ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

TRANSIZIONE AD ARCHITETTURE SOFTWARE A MICROSERVIZI E AD EVENTI PER SISTEMI INFORMATIVI SANITARI: ANALISI DEL CASO AUSL DELLA ROMAGNA E PROTOTIPAZIONE DELLA CARTELLA CLINICA ELETTRONICA

Elaborato in
SOFTWARE ARCHITECTURE AND PLATFORMS

Relatore
Prof. ALESSANDRO RICCI

Presentata da
MARCO FONTANA

Corelatore
Ing. SAMUELE BURATTINI
Ing. ANGELO CROATTI

Anno Accademico 2024 – 2025

A mia mamma

Abstract

Con questo elaborato si presenta uno studio sulle architetture software per sistemi informativi sanitari complessi, prendendo in considerazione il caso di studio di AUSL della Romagna, effettuando un'analisi generale sulle criticità e possibilità di miglioramento considerando transizioni ad architetture moderne a partire da microservizi e ad eventi. In particolare, si è identificato nel caso specifico della cartella clinica elettronica, la possibilità di re-ingegnerizzazione ad un'architettura distribuita in linea con i requisiti non funzionali moderni, per cui si è proposto un prototipo di progetto che fornisca le linee guida per l'ideazione di una soluzione architetturale in grado di superare le limitazioni attuali.

La ricerca si articola su tre contributi principali. Il primo riguarda un'analisi comparativa degli standard sanitari HL7 FHIR e OpenEHR, evidenziando come FHIR presenti vantaggi significativi in termini di semplicità implementativa e performance, pur mantenendo la possibilità di utilizzo complementare di entrambi gli standard. Il secondo contributo propone una transizione verso un'architettura distribuita caratterizzata da un partizionamento funzionale dei diversi contesti clinici, garantendo maggiore autonomia, disaccoppiamento e robustezza, in linea con i requisiti non funzionali richiesti. Il terzo contributo presenta una soluzione per l'identificazione di eventuali criticità e validazione delle architetture, mediante il tracciamento di metriche e creazione di scenari per la misurazione concreta delle proprietà garantite.

La metodologia adottata combina analisi teorica, progettazione architetturale e mostra come effettuare una validazione sperimentale attraverso la definizione di scenari per la misurazione quantitativa degli attributi di qualità garantiti. Questo avviene mediante l'implementazione di un prototipo funzionale, a titolo esemplificativo, che prende in considerazione uno scenario rappresentativo della cartella clinica elettronica, per chiarire alcuni aspetti chiave discussi in questo elaborato e fornire uno strumento da assumere come modello generale per una futura progettazione del sistema.

Indice

Elenco delle figure

Elenco dei codici

Introduzione

Con questo elaborato si presenta uno studio sulle architetture software per sistemi informativi sanitari complessi, prendendo in considerazione il caso di studio di AUSL della Romagna, effettuando un'analisi generale sulle criticità e possibilità di miglioramento considerando transizioni ad architetture moderne a partire da microservizi e ad eventi. In particolare, si è identificato nel caso specifico della cartella clinica elettronica, la possibilità di re-ingegnerizzazione ad un'architettura distribuita in linea con i requisiti non funzionali moderni, per cui si è proposto un prototipo di progetto che fornisca le linee guida per l'ideazione di una soluzione architetturale in grado di superare le limitazioni attuali.

I sistemi informativi sanitari rappresentano oggi una delle sfide architetturali più complesse nel panorama informatico contemporaneo. La necessità di gestire enormi volumi di dati critici, garantire l'interoperabilità tra sistemi eterogenei e assicurare performance elevate, richiede soluzioni architetturali innovative e robuste. Questo elaborato di tesi affronta uno studio riguardo le principali architetture software moderne, gli standard principalmente utilizzati e lo stato dell'arte delle architetture software in ambito sanitario, per poi proseguire con una analisi del caso specifico di AUSL della Romagna, mostrando criticità e possibilità di miglioramenti ed in infine la progettazione di un'architettura distribuita per il servizio di cartella clinica elettronica e sistemi correlati. L'analisi parte da una situazione attuale caratterizzata da un'architettura monolitica modulare che, pur funzionale, presenta limitazioni significative in termini di robustezza, performance e disaccoppiamento, il prototipo sviluppato fornisce le linee guida per l'ideazione di una soluzione architetturale in grado di superare le limitazioni attuali.

La ricerca si articola su tre direttrici principali, ognuna delle quali contribuisce a fornire linee guida per la definizione di una soluzione architetturale complessiva più resiliente ed efficace:

Il primo contributo riguarda l'analisi comparativa degli standard sanitari: attraverso un'analisi approfondita degli standard HL7 FHIR e OpenEHR, l'elaborato mette in evidenza le caratteristiche distintive di ciascun approccio. FHIR si concentra sullo scambio di dati sanitari con una complessità ridotta e

prestazioni generalmente superiori, utilizzando payload più leggeri e offrendo maggiore flessibilità nell'integrazione con sistemi legacy. OpenEHR, d'altra parte, eccelle nella modellazione strutturata e persistenza dei dati attraverso il suo sistema di archetipi e template. L'analisi conclude che, pur essendo entrambi gli standard efficaci, FHIR presenta vantaggi significativi in termini di semplicità implementativa e proprietà non funzionali, mostrando anche la possibilità di utilizzare entrambi gli standard in maniera complementare.

Il secondo contributo riguarda lo studio delle architetture sanitarie allo stato dell'arte ed un'analisi dell'architettura attualmente utilizzata in AUSL della Romagna, proponendo una transizione architetturale da monolitica a distribuita, in grado di superare le maggiori criticità attualmente rilevate e garantire le proprietà richieste: la tesi propone le linee guida per l'ideazione di un'architettura per la cartella clinica elettronica e sistemi correlati in linea con i requisiti non funzionali forniti. Il partizionamento proposto separa i diversi contesti clinici in maniera funzionale, sfruttando in parte la suddivisione già presente, ma ciascuno dei servizi utilizza il proprio modello di dati specifico e requisiti non funzionali dedicati, in modo da essere il più possibile indipendente rispetto agli altri.

Il terzo contributo riguarda la progettazione ed implementazione di pattern per la raccolta di metriche, per permettere la misurazione concreta dei requisiti non funzionali attualmente garantiti e la validazione di proposte architetturali future, in modo da poter identificare e quantificare dettagliatamente la presenza di criticità e la loro misura. Questi pattern possono essere inoltre utilizzati per effettuare comparazioni tra l'architettura attuale e quelle proposte, in modo da valutare l'effettiva efficacia delle soluzioni fornite in termini di proprietà non funzionali.

La ricerca adotta un approccio metodologico che combina analisi teorica, progettazione architetturale e propone degli scenari per la validazione sperimentale. Partendo dallo studio dello stato dell'arte delle architetture software (monolitiche, service oriented architecture, microservizi ed event-driven), viene condotta un'analisi del sistema attuale di AUSL della Romagna per identificare criticità e opportunità di miglioramento.

Le linee guida per una futura implementazione della cartella clinica elettronica in linea con i requisiti forniti, sono rafforzate dalla progettazione a livello architetturale di un prototipo funzionale e una sua implementazione, che mostra uno scenario rappresentativo sufficientemente generale da essere esemplificativo per la progettazione della cartella clinica elettronica e una validazione architetturale della stessa, mostrando come creare scenari per la validazione e misurazione degli attributi di qualità garantiti.

Nei capitoli successivi, questa introduzione si svilupperà attraverso un'analisi dettagliata delle tecnologie architetturali, al Capitolo 1, degli standard

sanitari, al Capitolo 2, dello stato dell'arte dei sistemi informativi sanitari, al sezione 3.1, dell'analisi del sistema attuale di AUSL della Romagna, criticità e opportunità di miglioramento, al Capitolo 3, fino alla presentazione di un prototipo, che considera uno scenario rappresentativo generale, a titolo esemplificativo, per una futura re-ingegnerizzazione della cartella clinica elettronica e validazione sperimentale degli attributi di qualità mediante scenari, al Capitolo 4.

Capitolo 1

Elementi di Architetture Software

In questa sezione si introducono alcuni concetti fondamentali che riguardano le architetture software, iniziando con una panoramica generale riguardo elementi necessari a comprenderne lo scopo e il funzionamento, seguita da un approfondimento riguardo le diverse tipologie di architetture che si sono susseguite negli anni, a partire da architetture tipo monolitico, fino ad architetture distribuite basate sui microservizi e ad eventi, in modo da introdurre e fornire la base per una migliore comprensione degli argomenti che verranno trattati nei successivi capitoli, limitando la trattazione alle sole tecnologie che verranno trattate nel caso di studio di AUSL della Romagna, ambito di analisi di questa tesi.

1.1 Architetture software monolitiche

Le architetture monolitiche rappresentano il paradigma tradizionale nello sviluppo di applicazioni software, dove tutte le funzionalità e i servizi sono integrati e operano come una singola unità coesiva. In questo modello architetturale, tutti i componenti dell'applicazione sono strettamente accoppiati all'interno di una base di codice unificata, condividendo risorse comuni come database, memoria e processi di esecuzione.

Le caratteristiche distintive delle architetture monolitiche includono la centralizzazione della logica applicativa in un singolo artefatto di cui fare il deploy, la condivisione diretta di risorse tra componenti senza sovraccarico di rete, e la gestione unificata del ciclo di vita dell'applicazione. Tutti i processi che compongono il sistema sono eseguiti all'interno dello stesso spazio di indirizzamento, permettendo comunicazioni dirette in memoria tra i diversi moduli funzionali.

Questo approccio architetturale offre vantaggi significativi in termini di semplicità di sviluppo e deployment, chiarezza dei processi di business attraverso un flusso end-to-end facilmente tracciabile, e ridotta complessità operativa dovuta all'assenza di comunicazioni di rete tra componenti. La validazione locale risulta semplificata, in quanto è possibile eseguire e testare l'intero ambiente applicativo su una singola macchina, facilitando il processo di sviluppo e debug.

Tuttavia, le architetture monolitiche presentano limitazioni significative che emergono con la crescita dimensionale del sistema e dei team di sviluppo. La scalabilità è vincolata alla necessità di dimensionare l'intera applicazione anche quando solo specifici moduli richiedono risorse aggiuntive, comportando inefficienze nell'utilizzo delle risorse. L'evoluzione tecnologica risulta complessa e la manutenibilità decresce progressivamente con l'aumento della complessità del sistema [76].

Il forte accoppiamento tra componenti introduce rischi significativi per la disponibilità del sistema, dove il malfunzionamento di un singolo modulo può compromettere l'intero ecosistema applicativo [19]. Inoltre, la gestione di team di sviluppo numerosi su una singola base di codice genera conflitti di integrazione e richiede coordinamento estensivo tra i diversi gruppi di lavoro, limitando la capacità di sviluppo parallelo e indipendente delle funzionalità [72].

1.1.1 Architetture monolitiche modulari

Tra le possibili architetture di tipo monolitico, si riporta una soluzione chiamata 'modulare' in quanto presenta le classiche caratteristiche di una architettura monolitica ma mantiene una separazione modulare tra i vari componenti, garantendo una maggiore flessibilità.

Il modello monolitico modulare è costituito da diversi componenti modulari con un singolo database, come un'unità quantica unica (vd. Figura 1.1). Questa architettura rappresenta un compromesso tra la semplicità del monolite tradizionale e la necessità di organizzazione modulare che faciliti la manutenibilità e l'evoluzione del sistema.

