza del server e visualizzare le statistiche ricevute, in base ai privilegi dell'account con il quale ci si logga. Le numerose altre porte sono necessarie per effettuare la comunicazione delle metriche che si desidera vedere registrate, una per formato con il quale queste vengono inviate. Graphite è infatti capace di gestire multipli formati di dato, per ognuno dei quali un pool di connessioni dedicato rimane in ascolto su una particolare porta.

- **PlainText** – Il protocollo PlainText viene accettato sulla porta 2003, o sulla 2023 se è richiesta la capacità di aggregazione con altre fonti. Il formato di invio consiste in `path.to.monitorize value timestamp`, ed è unicamente possibile l'invio di un valore alla volta.

Figura 3.5: Monitoring di un Ubuntu containerizzato, con in esecuzione uno stress test su memoria RAM e processori

- **Pickle** – Il protocollo Pickle è un'evoluzione del protocollo PlainText, ed offre la possibilità di inviare *batch* di informazione in una singola chiamata. Il formato è `[(path.to.monitorize (timestamp, value)), ...]`. Viene registrato dalla porta 2004, o dalla porta 2024 se necessaria l'aggregazione da fonti multiple

- **AMQP** – È possibile sfruttare l'implementazione preferita di Advanced Message Queuing Protocol, ma Graphite ne permetterà semplicemente l'utilizzo, non avendo altri strumenti di integrazione a disposizione.

La forza di Graphite sta nel parametro 'path.to.monitorize', completamente arbitrario se non per i punti, che delineano un albero di cartelle tramite il quale ogni processo autorizzato può esprimere qualsiasi *value* desideri vedere registrato. Esistono numerosi tool in grado di estrapolare le metriche di utilizzo delle risorse da un host, ma dato che andrebbero comunque trasformate nei protocolli precedentemente discussi, si è pensato di utilizzare semplicemente le pipeline *NIX per ottenere i dati in formato grezzo dalla macchina, ed inviarli tramite protocollo PlainText. Essendo puro codice, eseguibile direttamente in una Bash, si ha il vantaggio di poterlo 'iniettare' direttamente nei sistemi tramite la costruzione di Dockerfile appropriati, con i quali configurare automaticamente tutti i propri container sui quali vengono eseguiti i microservizi. I due script(listati 3.4 e 3.5) inviano costantemente i valori percentuali di memoria RAM occupata e CPU utilizzata dal microservizio al server Graphite.

```
while true;
   do
      echo -n
         "ubuntu0.used_memory_MB:
            ‘free -m
            | sed -n ’s/^Mem:\s\+[0-9]\+\s\+\([0-9]\+\)\s.\+/\1/p’‘
         |c"
      | nc -w 1 -u 172.20.0.2 8125;
done
```

Listato 3.4: Script in bash per l’invio della quantità di memoria utilizzata dal container al server Graphite

```
while true;
   do
      echo -n
         "ubuntu0.cpu_percentage:
            ‘mpstat
            | grep -A 5 "%idle"
            | tail -n 1
            | awk -F " " ’{print 100*100 - $13*100}’a‘
         |c"
      | nc -w 1 -u 172.20.0.2 8125;
done
```

Listato 3.5: Script in bash per l’invio della quantità di memoria utilizzata dal container al server Graphite

Nell’immagine 3.5 viene mostrato un esempio del monitoring effettuato su un container Docker di Ubuntu 18.04, con in esecuzione uno stress-test di venti minuti: entrambi i risultati vengono pubblicati all’interno della cartella ubuntu0, ma ognuno in un file diverso, chiamato a seconda delle richieste effettuate al servizio Graphite.

Il comportamento dell’intero sistema non è stato alterato per nulla, in quanto l’introduzione dei tool di Continuous Delivery e di Monitoring non impatta sulla struttura del sistema software, quanto su quella del processo produttivo e di analisi delle operazioni elaborate. Si sono finora quindi solo raffinate le qualità relative al processo: tuttavia, risultano indispensabili per lo sviluppo seriale dei microservizi.

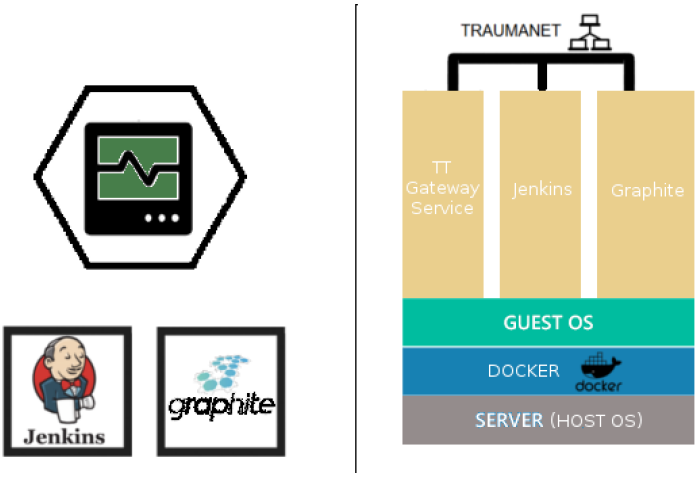

Figura 3.6: Tutti i microservizi in esecuzione devono comunicare costantemente a Graphite il livello di utilizzo delle proprie risorse

```
FROM TraumaTracker/BaseService
COPY ./ /TraumaLocationService/
WORKDIR /TraumaLocationService/
RUN tr -d '\r' < gradlew > new-gradlew
   && mv gradlew gradlew-old
   && mv new-gradlew gradlew
   && chmod +x gradlew
EXPOSE 8083:8083 TCP
RUN ./gradlew build
CMD ./gradlew run $GRADLEOPT
```

Listato 3.6: Dockerfile per la configurazione del TraumaLocationService

3.2.4 Segregazione implementativa

La procedura necessaria alla creazione di un container sul quale mettere in esecuzione l'immagine del GatewayService (paragrafo 3.2.1) può essere riproposta sia per il TraumaLocationService, che il TTService, ovviamente cambiando il riferimento delle porte esposte, ed aggiungendo eventuali altre librerie in base alle necessità. I Dockerfile relativi a questi due servizi sono visibili, rispettivamente, nei listati 3.6 e 3.7.

L'introduzione dei servizi di localizzazione geografica e di compilazione report avvicina finalmente il server sul quale è installata la piattaforma Docker alle funzionalità di base necessarie affinchè il TraumaTracker possa diventare operativo in questo nuovo constesto di virtualizzazione. La struttura, del siste-