La caratteristica distintiva di questo approccio risiede nella separazione logica dei domini all'interno della stessa unità di deployment. Ogni dominio applicativo viene implementato come componente separato, mantenendo però l'esecuzione all'interno dello stesso processo e la condivisione di un database comune. Questa strutturazione consente di preservare i vantaggi operativi del monolite, eliminando la complessità delle comunicazioni di rete, mentre introduce un'organizzazione che riflette la struttura del business.

La separazione accurata tra domini e design dei dati risulta particolarmente importante in questa architettura, poiché facilita potenziali migrazioni future

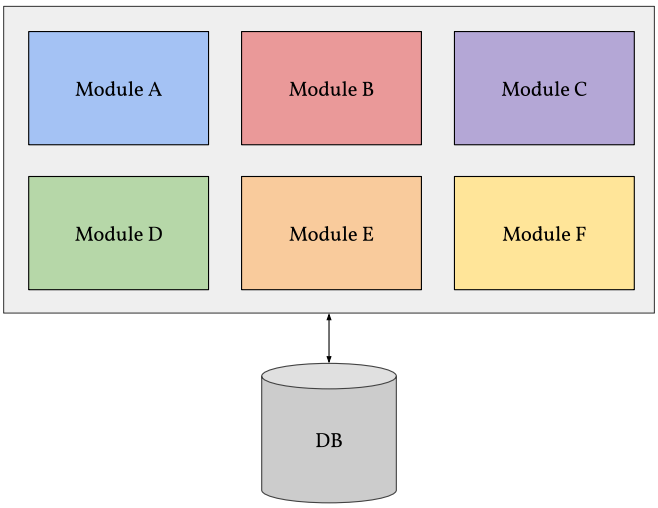

Figura 1.1: Architettura monolitica modulare

verso architetture distribuite. Mantenendo tabelle e asset di database logicamente separati per ciascun dominio, l'architettura prepara il terreno per una possibile evoluzione verso microservizi senza richiedere una ristrutturazione completa del sistema [72].

1.2 Architetture software distribuite

Le architetture distribuite rappresentano un paradigma fondamentale nell'ingegneria del software moderno, dove i componenti del sistema sono distribuiti attraverso più calcolatori collegati in rete, comunicando e coordinandosi mediante il passaggio di messaggi od eventi. Questo approccio si contrappone alle architetture monolitiche tradizionali, offrendo vantaggi significativi in termini di scalabilità, resilienza e flessibilità tecnologica.

Le caratteristiche distintive delle architetture distribuite includono la decomposizione funzionale, dove il sistema viene suddiviso in componenti distinti che operano in modo autonomo, e la distribuzione fisica, con componenti che risiedono su nodi diversi della rete. Questo paradigma introduce tuttavia anche nuove complessità, come la gestione della latenza e partizionamenti della rete, la consistenza dei dati distribuiti, la disponibilità dei servizi, e la necessità di meccanismi robusti per la gestione dei fallimenti del sistema.

Le architetture distribuite si sono evolute attraverso diversi paradigmi, dalle Service Oriented Architectures (SOA) con i loro Enterprise Service Bus cen-

tralizzati, alle architetture a microservizi che privilegiano il disaccoppiamento e la granularità fine, fino alle architetture event-driven che sfruttano eventi asincroni per la coordinazione tra componenti. Ciascun approccio presenta specifici vantaggi e compromessi che devono essere valutati in base ai requisiti del dominio applicativo e ai vincoli operativi dell'organizzazione.

La scelta di un'architettura distribuita comporta inevitabilmente l'accettazione di maggiore complessità operativa, come il deployment distribuito, in cambio di benefici in termini di scalabilità, flessibilità tecnologica, e capacità di evoluzione del sistema.

1.2.1 Architetture Orientate ai Servizi

Le architetture orientate ai servizi (in inglese, Service Oriented Architectures, 'SOA'), rappresentano un modello in cui la logica del sistema viene decomposta in diverse unità distinte, dette 'servizi'.

Un servizio rappresenta una componente software che è accessibile all'interno della rete in cui si trova. Collettivamente, tutti i servizi che fanno parte della stessa rete, compongono l'intera logica del sistema, utilizzando un approccio distribuito sia dal punto di vista logico che spaziale.

Le caratteristiche chiave di queste architetture, è che permettono di ottenere:

- **Indipendenza e autonomia tra i servizi**: ciascuno di essi definisce un proprio confine tecnico di cui ha pieno il controllo e permette alta riusabilità, confinando in un unico servizio la logica che lo caratterizza.
- **Eterogeneità**: di linguaggi e tecnologie utilizzate.
- **Interoperabilità**: sfruttando standard e convenzioni condivise tra i vari servizi mediante contratti, astraendo dalla logica sottostante.

Ogni servizio permette di essere trovato da altri servizi, ed espone un'interfaccia che abilita la comunicazioni tra essi.

Topologia Guidata dall'Orchestratore

Questa topologia è caratterizzata dalla decomposizione tecnica in diversi in servizi, per supportare la riutilizzabilità.

Sistemi che utilizzano architetture di tipo SOA sono composti da diversi servizi distribuiti, che rendono il sistema complesso, per cui è necessario definire un orchestratore che permetta di gestire la complessità del sistema.

I servizi sono separati in diversi 'layer' che identificano diversi livelli in cui un servizio si può trovare:

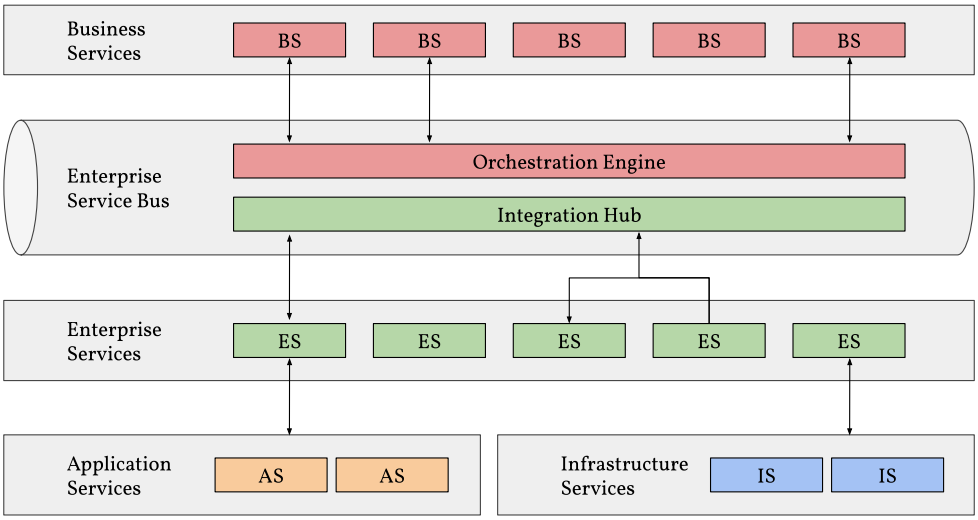

Figura 1.2: Topologia orientata all'orchestratore per architetture orientate ai servizi

- **Business services**: sono servizi posizionati al livello più alto, definiscono il punto di ingresso alla logica del sistema, sono definiti dall'utente e non contengono implementazioni ma solo interfacce per definire input e output, che vengono implementate dai servizi di tipo 'enterprise services'.
- **Enterprise services**: contengono le implementazioni specifiche delle interfacce dei 'business services', permettendo di sviluppare soluzioni che possono essere riutilizzate attraverso le medesime definizioni. Questo approccio permette di generare una collezione di asset riutilizzabili nella forma di enterprise services ma data la dinamicità del mondo reale, nonostante la possibilità di riutilizzare implementazioni già esistenti, rimane la problematica di dover adattare o estendere queste soluzioni in base a nuovi requisiti di dominio e tecnologie.
- **Application services e infrastructure services**: riguardano servizi che espongono operazioni di supporto ad altri servizi (come il monitoraggio, autenticazione e autorizzazione), e servizi che non richiedono di essere riutilizzati, per cui tendono ad essere implementazioni puramente concrete.

Business services ed enterprise services sono collegati tra loro mediante 'enterprise service bus' (ESB) che svolge la funzione di orchestratore e permette

l'interoperabilità tra i servizi, fornendo un bus omogeneo pensato per connettere tra loro servizi mediante una API [42], questo è caratterizzato dalle funzioni di:

- **Orchestration engine**: mette a disposizione funzionalità tipiche di un orchestratore come la coordinazione per le transazioni, gli sviluppatori devono solamente collegarsi al bus e specificare la destinazione, l'ESB si occupa autonomamente di rilevare e spedire le comunicazioni al endpoint corrispondente mediante i canali [9].

- **Integration hub**: funge da centro integrativo per permettere la comunicazione tra servizi eterogenei. Il suo scopo è quello di facilitare l'interoperabilità tra servizi diversi che altrimenti potrebbero non riuscire a comunicare tra loro a causa, ad esempio, della eterogeneità delle strutture, formati o protocolli con cui sono rappresentate le informazioni. L'ESB si occupa di mappare e trasformare questi dati in modo che ogni servizio possa interpretarli correttamente. Ciascun servizio comunica con gli altri mediante dei canali che sono qui definiti (comunicazione che può essere punto-punto o punto-multipunto, quindi tra coppie di servizi o broadcast) e la logica di traduzione e interpretazione dei dati è definita per ciascun canale.

In questa architettura, tutte le richieste attraversano l'Enterprise Service Bus, che agisce come intermediario per tutte le chiamate che vengono effettuate all'interno dell'architettura: un servizio inoltra una richiesta per messaggio verso il servizio target mediante la API esposta dall'ESB, questa viene raccolta dall'orchestratore, viene effettuata una traduzione attraverso l'integration hub che effettua le operazioni di traduzione e routing, e il messaggio viene poi inoltrato verso il servizio enterprise obiettivo [72]; in questo modo si permette la comunicazione tra i processi rappresentanti le diverse applicazioni che compongono il sistema, che possono essere invocate a seguito di eventi o consumer esterni. Non esistono standard ufficiali che definiscono esattamente quali servizi debbano essere implementati all'interno di un ESB, ma tipicamente vengono forniti servizi riguardo il trasporto, eventi e mediazione tra servizi, in modo da agevolare l'integrazione in sistemi complessi eterogenei che comunicano con esso mediante adattatori.

1.2.2 Architetture a Microservizi

Le architetture a microservizi rappresentano un approccio architetturale che struttura un'applicazione come una collezione di servizi distribuiti, caratterizzati da granularità fine, debolmente accoppiati e sviluppabili in modo il

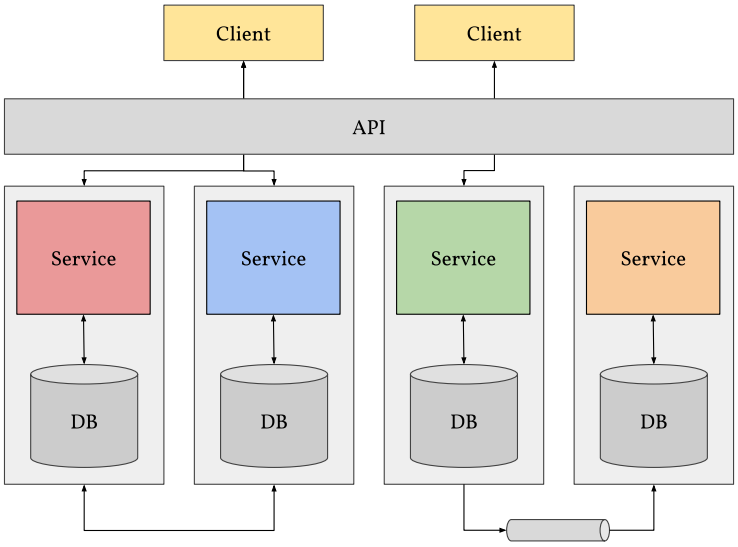

Figura 1.3: Architettura a microservizi

più possibile indipendente dagli altri. Questo paradigma architetturale è emerso come evoluzione delle architetture orientate ai servizi (SOA), introducendo principi più rigorosi di modularità e autonomia dei servizi (vd. Figura 1.3).