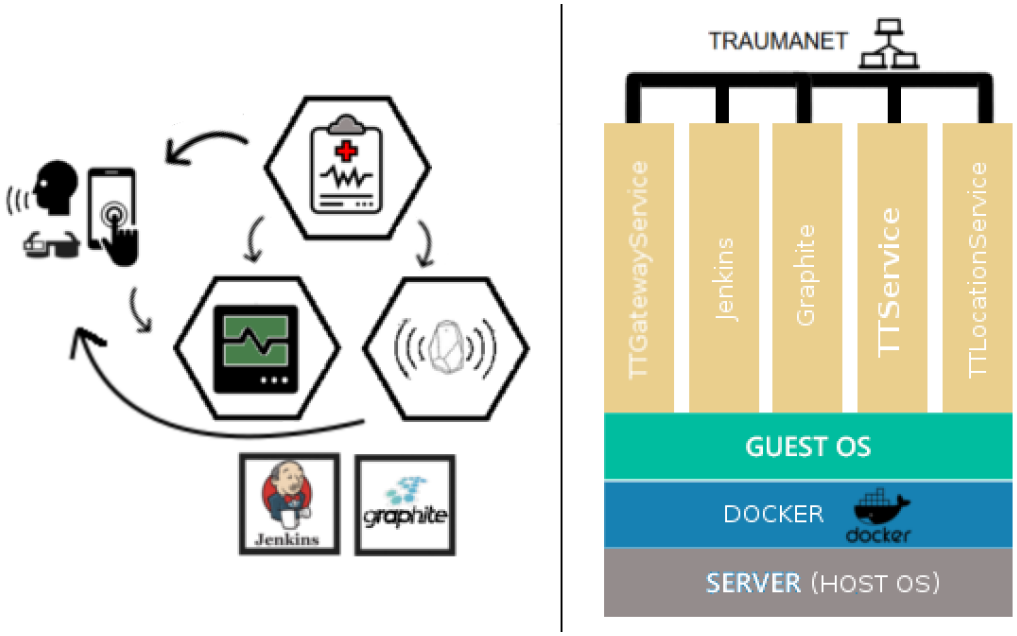

Figura 3.7: Condivisione del database fra i microservizi *TTService* e *TraumaStatService*

```
FROM TraumaTracker/BaseService
COPY ./ /TTService/
WORKDIR /TTService/
RUN tr -d '\r' < gradlew > new-gradlew
   && mv gradlew gradlew-old
   && mv new-gradlew gradlew
   && chmod +x gradlew
EXPOSE 8081:8081 TCP
RUN ./gradlew build
CMD ./gradlew run $GRADLEOPT
```

Listato 3.7: Dockerfile per la configurazione del TTService

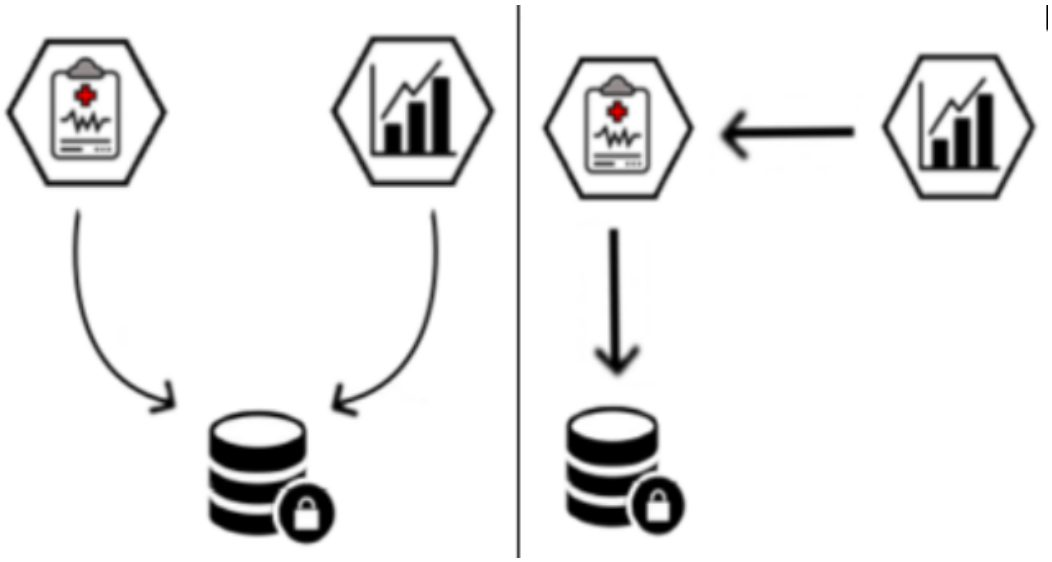

Figura 3.8: Il cambiamento concettuale da realizzare nella coesistenza fra TTService e TraumaStatService

ma e della piattaforma, è raffigurata in figura 3.7, con la direzione delle freccie che indica il flusso delle richieste. Questa è la stessa struttura logica definita al paragrafo 1.2, e rappresenta la prima concretizzazione funzionale del progetto TraumaTracker [6]. Nel periodo temporale che va dal 'lancio' del sistema in corsia ospedaliera alla stesura di questa dissertazione, è stato introdotto un altro servizio, TraumaStatService, che, sulla base delle informazioni contenute nei report generati, calcola una serie di statistiche relative al numero di pazienti sui quali sono state effettuate tecniche di resuscitazione, quale fosse la gravità della loro condizione, le modalità con cui sono arrivati in ospedale, e molte altri parametri utili al processo di *trauma documentation*.