Tra le caratteristiche fondamentali si riporta principalmente la granularità fine con cui è partizionato il sistema, in modo che ciascun servizio svolga un'unica funzionalità in maniera autonoma rispetto agli altri, garantendo una maggiore flessibilità nella progettazione e nell'evoluzione del sistema, consentendo modifiche localizzate senza impatti significativi su altri componenti del sistema. Ogni microservizio mantiene la propria base di dati, quindi mantiene la propria persistenza senza condivisione diretta con altri servizi e gestisce autonomamente il proprio ciclo di vita, dallo sviluppo al deployment. Questa autonomia si estende alla scelta delle tecnologie, dei linguaggi di programmazione e degli stack tecnologici più appropriati per il dominio specifico del servizio.

La comunicazione avviene esclusivamente attraverso interfacce di rete ben definite, tipicamente utilizzando protocolli come HTTP/REST, messaggistica asincrona o gRPC. Questa caratteristica elimina la condivisione diretta di risorse in memoria o database, rafforzando l'isolamento e l'indipendenza dei servizi [72] [68].

Le architetture a microservizi traggono ispirazione dai principi del Domain-Driven Design (DDD) [15], in particolare dal concetto di bounded context. Ciascun microservizio dovrebbe idealmente rappresentare un bounded context

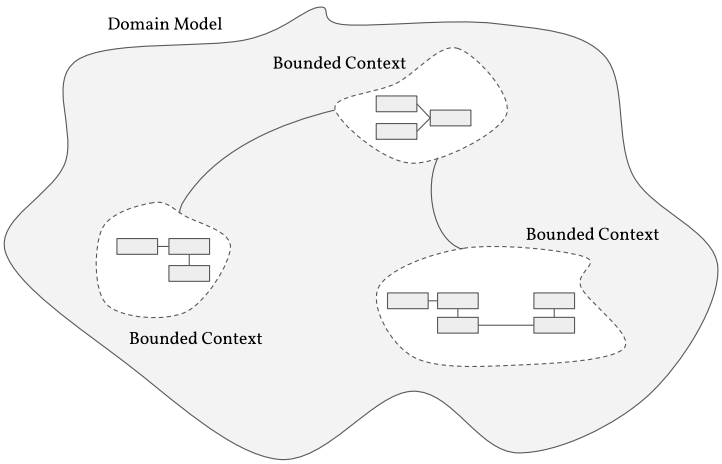

Figura 1.4: Bounded Contexts

distinto, con un modello di dominio coerente e ben definito (vd. Figura 1.4). Questo approccio facilita:

- **Coesione interna**: elementi strettamente correlati dal punto di vista funzionale rimangono all'interno dello stesso servizio
- **Accoppiamento ridotto**: minimizzazione delle dipendenze tra diversi domini di business
- **Evoluzione indipendente**: possibilità di effettuare modifiche riguardo il proprio contesto senza impatti esterni
- **Linguaggio ubiquo**: mantenimento di un vocabolario coerente all'interno di ciascun contesto

L'integrazione tra DDD e architetture a microservizi offre un framework robusto per la progettazione di sistemi distribuiti. Il DDD fornisce le basi metodologiche per identificare i confini appropriati dei servizi, mentre le architetture a microservizi offrono l'infrastruttura tecnica per implementare questi confini in modo efficace.

La chiave del successo risiede nell'applicazione rigorosa dei principi DDD nella fase di design, seguita da un'implementazione pattern all'interno di ogni microservizio. Questo approccio garantisce sistemi che non solo sono tecnicamente robusti, ma che riflettono accuratamente la complessità e l'evoluzione del dominio che supportano.

1.2.3 Architetture ad Eventi

Le architetture ad eventi (Event-Driven Architecture, EDA) rappresentano un paradigma architetturale che organizza l'applicazione attorno alla produzione, rilevazione e consumo di eventi. Questo approccio si basa sul concetto che le modifiche di stato nel sistema vengono rappresentate come eventi che vengono propagati attraverso il sistema per notificare componenti interessati, permettendo una reazione dinamica e asincrona ai cambiamenti.

Un evento rappresenta un cambiamento significativo nello stato del sistema o un'azione che si è verificata in un determinato momento. Gli eventi sono immutabili e contengono informazioni sufficienti per permettere ai consumatori di comprendere cosa è accaduto e reagire di conseguenza.

La comunicazione publish-subscribe tipicamente adottata da architetture ad eventi è una caratteristica fondamentale che distingue queste architetture dai modelli tradizionali basati su richiesta-risposta, permettendo maggiore disaccoppiamento e abilitando la reazione ai cambiamenti notificati dagli altri servizi, dove gli eventi sono generati da un produttore, che pubblica un aggiornamento ogni volta che si verifica un cambiamento di stato all'interno del proprio dominio, che viene raccolto ed inoltrato da un intermediario verso i consumatori, componenti che ricevono ed elaborano solamente gli aggiornamenti a cui sono interessati [72].

1.3 Differenze tra le architetture descritte

Tutte le architetture descritte in precedenza presentano proprietà non-funzionali uniche che rendono ciascuna di esse una valida soluzione a differenti questioni architetturali, per cui non esiste una soluzione unica, ma a seconda delle caratteristiche e requisiti che devono essere garantiti, è necessario selezionare la soluzione architetturale ottimale.

Architetture basate sui servizi, come SOA e MSA, e ad eventi (EDA), sono pensate per sistemi di grandi dimensioni e sono molto utilizzate per l'implementazione di applicazioni enterprise e web-based che richiedono alta scalabilità e facile manutenibilità, oltre a garantire incapsulamento, basso accoppiamento, alta componibilità e riuso, caratteristiche che non sono generalmente garantite da architetture monolitiche.

Le SOA sono efficiaci per sistemi enterprise complessi composti dall'eterogeneità di applicazioni e servizi [42], per cui tende ad essere una delle soluzioni più adottate da aziende dove c'è forte condivisione e dipendenza tra servizi che utilizzano dati e protocolli di tipo diverso, per via dell'acquisizione in tempi diversi delle varie tecnologie; architetture a microservizi sono più adatte a sistemi di dimensione ridotta e ben partizionati, piuttosto che sistemi en-

terprise su larga scala e permettono maggiore modularità, riuso e scalabilità rispetto alle altre, esponendo API che possono essere utilizzate da altri servizi [73] e riguarda la creazione di componenti con funzionalità atomiche, che possono essere esposte come servizi, i maggiori benefici di queste architetture riguardano lo sviluppo e il deploy indipendente dai servizi che si appoggiano ad esse, e favoriscono la scalabilità in maniera indipendente rispetto all'intero ecosistema delle applicazioni; servizi progettati in questo modo facilitano la comunicazione mediante API REST o Messaging Queues (MQ) [79]; inoltre presentano basso accoppiamento, in modo tale da ridurre la dipendenza tra i servizi e facilitare la modifica senza causare effetti a cascata su altri [79]; favoriscono il riuso, in quanto modulari, ma necessitano di capacità robuste in termini di CI-CD per garantire la compatibilità tra i servizi ed implementare correttamente architetture a microservizi che garantiscano le caratteristiche sopra riportate [3].

1.3.1 Differenza tra Enterprise Service Bus e Message-Oriented Middleware

Nell'ambito dell'integrazione di sistemi distribuiti, la scelta tra Enterprise Service Bus (ESB) e Message-Oriented Middleware (MOM) rappresenta una decisione architettuale cruciale che impatta significativamente sui requisiti non-funzionali del sistema.

Entrambe le tecnologie offrono soluzioni per la comunicazione inter-servizio, ma si differenziano sostanzialmente nell'approccio, nella complessità implementativa e nelle garanzie fornite:

- **Enterprise Service Bus**: tecnologia più complessa e costosa da implementare e mantenere rispetto ai message-oriented middleware, può essere utilizzato come MOM, ma aggiunge anche altre funzionalità, agisce come mezzo di comunicazione tra diverse applicazioni enterprise fornendo un'integrazione tra i diversi servizi mediante l'uso di API (ogni servizio che utilizza l'ESB, espone mediante esso una API che attraverso diversi canali può essere utilizzata per la comunicazione e traduzione dei dati tra i servizi). Permette la conversione tra diversi protocolli e trasformazione dei dati tra diversi formati. Sono tipicamente utilizzati per integrazioni complesse tra diversi sistemi e servizi ed è possibile utilizzare soluzioni decentralizzate per evitare che diventi un 'single point of failure' [21] [50].

  Il principale svantaggio è dovuto al fatto che non comprende una logica di rilevazione automatica del destinatario di un messaggio, ma il servizio che utilizza un ESB deve comunicare mediante uno specifico canale per inoltrare un messaggio ad un servizio in un certo formato, ed utilizzarne

un altro per altre tipologie di formati, meccanismo che non è automatizzato dall'ESB ma ogni servizio deve selezionare autonomamente il canale corrispondente.

La centralizzazione di ogni tipo di comunicazione tra servizi, rallenta la capacità di evoluzione ed adattamento ai cambiamenti rispetto a nuove tecnologie, che devono essere adattate ed implementate ad ogni nuovo cambiamento. I vantaggi riguardano la traduzione dei dati da un formato ad un altro per permettere l'interoperabilità tra servizi sia legacy che più recenti [50].

- **Message-Oriented Middleware**: tecnologia più semplice rispetto ad un ESB, abilita lo scambio di messaggi tra servizi di uno stesso sistema, effettuando routing di messaggi tra una e più destinazioni in un ambiente distribuito composto da piattaforme eterogenee; permette un forte disaccoppiamento tra producer e consumer, che non necessitano di connessioni dirette, per cui non è necessario che ciascun servizio conosca la locazione degli altri, mantenendo quindi un livello di astrazione più alto; i messaggi sono garantiti essere inviati nello stesso ordine in cui sono ricevuti.

  Alcune implementazioni allo stato dell'arte hanno architetture distribuite composte da diversi broker per cluster che garantiscono maggiore scalabilità e rapidità nell'invio di messaggi, oltre che facilitare la raccolta ed elaborazione di log scambiati tra i servizi mediante propri database, inoltre, considerando il caso di architetture distribuite, si evitano centralizzazioni che possono diventare colli di bottiglia e rallentare l'intero sistema, non rivelandosi 'single point of failure' e quindi permettendo maggiore resilienza [43] [80].

  Il principale svantaggio è la mancanza di traduzione da un formato ad un altro dei messaggi, come può essere JSON a XML e viceversa, per cui è necessario concordare a priori tra i diversi servizi un linguaggio comune per lo scambio dei dati o una logica di traduzione dei dati incapsulata all'interno di ogni servizio per poter continuare ad utilizzare il sistema già esistente.