La modalità con cui le informazioni vengono recuperate sono tuttavia problematiche: il TraumaStatService effettua degli accessi diretti al sistema di persistenza in cui sono conservati i report, di proprietà formale del TTService, rompendo quindi la segregazione fra microservizi, principio descritto al paragrafo 2.4. È improrogabile che questo venga rifattorizzato con la massima priorità in una forma nella quale l'*ownership* della banca dati sia restituita al microservizio che la detiene, permettendo l'accesso alle informazioni soltanto attraverso apposite richieste, in maniera tale che non vengano esposti dettagli implementativi, pericolosi per la resilienza dell'intero sistema.

Effettuare questo cambiamento renderebbe possibile il deployment del TraumaStatService (listato 3.8) nella nuova ottica dei microservizi, portando il sistema nello stato mostrato in figura 3.9, raggiungendo quindi la forma operativa

```
FROM TraumaTracker/BaseService
COPY ./ /TraumaStatService/
WORKDIR /TraumaStatService/
RUN tr -d '\r' < gradlew > new-gradlew
   && mv gradlew gradlew-old
   && mv new-gradlew gradlew
   && chmod +x gradlew
EXPOSE 8085:8085 TCP
RUN ./gradlew build
CMD ./gradlew run $GRADLEOPT
```

Listato 3.8: Dockerfile per la configurazione del TraumaStatService

presente attualmente in ospedale.

3.2.5 Resilienza agli errori

Il pattern *Circuit Breaker* ed il pattern *Bulkhead*, individuati e descritti al paragrafo 2.7, garantiscono la resilienza del sistema rispetto alla latenza, ma coprono uno spettro di possibilità in cui il malfunzionamento è esclusivamente legato alla rete. Nonostante svolgano un lavoro eccellente nel prevenire che il corretto funzionamento del software distribuito venga alterato dall'indeterminabilità del comportamento della rete, questa non è l'unica causa di natura tecnica che può impedire il funzionamento del sistema. Il fallimento degli host sui quali sono in esecuzione i microservizi va tenuto in considerazione in maniera altrettanto importante, poichè è un fenomeno che può accadere in qualsiasi momento, con la stessa impredicibilità di un fallimento di rete. La replicazione dei servizi su macchine diverse, insieme all'aggiunta di server capaci di effettuare *service discovery* in maniera tale da indirizzare le richieste alle macchina attive, sembra essere una soluzione definitiva. Tuttavia, la possibilità che tutte le macchine sulle quali le repliche di un particolare servizio sono in esecuzione falliscano è ancora presente, e altrettanto pericolosa per la continuità di un sistema orientato al supporto medicale. Inoltre, la rimessa in funzione di questa moltitudine di macchine prevede comunque una qualche forma di intervento manuale, l'esatto opposto dei precetti finora discussi.

Una parte della soluzione consiste nell'affidarsi ad un servizio di *Infrastructure-as-a-service*, completamente o affiancandolo in parte al proprio cluster di macchine. Questi sistemi garantiscono un *uptime* tendenzialmente elevato, oltre che alla rilocalizzazione automatica di una macchina guasta.