  Questa tecnologia è una buona soluzione per architetture a microservizi dove è necessaria alta disponibilità e scalabilità, e permette una maggiore robustezza rispetto a soluzioni centralizzate. Considerando 'event brokers' è possibile utilizzare broker per scambio di eventi che mantengono le informazioni scambiate tra i servizi all'interno di un proprio database (a differenza di 'message brokers' dove le comunicazioni tra i servizi sono di tipo 'fire-and-forget' dove le informazioni scambiate non sono mante-

nute), e che può essere eventualmente utilizzato come 'single source of truth' tra i servizi che ne fanno uso, in modo da poter richiedere nuovamente tutti gli eventi passati, con diverse applicazioni tra cui quelle di riallineamento dei dati o monitoraggio e analisi della sequenza di eventi scambiati.

1.3.2 Differenza tra le architetture SOA e MSA

La progettazione di sistemi enterprise moderni si trova spesso di fronte alla scelta tra Architetture Orientate ai Servizi e Architetture a Microservizi, due paradigmi che, pur condividendo filosofie simili di decomposizione funzionale, presentano caratteristiche architetturali distintive.

Le differenze tra questi approcci si manifestano principalmente a livello di requisiti non-funzionali, influenzando aspetti cruciali come scalabilità, manutenibilità e resilienza del sistema.

- **Disaccoppiamento e modularità**: entrambe le architetture offrono un buon livello di indipendenza e disaccoppiamento tra i servizi, dove però le architetture a microservizi mostrano un livello maggiore di essi.

  Ciascuna architettura definisce un proprio confine tecnico di cui ha pieno controllo e permette alta riusabilità, mediante API, confinando in un unico servizio la logica che lo caratterizza, questo rende i servizi tra loro sufficientemente isolati da permettere la riduzione di effetti dovuti alla dipendenza tra servizi, rendendo il sistema più solido rispetto ai cambiamenti, ed evitando malfunzionamenti a cascata dovuti tipicamente al forte accoppiamento tra essi.

  Entrambe le architetture hanno tra i principali obiettivi quello del riutilizzo dei servizi, cioè identificare e unificare in un unico servizio tutte le parti comuni a ciascuna divisione che componeva il precedente il sistema, in modo da renderla autonoma e indipendente dalle altre e poterla riutilizzare modularmente.

  La sostanziale differenza tra esse è la granularità con cui viene effettuato il partizionamento dei servizi: nelle architetture a microservizi la granularità è più fine, e questo garantisce diversi vantaggi tra cui quello di ottenere una maggiore modularità e ridurre maggiormente l'accoppiamento tra i servizi, in quanto ogni servizio rappresenta una singola funzionalità e non un insieme di essi, per cui non è necessario dipendere dall'intero servizio per richiedere solo una parte delle funzionalità che offre, come invece accade nelle SOA, per cui ciascun cambiamento ha una maggiore possibilità di causare malfunzionamenti in altri servizi a cui dipende [72].

- **Eterogeneità**: essendo architetture distribuite con un confinamento specifico dei servizi, è possibile l'utilizzo di diversi linguaggi e tecnologie per ciascun servizio, pur garantendo l'interoperabilità tra essi.

- **Interoperabilità**: entrambe le architetture garantiscono l'interoperabilità dei servizi sfruttando standard e convenzioni condivise tra di essi mediante contratti che sono indipendenti dalla logica sottostante.

  Utilizzare l'integration hub di un ESB permette inoltre la comunicazione tra servizi che utilizzano protocolli e standard di comunicazione e scambio di dati differenti, garantendo una maggiore interoperabilità ma aumentando la complessità del sistema, dove invece un broker permette la comunicazione tra servizi che utilizzano lo stesso protocollo e rappresentazione delle informazioni, ma essendo meno complesso permette di ottenere prestazioni maggiori.

- **Scalabilità**: entrambe le architetture mostrano un ottimo livello di scalabilità, dove le MSA mostrano un livello leggermente superiore, dovuto principalmente alla granularità più fine del partizionamento per cui è possibile aggiungere facilmente risorse al solo servizio in cui sono richieste, anziché all'intero ecosistema delle applicazioni, rendendo più semplice la scalabilità che è invece più limitata nel caso di SOA, in quanto la granularità dei servizi è meno fine [3]; inoltre, utilizzare server con architetture a microservizi permette di effettuare scaling orizzontale dove invece architetture monolitiche permetterebbero solamente scaling verticale [81] [37].

- **Performance**: a differenza di altre caratteristiche dove si mostra una netta differenza tra queste due architetture, non c'è una sostanziale differenza nelle prestazioni in quanto entrambe condividono lo stesso principio di partizionamento in diversi servizi, con una architettura distribuita, in cui i servizi comunicano tra loro per lo scambio di informazioni allo stesso modo in entrambe i casi. La differenza sostanziale risiede nell'implementazione e nelle risorse a disposizione dei singoli servizi, che può permettere maggiori prestazioni in una maniera che è però indipendente dall'architettura del sistema complessivo.

- **Robustezza**: architetture a microservizi mostrano una forte robustezza, se implementate in modo da poter governare correttamente eventuali problematiche riguardo al partizionamento della rete e l'isolamento dagli altri microservizi, in quanto ciascun microservizio contiene un proprio database che non è condiviso con altri servizi e permette, mediante l'isolamento dei dati, di garantire il corretto funzionamento nonostante even-

tuali malfunzionamenti di altri servizi [72]. Inoltre essendo la granularità fine e fornendo ciascun servizio una funzionalità atomica rispetto a quanto accade nelle SOA che permette minor accoppiamento e dipendenza tra servizi, il rischio di malfunzionamenti a cascata è ridotto.

Capitolo 2

Standard per l'Interoperabilità nella Sanità Digitale

Prima di parlare delle architetture software utilizzate nel caso di studio di AUSL Romagna, è necessario un approfondimento riguardo gli standard per la codifica, rappresentazione e scambio di dati utilizzati in ambito sanitario.

Al momento della stesura di questo documento, gli standard utilizzati per la codifica dei dati sono regolati a livello nazionale ed internazionale, per cui è necessario seguire le regole lì definite. Per quanto riguarda standard per la persistenza e la condivisione di dati sanitari, negli ultimi anni, due di questi hanno suscitato un certo interesse: HL7 FHIR e OpenEHR, e sono attualmente utilizzati dalla maggior parte delle aziende di questo settore, motivo per cui verranno descritti e trattati più nel dettaglio nelle prossime sezioni, concludendo con un confronto che mette in luce le differenze tra i due standard.

2.1 Standard nell'ambito informatico

Gli standard sono documenti che stabiliscono regole, linee guida o specifiche tecniche che garantiscono l'interoperabilità e la compatibilità tra diverse tecnologie che possono essere utilizzate nei sistemi informatici.

Riguardo lo specifico ambito dell'informatica medica, i dati che sono presi in considerazione sono sono denominati EHR (Electronic Health Record, in italiano Fascicolo Sanitario Elettronico o Cartella Clinica Elettronica) ed è la raccolta digitale e strutturata di tutte le informazioni cliniche di un paziente prodotte e utilizzate nell'ambito del sistema sanitario.

Standard informatici per la rappresentazione comune di EHR, sono utilizzati per garantire l'interoperabilità tra diverse tecnologie e servizi del sistema informativo sanitario.

2.2 Standard per la codifica dei dati sanitari

Regole nazionali ed internazionali definiscono standard per la codifica dei dati, in modo da avere una rappresentazione comune di essi:

- **International Classification of Diseases (ICD)**: sistema internazionale di classificazione delle malattie, interventi chirurgici e procedure diagnostiche e terapeutiche
- **Logical Observation Identifiers Names and Codes (LOINC)**: standard internazionale di codifica e descrizione delle osservazioni cliniche e di laboratorio che permette di leggere ed interpretare dati provenienti da sistemi operativi differenti, identificando univocamente gli esami.
- **Systematized Nomenclature of Medicine, Clinical Terms (SNOMED CT)**: dizionario multilingue di termini clinici standardizzati che facilita lo scambio di informazioni rilevanti tra gli operatori sanitari

2.3 Standard per la condivisione e la persistenza dei dati elettronici sanitari

Come anticipato in precedenza, gli standard attualmente più utilizzati dalle aziende di questo settore, sono FHIR ed OpenEHR. Ciascuno di essi ha come obiettivo quello di garantire l'interoperabilità tra i servizi del sistema informativo sanitario e permettere di raggiungerlo mediante approcci differenti; l'uno definendo delle regole per la condivisione e il recupero di informazioni sanitarie tra i diversi servizi del sistema informativo, e l'altro proponendo un modello comune per la persistenza dei dati.

2.3.1 HL7 FHIR R5

Health Level Seven (HL7) [36] fa riferimento ad un insieme di standard e framework pensati per facilitare lo scambio, l'integrazione, la condivisione e il recupero di informazioni sanitarie in modo efficiente tra diverse applicazioni e sistemi informativi, e si concentra sul livello 7, livello applicativo del modello Open Systems Interconnection (OSI) [38].

Tra questi standard, figura lo standard FHIR [27], acronimo di Fast Healthcare Interoperability Resources, standard di nuova generazione che permette l'interoperabilità tra sistemi informativi sanitari e dispositivi biomedicali. In

questa trattazione si farà riferimento all'ultima versione rilasciata al momento della stesura di questo documento, la release 5 (R5).

Lo standard FHIR è pensato per uno scambio sicuro di dati elettronici di tipo sanitario (EHR), in modo flessibile e adattabile, in modo da poter essere utilizzato per diverse tipologie di dispositivi e sistemi informativi, e per la semplicità e rapidità di adottamento, comprensione e utilizzazione, con una una filosofia fortemente orientata al dominio.

Combina le caratteristiche migliori e principali di diversi altri standard che lo hanno preceduto con standard moderni web-based per permettere l'intercomunicazione dei sistemi informatici, imponendo una definizione per la rappresentazione dei dati ad esempio per API REST in JSON, e XML o RDF.

In questo elaborato ci è concentrati solamente su alcuni aspetti che sono stati ritenuti rilevanti per il caso d'uso specifico di AUSL Romagna che verrà affrontato in seguito, in quanto l'applicazione di questo standard non sempre utilizza correttamente le regole di conformità definite dallo standard ed atte a garantire l'interoperabilità tra i servizi del sistema informativo sanitario eterogeneo considerato. Se applicate, permettono di sfruttare tutti i benefici di interoperabilità, flessibilità e adattabilità che il modello mette a disposizione, se applicato in modo più conservativo si rischia di ottenere l'effetto contrario.

Moduli

Il framework FHIR è composto di diversi moduli, ciascuno dei quali si occupa di una specifica area del dominio, per cui il linguaggio ubiquo è già parte del modello, ciascun modulo contiene un insieme di terminologie e un modello specifico per come le informazioni devono essere codificate e sono visualizzabili alla Figura 2.1. Con questo elaborato ci si concentrerà prevalentemente sui primi due livelli.

Risorse

Il componente base del modello FHIR è la 'resource' [30], ogni categoria di dato disponibile nel settore sanitario può essere rappresentato come una risorsa (e.g., battito cardiaco, procedure, farmaci, allergie, ecc...).

Le diverse tipologie di risorse sono raggruppate per livelli, ciascuno dei quali contiene informazioni riguardo una specifica area del dominio, un elenco delle risorse è visualizzabile alla Figura 2.2).