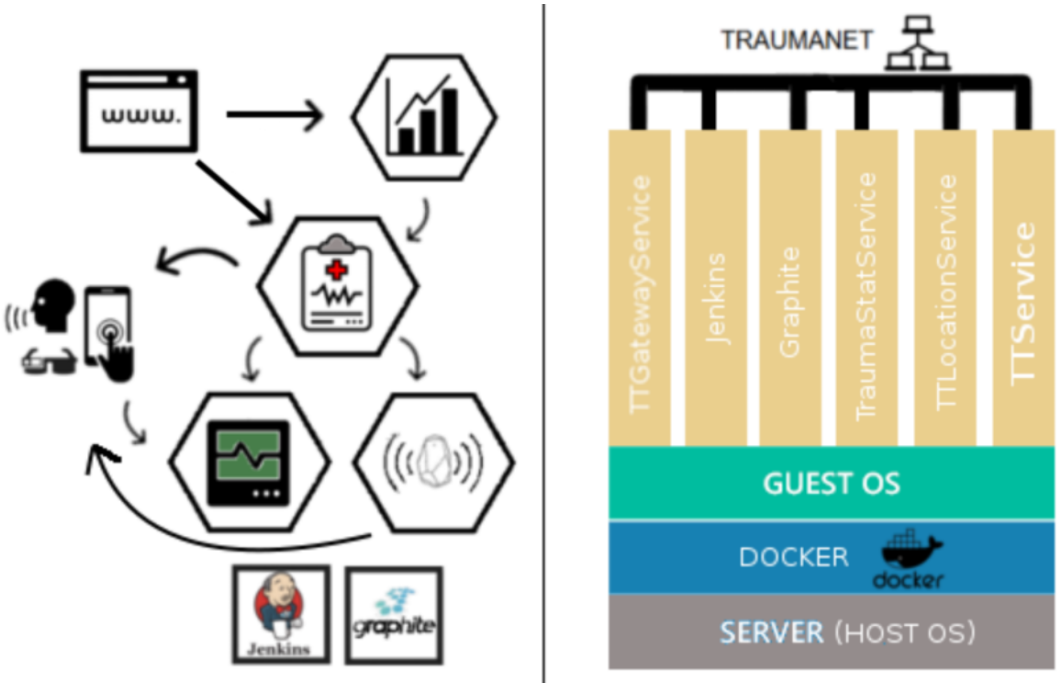

Figura 3.9: Struttura completa del sistema TraumaTracker

Un sistema di orchestrazione container automatizzerebbe il processo di rimessa in attività dei microservizi sulle macchine che hanno subito un arresto delle proprie funzionalità. Il tool individuato per adempiere a questo compito è Kubernetes, un progetto Google inizialmente realizzato per soddisfare le necessità interne della multinazionale, e dal 2014 reso open-source.

La struttura di Kubernetes consiste in un modulo chiamato 'Kubernetes cluster service' (KCS) ed un moltitudine di 'Worker', ognuno dei quali è un host equipaggiato di una piattaforma di gestione container. I Worker espongono delle API che consentono al KCS di gestire quali e quante repliche delle immagini tenere in esecuzione, sulla base di una configurazione decisa tramite uno script dichiarativo, scritto in linguaggio YAML. Inoltre, il KCS è incaricato di asserire come la situazione dei Worker rispecchi quella della configurazione: nel momento in cui un host venisse trovato fallimentare, il KCS istanzierebbe le repliche dei microservizi venute a mancare sui Worker rimanenti, ripristinando la *quality of service* necessaria. Nel momento in cui il Worker fallimentare torni in comunicazione con il KCS, verrebbe di nuovo considerato fra gli host disponibilie per farsi carico di repliche di microservizi.

Questo meccanismo diventa essenziale in un contesto ospedaliero, in quanto Kubernetes potrebbe mantenere un'intera copia dell'infrastruttura del TraumaTracker, e garantire la continuità del servizio offerto dai microservizi nono-

stante l'avvenire di qualsiasi evenienza.

Conclusioni

Lo studio dell'approccio architetturale orientato ai microservizi, e dei risultati ottenuti dalle organizzazioni che hanno adottato tale pratica, dimostra come questo stile riesca ad impedire un calo nella velocità di sviluppo di nuove funzionalità nel momento in cui aumenta la complessità del dominio applicativo, specialmente in visione di problematiche non ancora note.