Ciascuna risorsa è composta da una serie di caratteristiche comuni, tra cui identificatori, versioni FHIR di riferimento a cui la risorsa è conforme, etichette di sicurezza che verificano l'autorizzazione a letture e scritture sulle risorse, i dati che possono essere restituiti in base al ruolo di chi richiede la risorsa, e 'riferimenti' ad altre risorse, che permettono di combinare risorse per formare

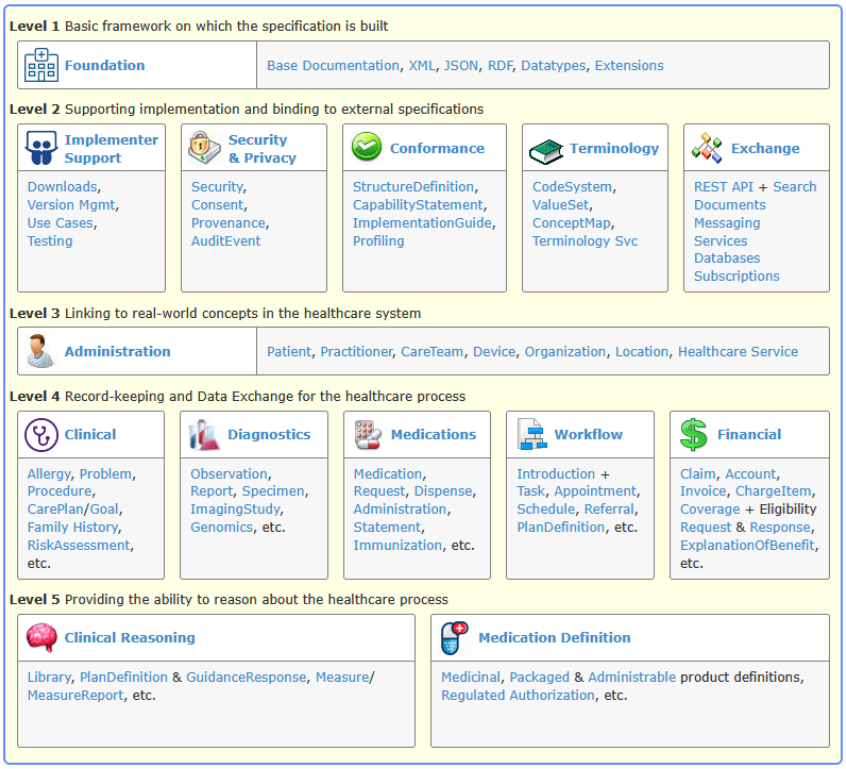

Figura 2.1: Moduli che compongono lo standard FHIR

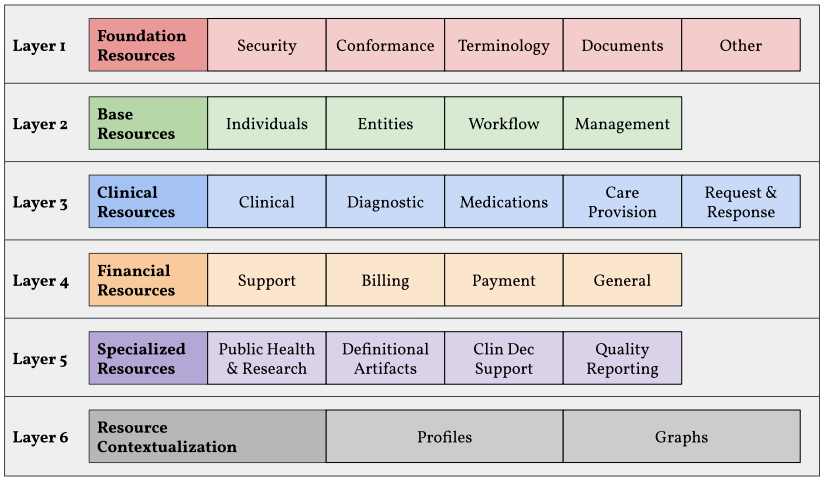

Figura 2.2: Suddivisione in livelli delle risorse FHIR

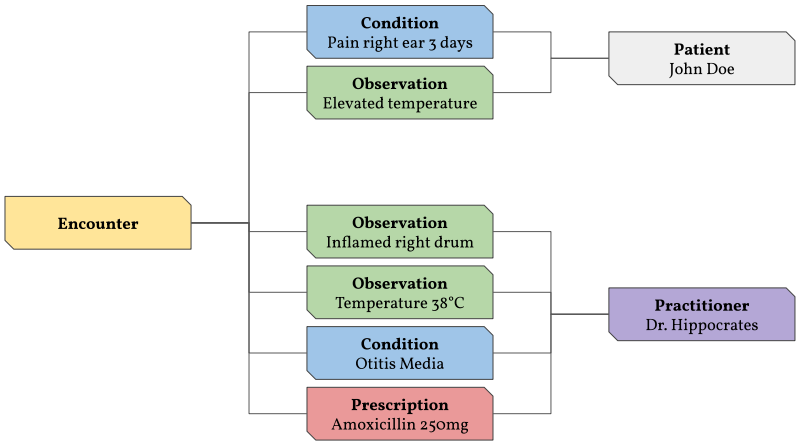

Figura 2.3: Esempio di risorse FHIR correlate

casi d'uso specifici mediante una rete di informazioni di URL o URI (come 'uuid' e 'oid'), senza così deviare dalla struttura base definita dalla specifica FHIR.

Per raggruppare un insieme di risorse tra loro correlate in una collezione, è possibile utilizzare la risorsa 'Bundle', che è utilizzata come contenitore di risorse, in modo da essere acceduto come un'unica risorsa formata da un aggregato di esse (vd. Figura 2.4).

Per inserire vincoli e regole sugli elementi che compongono le singole risorse, sono previsti meccanismi denominati 'profili', che consentono di adattare le risorse standard ad esigenze specifiche senza deviare dalla struttura di base, adattando quelle generiche già esistenti a casi d'uso specifici, tramite le 'estensioni'.

L'aggiunta di restrizioni mediante i profili deve seguire una 'definizione strutturale' che definisce le regole da rispettare per effettuarlo. Alcuni vincoli riguardano ad esempio la cardinalità degli elementi all'interno di una risorsa, tipi di elementi permessi, elementi di cui si hanno riferimenti che devono essere in linea con altri profili pre-esistenti e definizioni di codifiche del dato da rispettare.

Regole di conformità

Applicazioni che seguono alcune regole riguardo framework per lo scambio dei dati, sono dette conformi ad esse. Non è obbligatorio seguire queste regole, ma le applicazioni conformi possono dichiarare che seguono le regole definite nella documentazione e permettono una maggiore interoperabilità tra esse.

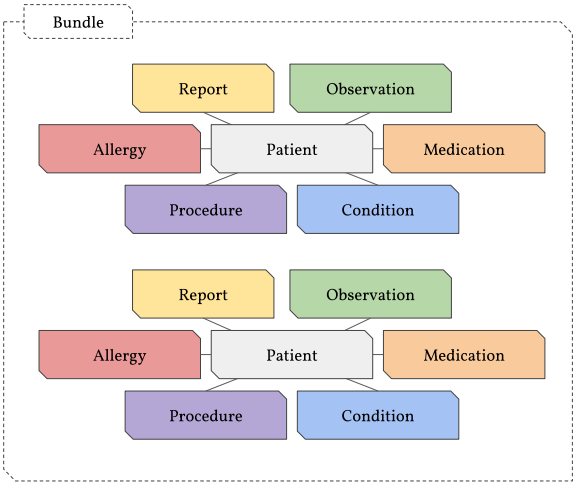

Figura 2.4: Esempio di Bundle di risorse FHIR

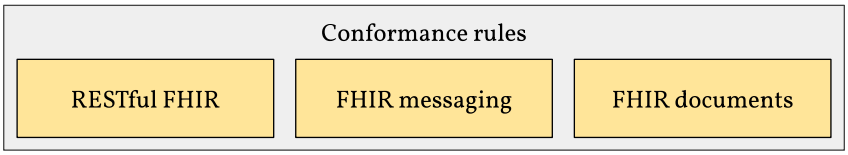

Figura 2.5: Regole di conformità

Essendo pensate per applicazioni di diversa natura, le regole definite per le applicazioni non garantiscono comunque totale interoperabilità.

Le applicazioni possono dichiarare di essere conformi a diverse tipologie di framework (vd. Figura 2.5):

- **RESTful FHIR**: riguardo la conformità alle regole sulle API REST.
- **FHIR messaging**: riguardo la conformità sullo scambio di messaggi tra applicazioni.
- **FHIR documents**: riguardo la conformità sullo scambio di documenti tra applicazioni.

Ciascuna di esse verrà spiegata nelle prossime sezioni più nel dettaglio. Per confermare la conformità ad esse, è necessario pubblicare una dichiarazione di capacità, 'CapabilityStatement', e può anche contenere alcune informazioni come profili da seguire per rappresentare bundle di risorse che rappresentano casi d'uso specifici.

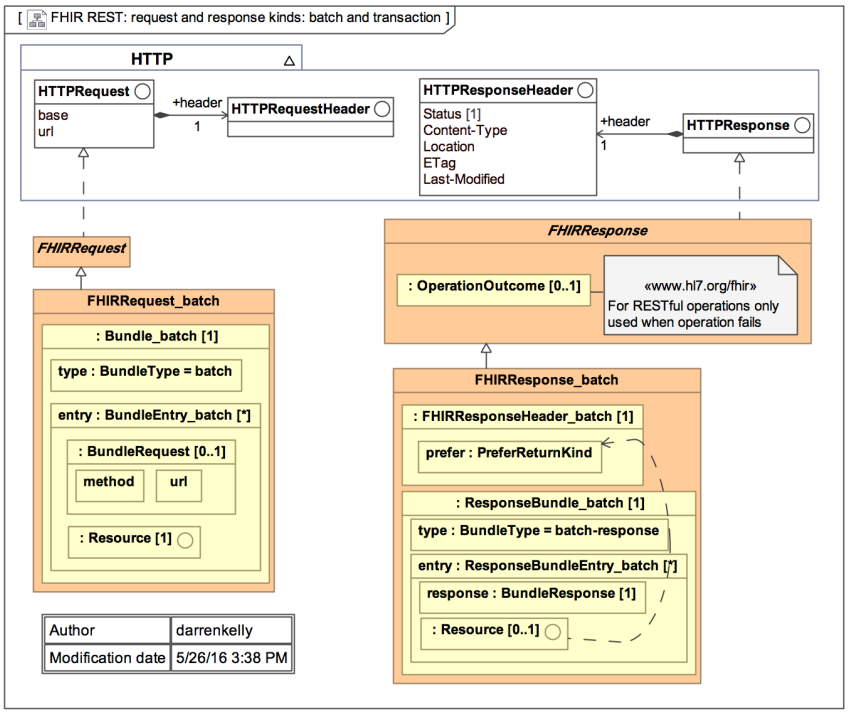

Figura 2.6: Struttura richieste e risposte HTTP standard su FHIR

RESTful FHIR

Anche se non obbligatorio, la specifica FHIR è progettata per essere usata principalmente con interfacce RESTful. L'utilizzo di HTTPS è opzionale ma per ragioni di sicurezza, è fortemente consigliato. Le transazioni FHIR seguono un semplice pattern di request e response, e le modalità con cui questo può avvenire sono riportate alla Figura 2.6.