Non essere vincolati ad un unico stack tecnologico o linguaggio di programmazione permette di affrontare ogni singolo nuovo problema usando le astrazioni esistenti più 'naturali', e questo permette agli sviluppatori di dedicare le proprie energie allo studio e ad una resa sempre più raffinata del Bounded Context in questione, piuttosto che a come rendere operative tali funzionalità. Questo diventa essenziale nel momento in cui ci si trova davanti a problematiche finora non espresse, o ad evoluzioni nel dominio talmente dirompenti da rendere complicato il versionamento dei microservizi finora sviluppati: seguire lo stack tecnologico non è richiesto nemmeno fra versioni diverse dello stesso servizio. La velocità di risposta nei confronti dell'evoluzione del sistema è quindi massimizzata, ed è possibile accettare qualsiasi cambiamento in serenità.

Il gap tecnologico e procedurale affrontato nella teorizzazione dello *shift* ai microservizi si è rivelato essere facilmente valicabile grazie agli strumenti di virtualizzazione sviluppati negli ultimi anni: effettuare il *provisioning* automatico di molti sistemi dalle caratteristiche diverse risulta estremamente facilitato dalla piattaforma Docker e dalle infrastrutture che sono nate a supporto della tecnologia *container*.

I refactoring interni necessari a trasformare gli attuali 'piccoli servizi dai confini ben delineati' in effettivi 'microservizi' sono pochi, ben mirati, e dovuti a problematiche così ben studiate durante l'intero corso di vita di questa branca della *service orientation* da diventare quasi banali una volta delineato il perimetro all'interno del quale lavorare.

I cambiamenti principali nell'infrastruttura riguardano l'aggiunta di strumenti che permettano la garanzia di utilizzare le migliori tecniche di automazione nell'intero processo produttivo, il tutto orientato verso la miglior qualità possibile del software prodotto.

Considerando il contesto accademico in cui il progetto TraumaTracker nasce, la cultura DevOps che viene abbracciata dai team di sviluppo di microservizi si pone come un eccellente collante delle conoscenze acquisite durante l'intero corso degli studi, permettendo al principio della decentralizzazione pervasiva di giocare un ruolo didattico estremamente formativo. L'approccio Continuous consegnerebbe agli sviluppatori/studenti le nozioni che regolano l'intero mondo informatico al di fuori dell'accademia, formando *craftwomen* e *craftmen* esperti nel rigore delle procedure di sviluppo.

L'unione di questi fattori rende l'adozione completa dell'approccio ai microservizi uno step naturale nel processo evolutivo del sistema TraumaTracker, le cui funzionalità non possono che incrementare in numero e in qualità.

Gli sviluppi futuri relativi agli argomenti trattati in questa tesi prevedono un'ampia analisi delle piattaforme e delle tecniche necessarie ad effettuare l'orchestrazione dei container sui quali verrà effettuato il *deploying* dei microservizi. Le possibilità messe a disposizione da un tool di *orchestration* e *provisioning* avanzato come Kubernetes rappresenterebbe il principale attore nella resilienza del sistema, ma perchè sia effettivamente studiabile è necessario possedere concretamente un pool di macchine fisiche, o l'accesso ad un sistema di *Infrastructure-as-a-Service*, la cui gestione venga consegnata all'orchestratore. Per quanto il funzionamento concettuale sia stato studiato ed espresso nel paragrafo dedicato, l'attuazione delle relative pratiche non è mai stata applicata al progetto, e sarà obiettivo di analisi una volta superata la fase iniziale di migrazione ai microservizi.

Ringraziamenti

Desidero ringraziare il professore Alessandro Ricci e l'ingegnere Angelo Croatti per avermi coinvolto in un progetto interessante e complesso come il TraumaTracker, e per averlo realizzato in primo luogo.
Un grazie alla mia famiglia, che mi ha permesso di seguire la strada che più desideravo, e mi ha supportato e consigliato nelle mie decisioni.
Un grazie a tutti i miei coinquilini, passati e presenti, per aver sopportato l'esplosività del mio caratteraccio.
Un grazie ai miei amici lontani, che sopportano le mie assenze eremitiche e ogni volta mi abbracciano come se non ci vedessimo da un paio di giorni.

Bibliografia

[1] Eric Evans, *Domain-Driven Design: Tackling complexity in the heart of software*, Addison-Wesley, 2004, 364 - 370.