Seguendo lo stile architetturale RESTful definito dalla specifica FHIR [34], che impone un albero di navigazione di una certa tipologia in modo da rendere una applicazione conforme a 'RESTful FHIR', la API descrive le risorse come un set di operazioni (chiamate 'interazioni') su risorse dove le istanze per la risorsa individuale sono gestite in collezioni in base al loro tipo.

Lato server si può scegliere quali interazioni rendere disponibili e quali tipi di risorse supportano, e successivamente dichiarare quali interazioni e risorse sono supportate. Un elenco dettagliato di tutte le possibili interazioni, e come devono essere definite per essere conformi, è visualizzabile alla documentazione riguardo lo stile RESTful definito nella specifica FHIR.

Le interazioni sono definite come:

```
VERB [base]/[type]/[id] ?_format=[mime-type]
```

Un esempio è il seguente, dove si mostrano le interazioni **read** e **vread** che permettono di effettuare la lettura di una risorsa (all'ultima versione disponibile) e la lettura di una specifica versione di una risorsa rispettivamente.

```
GET http://server.org/patients/Patient/23424?_pretty=true
```

```
GET
http://server.org/patients/Patient/23424/_history/1?_pretty=true
```

Alcune opzioni che possono essere specificate riguardano ad esempio l'inclusione di tutte le risorse anziché riferimenti ad esse, utilizzando l'opzione `_include`:

```
GET /Patient/123?_include=Patient:organization
```

In questo caso si richiede la risorsa paziente e invece del riferimento alla risorsa organization, richiede esplicitamente anche quella risorsa per intero.

Come verrà mostrato alla sezione 4.4, se le richieste da parte del client non richiedono le risorse complete (quindi non è stata inserita l'opzione `_include`), allora la risposta conterrà la risorsa Bundle con solamente la risorsa richiesta e i riferimenti alle altre a cui è correlato; in caso contrario, un Bundle con tutte le risorse richieste per intero.

Per garantire questo tipo di funzionamento, nel primo caso sarà sufficiente inserire la risorsa nella risposta, mentre nel secondo sarà necessario effettuare più richieste da parte del client in modo da ottenere per intero tutte le risorse richieste.

L'API RESTful può essere estesa per supportare altre operazioni oltre alle classiche interazioni descritte in precedenza che seguono le azioni CRUD (**C**reate, **R**ead, **U**pdate, **D**elete) sulle repository della risorsa considerata. Queste funzionalità sono chiamate 'operations', e forniscono alcune funzioni simili al paradigma RPC dove le operazioni hanno parametri di input e output, fornendo l'estensione **E**xecute, e sono utilizzate nel caso in cui il server ricopra un ruolo essenziale per l'elaborazione della risposta e non solo restituire l'informazione richiesta, e si richieda una elaborazione di risorse non trasmesse dalla richiesta, come 'side effects' [29] [26].

FHIR messaging

Per scambiare messaggi tra i sistemi, è possibile utilizzare RESTful descritto in precedenza, anche se non obbligatorio.

In FHIR un messaggio è inoltrato da una applicazione sorgente ad una applicazione di destinazione quando avviene un evento, che solitamente corrispondono a cose che accadono nel mondo reale [28].

I messaggi devono essere formattati seguendo una specifica formattazione descritta nella documentazione, e sono risorse di tipo Bundle che devono essere definiti secondo regole specifiche per cui possono contenere riferimenti ad altre risorse, in modo da ridurre la dimensione del payload delle richieste, ma richiedono quindi un numero maggiore di messaggi per poter raccogliere tutte le informazioni necessarie [28] [22].

Riguardo lo specifico caso d'uso RESTful, i messaggi sono rappresentati dalla risorsa Bundle, dato che l'identità del messaggio è rappresentata dall'identificativo del Bundle stesso. Un esempio di richiesta e risposta sono visualizzabili ai collegamenti riportati in bibliografia [32][33].

Riguardo il capability statement per 'FHIR messaging', questo deve contenere una lista di tutti i messaggi di eventi supportati (sia come ricevitore che come mittente) e per ciascun evento, un profilo che indica le risorse che devono essere contenute all'interno del bundle (se mittente) o che è richiesto siano contenute (se ricevitore), e ciascuna regola riguardo il contenuto informativo di ciascuna risorsa, in modo da esporre all'esterno dell'applicazione quali richieste si aspetti di ricevere e quali risposte inoltrerà.

FHIR documents

La specifica FHIR definisce anche delle regole per lo scambio di documenti, dove il contenuto da scambiare è contenuto all'interno della risorsa 'Composition', una risorsa immutabile che definisce una struttura e contenuti necessari per documenti definendo un proprio contesto e una attestazione clinica relativa a chi rilascia la dichiarazione. E' definita come il primo elemento di un Bundle formattato secondo regole specifiche visualizzabili alla documentazione [24] [25].

Un documento rappresenta un insieme di informazioni immutabili che riguardano una dichiarazione a proposito di informazioni di tipo sanitario, comprese osservazioni e servizi, ed è redatto da umani, organizzazioni e/o servizi (in modo che chi effettui queste dichiarazioni sia tracciato). Le risorse di cui si ha riferimento all'interno di Composition sono diverse, ed è necessario che siano tutte presenti. Un esempio di documento è riportato alla Figura 2.7.

Archiviazione di dati FHIR

Non esiste uno standard o un approccio consigliato su come debbano essere archiviate le risorse FHIR nei database. Generalmente sono salvate per intero per come sono descritte nella documentazione, utilizzando le tecnologie a disposizione come database relazionali SQL (ad esempio con supporto JSON), o non relazionali NoSQL (e.g., MongoDB, Hadoop), nel formato più convenien-

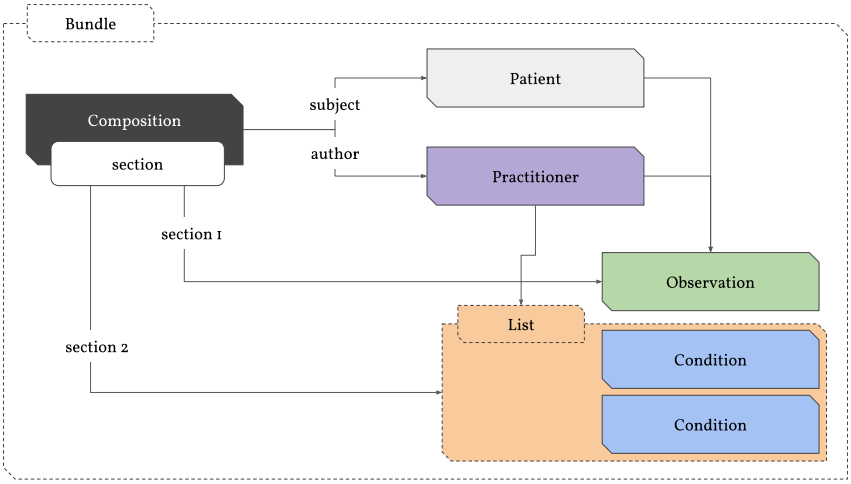

Figura 2.7: Esempio di documento

te (e.g., JSON, RDF in formato Turtle, ecc...) [35], come anche riportato in diversi casi d'uso specifici [82] [2]

Obiettivi rispetto ai principi non-funzionali

FHIR nasce per permettere una migliore interoperabilità nel settore sanitario, garantendo che i sistemi possano scambiarsi dati con struttura e significato precisi. L'obiettivo è quello di utilizzare modelli di dati espressivi insieme a metodi di scambio semplificati, affinché le informazioni possano essere comprese e utilizzate da diversi sistemi in modo coerente [31]. Tra i principi architetturali su cui si basa troviamo:

- **Riutilizzo e Componibilità**: è progettato per coprire la maggior parte delle esigenze di interoperabilità con un basso numero di specifiche, concentrandosi sui requisiti comuni e generali. Questo significa che le risorse a disposizione (ad esempio, "paziente", "procedure", ecc..) sono create per soddisfare in modo generico le necessità più frequenti, evitando la creazione di un numero eccessivo di risorse parzialmente sovrapposte o ridondanti, e possono essere estese con vincoli mediante profili o formare risorse più complesse mediante riferimenti, in modo da rimanere conformi al modello base.

- **Scalabilità**: L'adozione dello stile REST per le API FHIR rende ogni transazione 'stateless', il che significa che ogni richiesta contiene tutte le informazioni necessarie, riducendo l'uso della memoria, ed eliminando la necessità di 'sticky sessions' in un ambiente con più server, supportando la scalabilità orizzontale.

- **Performance**: le risorse sono pensate per essere leggere, in modo da poter essere scambiate rapidamente attraverso la rete, anche in transazioni complesse che coinvolgono più sistemi, garantendo payload di dimensione più ridotta a fronte di un maggiore numero di richieste per ottenere informazioni complete.

- **Usabilità**: le risorse sono scambiate in formato XML o JSON che è visualizzabile tramite browser, rendendo semplice la comprensione del loro contenuto da chiunque.

2.3.2 OpenEHR

OpenEHR è un'organizzazione non-profit che pubblica standard per la realizzazione di piattaforme per la gestione di dati elettronici di tipo sanitario (EHR) in modo uniforme, strutturando i dati in un formato standard, con un focus sulla gestione e persistenza del dato, il reperimento e lo scambio di dati sanitari nelle cartelle cliniche elettroniche (EHR), in modo da avere il paziente al centro, il mantenimento di uno storico ed il supporto ai processi clinici [61].

I principi base su cui si fonda sono l'interazione con gli esperti del dominio, per creare modelli che aderiscano alle esigenze piuttosto che usare modelli sviluppati ad hoc da aziende informatiche, una personalizzazione efficace tramite strumenti low-code, ed interoperabilità garantita a livello architetturale. I contributi di OpenEHR sono di diverso tipo, organizzati secondo diversi programmi.

Il principale riguarda la produzione di specifiche che permettano di uniformare architetture, terminologia e processi legati alla gestione dei dati sanitari. Altri contributi sono invece in campo clinico, con la raccolta di dataset e basi di conoscenza legate al dominio medico, attività di formazione e disseminazione e attività di sviluppo software per fornire implementazioni base degli standard.

In questa sezione si ci soffermerà principalmente sull'aspetto di persistenza dei dati, in quanto scopo principale di questo standard, che è ottimizzato per raggiungere l'interoperabilità riguardo l'archiviazione e l'interrogazione dei dati sanitari.

Panoramica generale dal punto di vista architetturale

La specifica di OpenEHR è suddivisa in diversi componenti (vd. Figura 2.8) [52]. Queste possono poi essere mappate sui specifici riferimenti tecnologici come JSON ed XML che ne consentono l'uso diretto nello sviluppo

L'architettura è interamente basata sul concetto di avere diversi livelli di modellazione:

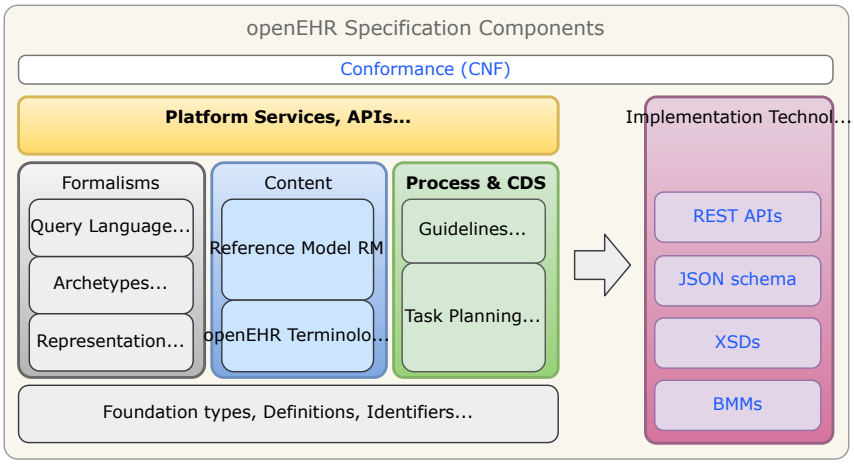

Figura 2.8: Componenti della specifica OpenEHR

- **Reference model (RM)**: il primo livello, è dato da un modello generico e stabile da usare come riferimento
- **Re-usable content element definitions**: il secondo livello, definisce degli 'archetipi' che rappresentano definizioni di elementi riusabili.
- **Context-specific data set definitions**: il terzo livello, definizioni formali per casi d'uso specifici, è dato dalla aggregazione di diversi archetipi nella forma di 'template'.

Questo ha un impatto sul processo di software engineering e design del sistema. Il cuore del sistema è infatti basato sul 'reference model' che essendo stabile permette di mantenere il sistema nel tempo, mentre gli esperti del dominio producono archetipi e template che vengono poi utilizzati per la creazione delle applicazioni e servizi che utilizzano la piattaforma di EHR [62].

A partire dal RM che contiene la struttura sulla rappresentazione dei dati, si generano degli archetipi che contengono gli elementi base lì contenuti, per poter definire possibili dati specifici definiti dagli esperti del dominio in base al contesto considerato.

Mediante template si possono formare aggregati di archetipi che possono essere utilizzati come base per creare dati sanitari (EHR) che sono poi archiviati nei database [53] [56]. Per effettuare l'estrazione dei dati, si utilizza una estensione del linguaggio d'interrogazione SQL denominata AQL (Archetype Query Language) [54], (vd. Figura 2.9).

Il secondo principio alla base della specifica OpenEHR è la separazione delle responsabilità, in domini complessi (come quelli sanitari), per gestire la complessità è necessario suddividere le funzionalità in aree ampie e ben definite,

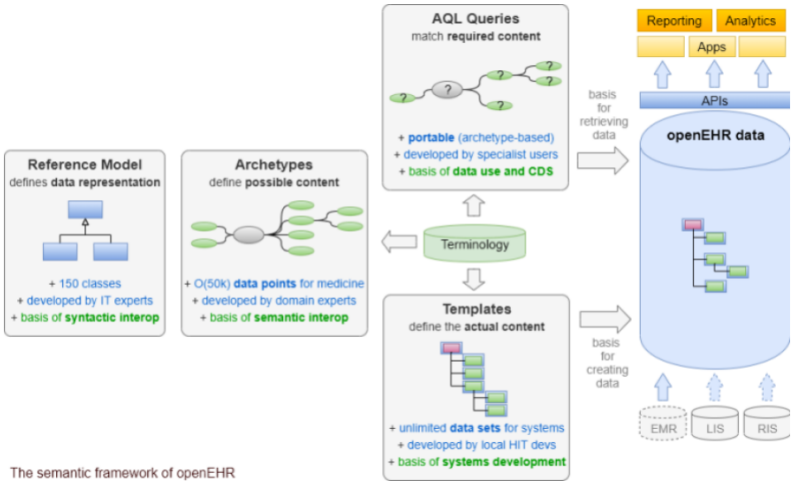

Figura 2.9: Modellazione multi-livello di dati su OpenEHR

creando un sistema di diversi servizi, il modello tipicamente più utilizzato, è quello basato sulle SOA, per dividere le responsabilità di un complesso sistema sanitario in un insieme di servizi che comunicano tra loro [55].

L'idea è che adottando un modello comune questi servizi possano collaborare in modo più efficace, dalla scala locale a quella nazionale e in diverse fasi del percorso clinico del paziente.

Gestione dei dati

Ogni record EHR ha una struttura dati complessa [60] che è riportata in Figura 2.10 e comprende un oggetto che regola l'accesso alle informazioni, controllando le impostazioni a riguardo, un oggetto contenente informazioni di controllo e di stato del record e un contenitore di tutti i dati clinici e amministrativi del record, a loro volta sono strutturate in sezioni gerarchiche che possono contenere più osservazioni, azioni, istruzioni e informazioni amministrative. Ciascuna osservazione può comprendere anche uno storico di eventi correlati (vd. Figura 2.11) [65] [64].

Tutte le informazioni contenute in un EHR sono espresse come 'entries'. Ciascuna di esse fa riferimento ad una qualunque singola informazione di tipo clinico, può fare riferimento ad esempio al risultato di un referto o una valutazione psichiatrica. Definiscono la semantica alla base del record per cui sono elemento fondamentale di esso, sono pensate per essere personalizzate in base al tipo di informazione che devono modellare, per cui possono essere modellate con archetipi [59] [57].

Un aspetto importante di questo modello è il modo in cui le informazioni cliniche sono espresse inequivocabilmente con una collocazione temporale che

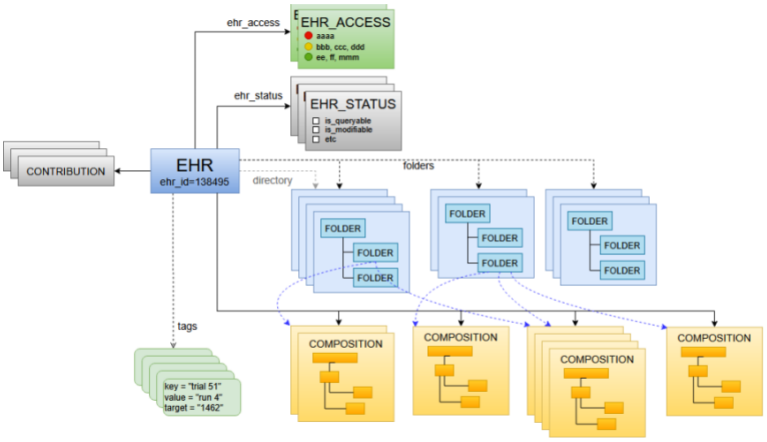

Figura 2.10: Struttura dati OpenEHR

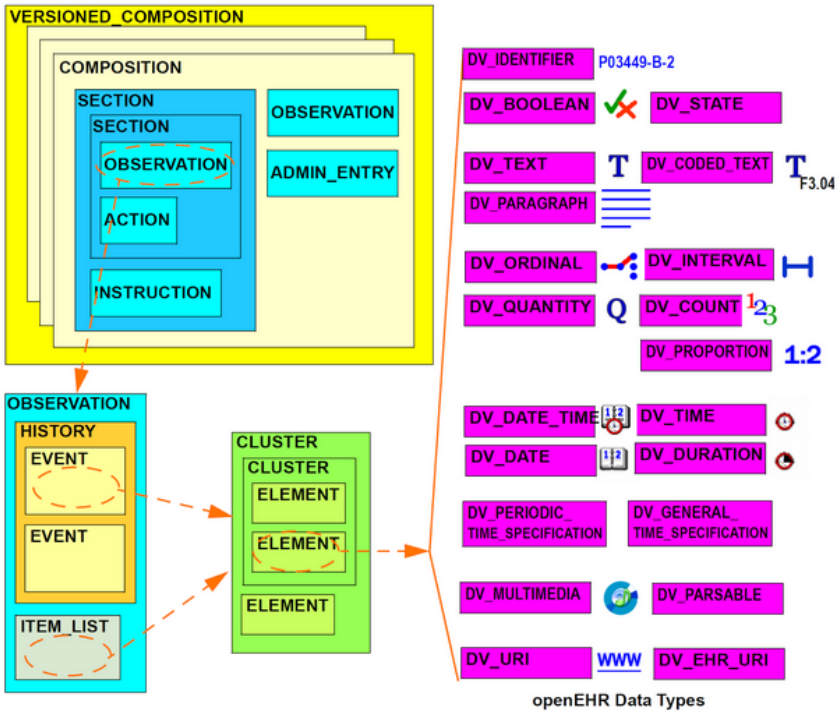

Figura 2.11: Composizione dati OpenEHR

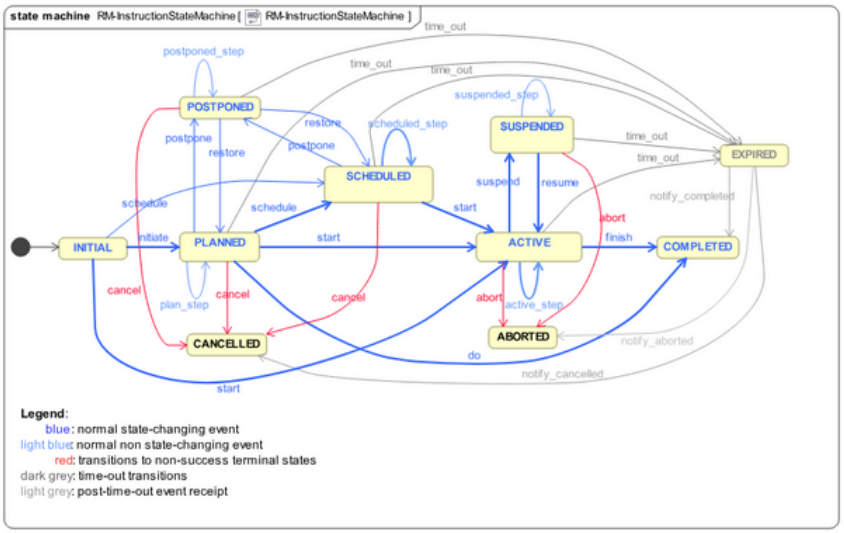

Figura 2.12: Diagramma a stati finiti standard per le istruzioni (ISM)

riguarda se il EHR rappresenta ad esempio una diagnosi da verificare, una misurazione, un fattore di rischio etc.. i diversi sottotipi del EHR comportano una semantica chiara per questo elemento fondamentale nella gestione delle basi di dati cliniche.

Questo è anche riportato nella gestione degli 'interventi' come sopra-tipo di qualunque prestazione medica. Lo stato di un intervento ha una forma complessa che deve tenere conto di diversi fattori che possono variare lo stato dell'intervento stesso in ogni momento (es: reazione ad un farmaco) e può fare riferimento sia a interventi chirurgici che semplici prescrizioni.

Per modellare correttamente questo sistema è necessario quindi definire una entità 'instruction', composta da:

- **Activity**, che rappresenta gli interventi programmati.
- **Action**, riferita a quello che è effettivamente successo.

In questo modo è possibile modellare diversi interventi, sia semplici somministrazioni di farmaci che terapie più complesse.

Lo stato di ogni intervento può essere modellato con un diagramma a stati finiti, in modo da monitorare in ogni momento il suo stato, le attività sono gli stati e le azioni gli archi (vd. Figura 2.12) [63].