[2] Eric Evans, *Domain-Driven Design: Tackling complexity in the heart of software*, Addison-Wesley, 2004, 335 - 340.

[3] Eric Evans, *Domain-Driven Design: Tackling complexity in the heart of software*, Addison-Wesley, 2004, 193 - 202.

[4] Mehta, N. J., and Khan, I. A. (2002). *Cardiology's 10 Greatest Discoveries of the 20th Century.* Texas Heart Institute Journal, 29(3), 164–171.

[5] Bernadette G. van den Hoogen, Jan C. de Jong, Jan Groen, Thijs Kuiken, Ronald de Groot, Ron A.M. Fouchier, Albert D.M.E. Osterhaus *A newly discovered human pneumovirus isolated from young children with respiratory tract disease*, Nature Medicine, Nature Publishing Group.

[6] Sara Montagna, Angelo Croatti, Alessandro Ricci, Vanni Agnolotti, Vittorio Albarello, Emiliano Gamberini, *Real-Time Tracking and Documentation in Trauma Management*, Health Informatics Journal, 2018

[7] Zikos D, Diomidous M and Mpletsa V. *The effect of an electronic documentation system on the trauma patient's length of stay in an emergency department.* Journal of Emergency Nursing 2014; 40(5): 469 – 475.

[8] James Lewis, Martin Fowler, *Microservices: a definition of this new architectural term*, https://martinfowler.com/articles/microservices.html, 2014

[9] Charlene O'Hanlon, *A conversation with Werner Vogels*, Queue Volume 4 Number 4, ACM, May 2006

[10] Sam Newman, *Building microservices: designing fine-grained systems*, O'Reilly, 2015, 2 - 2

[11] Sam Newman, *Building microservices: designing fine-grained systems*, O'Reilly, 2015

[12] Alexandra Noonan, *Goodbye Microservices: from 100s of problem children to 1 superstar*, https://segment.com/blog/goodbye-microservices/, 2018

[13] Robert C. Martin *Clean Code: A handbook of agile software craftmanship*, Prentice Hall, 2017, 61 - 67

[14] Robert C. Martin *Clean Code: A handbook of agile software craftmanship*, Prentice Hall, 2017, 87 - 91

[15] Arnon Rotem-Gal-Oz, *Nanoservices*, https://arnon.me/wp-content/uploads/2010/10/Nanoservices.pdf, 2010

[16] Ben Morris *How big is a microservice?* http://www.ben-morris.com/how-big-is-a-microservice/, 2015

[17] Matthew Clark, *Powering BBC Online with nanoservices*, https://medium.com/bbc-design-engineering/powering-bbc-online-with-nanoservices-727840ba015b, 2017

[18] Martin Fowler, *Database Thaw*, https://martinfowler.com/bliki/DatabaseThaw.html, 2008

[19] Martin Fowler, *TolerantReader*, https://martinfowler.com/bliki/TolerantReader.html, 2011

[20] Michael T. Nygard, *Release It! Design and deploy Production-Ready software*, 2007

[21] Beth Scurrie, *Enter the Pact Matrix* http://rea.tech/enter-the-pact-matrix-or-how-to-decouple-the-release-cycles-of-your-microservices/, 2014

[22] Thoughtworks, *Pacto*, https://github.com/thoughtworks/pacto, 2014

[23] Sam Newman, *Building microservices: designing fine-grained systems*, O'Reilly, 2015, 161 - 161

[24] Martin Fowler, *MicroservicePremium*, https://martinfowler.com/bliki/MicroservicePremium 2015

[25] Sam Newman, *Principles of microservices*, https://www.youtube.com/watch?v=PFQnNFe27kU, Devoxx Belgium, 2015

[26] Steffen Jacobsm *SpringMVC vs. Vert.x*, https://blog.oio.de/2016/07/29/springmvc-vs-vert-x/, blog.oio.de, 2016

[27] Sam Newman *Deploying And Scaling Microservices* https://www.youtube.com/watch?v=sO0RKuVhfw4, GOTO;conference, 2016