Utilizzare una macchina a stati standard per le istruzioni (ISM) come quella mostrata sopra, permette di definire stati e transizioni standardizzate per ciascuna attività, indipendentemente dal dato clinico che si vuole modellare. In questo modo è possibile effettuare una ricerca (query) riguardo gli stati che

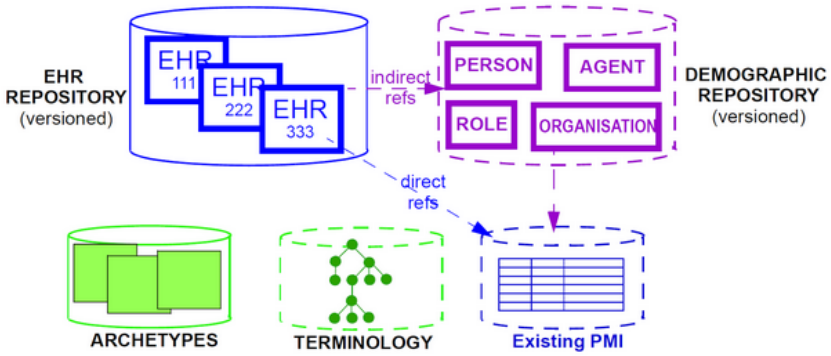

Figura 2.13: Modello architetturale minimo OpenEHR

sono standardizzati, per trovare tutti gli interventi di un certo tipo in uno specifico stato.

Modello Architetturale

Dal punto di vista architetturale OpenEHR identifica un 'sistema' come un repository logico di cui una organizzazione è responsabile, può quindi rappresentare ad esempio sia il servizio sanitario regionale che un singolo ospedale. Il sistema è quindi responsabile dei dati che sono lì contenuti [58].

Un sistema EHR minimale è composto da:

- Un repository di EHR
- Un repository di archetipi e template
- Un repository di informazioni demografiche
- Una formalizzazione della terminologia (opzionale)

Archetipi e Template

A partire dal reference model, dove sono definiti concetti componibili di base invarianti, si possono definire oggetti di dominio e definizioni nella forma di archetipi e template. Gli archetipi sono separati dai dati e salvati in repository a parte, ed il deploy è effettuato mediante template, che specificano un set di archetipi da utilizzare per uno scopo specifico, solitamente un 'form' [66].

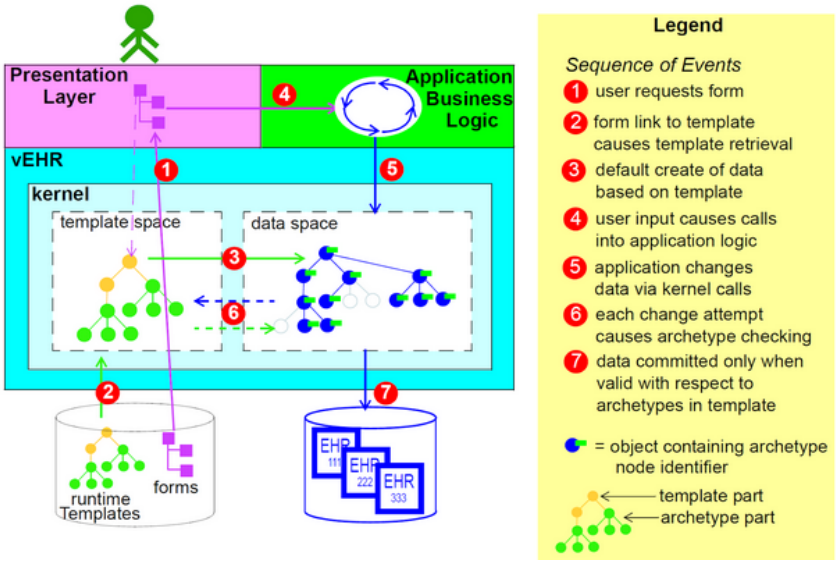

Figura 2.14: Validazione dati su OpenEHR mediante archetipi e template

Gli archetipi sono pensati per essere riutilizzabili e componibili, mediante template è possibile creare, combinando uno o più archetipi, definizioni di contenuti (e.g., referto radiologico, lettera di dimissione ospedaliera, ecc...).

Vengono utilizzati per validare i dati, e salvare solamente le informazioni effettivamente corrette sull'EHR.

1. L'utente richiede un form.
2. Essendo il form associato ad un template a runtime, questo viene estratto dallo spazio dei template.
3. L'input fornito dall'utente causa chiamate nel layer application, che modifica i dati del template.
4. Ogni tentativo di cambiamento dei dati viene controllato e validato in base all'archetipo utilizzato.
5. I dati inseriti sono mantenuti e registrati solamente se considerati validi rispetto agli archetipi del template.

Quanto descritto è rappresentato in Figura 2.14.

Gli archetipi vengono anche utilizzati come supporto alle query, con una estensione di SQL: AQL (Archetype Query Language), che supporta la definizione di query basate sulla struttura degli archetipi. Essendo gli archetipi

generici e legati da una struttura gerarchica questo permette di specificare query con grande flessibilità.

2.4 Differenze tra gli standard HL7 FHIR ed OpenEHR

Gli standard sopra descritti pongono l'attenzione su aspetti differenti, FHIR si concentra sullo scambio di dati sanitari, mentre OpenEHR mette il focus sulla modellazione dei dati e la persistenza.

Entrambi utilizzano standard IT già esistenti e comunemente utilizzati, come il protocollo HTTP per il layer di application e XML, JSON per la formattazione dei dati, ed entrambi assumono transazioni di tipo stateless.

Di seguito una analisi e comparazione tra i due standard per mettere in luce problematiche e soluzioni riguardo requisiti architetturali di interoperabilità ed altro [39].

Entrambi gli standard sono stati definiti con l'intento di facilitare l'interoperabilità, la garantiscono in modi differenti:

Lo standard FHIR non richiede che il dato venga archiviato in un modo specifico, ma necessita solo di un formato specifico per lo scambio di esso, per cui la persitenza del dato può essere effettuata nel formato più appropriato scelto dal servizio stesso, alcune applicazioni concrete dimostrano come si possano utilizzare database NoSQL per archiviare documenti e messaggi FHIR come sono, garantendo maggiore flessibilità [2].

È inoltre pensato per essere compatibile con gli altri standard HL7, per cui l'interoperabilità tra servizi che usano già standard HL7 è agevolata

Questo standard permette ad ogni servizio di comunicare con gli altri seguendo le regole di conformità definite nella documentazione, per cui è solo necessaria una traduzione dei dati dal modello utilizzato internamente al servizio, a quello di scambio dei dati FHIR conforme a tutti i servizi, in modo da garantire la corretta comunicazione tra servizi pre-esistenti provenienti da diversi fornitori [28] [24].

Lo standard OpenEHR fornisce invece una soluzione per la persistenza del dato, ogni sistema che usa OpenEHR è capace di interpretare e processare dati di qualunque altro sistema che usa lo stesso standard e dovrebbe essere possibile scambiare repository tra servizi senza effettuare alcuna modifica.

In questo modo l'interoperabilità è garantita solo se tutti i servizi utilizzano OpenEHR e in tal caso non ci sarebbe nessuna necessità di traduzione dei dati tra sistemi, anche se alcuni articoli hanno messo in evidenza che la semplicità della struttura di archiviazione dei dati induce una logica di recupero dati

complessa che impatta negativamente sulle prestazioni riguardo il recupero dei dati, che richiedono flessibilità [89].

Per quanto riguarda lo scambio dei dati, FHIR adotta una rappresentazione meno stratificata dei dati rispetto all'organizzazione gerarchica di OpenEHR, utilizzando un riferimento alle risorse piuttosto che includere l'intera risorsa in ciascun messaggio, ed è sempre possibile una traduzione dall'uno all'altro standard [39] abilitando di fatto anche l'uso parallelo di entrambi, in modo complementare, permettendo perciò l'uso di uno e dell'altro in modo da superare le limitazioni di ciascuno e applicarli per i casi d'uso per cui sono stati pensati; è quindi possibile utilizzare OpenEHR per la persistenza dei dati e FHIR per il loro scambio, in modo da applicarli solamente laddove ottimizzati [69] [1]. Inoltre, indipendentemente dallo standard, è possibile supportare qualunque database, sia relazionale che NoSQL.

Diversi studi che hanno raccolto progetti e articoli riguardo l'uso di FHIR per l'interoperabilità tra diverse applicazioni sanitarie, ha mostrato l'efficacia di questo standard e maggiori prestazioni rispetto allo standard OpenEHR [70] [44]. Altri studi hanno scelto di utilizzare lo standard HL7 FHIR per via della sua semplice conversione ad altri formati di dati tipicamente utilizzati nell'analisi dei dati sanitari [10].

Si conclude che entrambi gli standard possono essere utilizzati in maniera efficace per ogni applicazione di tipo sanitario, adottando le risorse del modello di riferimento in base al caso d'uso specifico mediante profili FHIR e archetipi OpenEHR, ma lo standard FHIR presenta una complessità più ridotta, soprattutto dovuta alla struttura dei messaggi che permettono di scambiare riferimenti alle risorse, anziché le risorse per intero come invece accade in OpenEHR, riducendo la dimensione del 'payload' delle richieste HTTP a scapito di un numero di richieste più alto per poter ottenere informazioni complete; questo permette generalmente prestazioni migliori rispetto allo standard OpenEHR, e semplifica la traduzione da questo standard ad altri, in modo da poter utilizzare gli stessi dati per altri scopi. Lo standard OpenEHR ha una strutturazione più adatta alla persistenza dei dati e la loro interrogazione, per cui è più conveniente se applicato in questo ambito.

Capitolo 3

Architettura del sistema informativo sanitario di AUSL della Romagna

3.1 Stato dell'arte delle architetture per sistemi informativi sanitari

I sistemi informativi (S.I.) sono l'insieme dei flussi di informazione gestiti all'interno di un'organizzazione, che acquisisce, elabora, conserva e produce le informazioni di interesse che consentono all'organizzazione il perseguimento dei propri scopi, dove massima importanza è data alle modalità con cui i dati vengono interpretati (cioè trasformati in informazioni) e resi comprensibili, chiari e univoci.

Per quanto riguarda la sanità, il sistema informativo sanitario (S.I.S.) ha la funzione di gestire le informazioni utili alla misura e valutazione dei processi gestionali e clinici, riguarda perciò la rilevazione, elaborazione e diffusione di dati informativi riguardanti la condizione di salute dei pazienti, fattori che determinano stati di malattia e rischio, e aspetti riguardo il funzionamento dei sistemi e aspetti relazionali tra soggetti e sistema sanitario [90].

I primi sistemi informativi sanitari, nati intorno agli anni '80 del secolo scorso, si occupavano principalmente di aspetti amministrativi e contabili delle strutture sanitarie, a questi è seguita l'adozione di sistemi per la registrazione e rendicontazione della movimentazione dei pazienti all'interno della struttura sanitaria (ADT), per registrare le presenze dei pazienti e il transito tra i reparti. La gestione degli accessi ambulatoriali è stata affidata ai sistemi di prenotazione CUP.

Alcune discipline con una naturale predisposizione per l'automazione e ge-

stione dei dati come la radiologia o settori che potevano beneficiare di una integrazione dei processi interni come pronto soccorso e cardiologia, guidarono la nascita dei primi sistemi dipartimentali, sistemi informatici nati per soddisfare esigenze riguardo una specialità confinata (sistemi verticali). Questo ha però causato la frammentazione dei sistemi informativi sanitari, che non ha eguali in nessun altro ambito industriale o di servizi, problematica presente ancora oggi come principale criticità strutturale del settore.

Alcuni approcci a sistemi monolitici sono stati fallimentare in quanto impossibilitati a soddisfare contemporaneamente tutte le discipline del settore in un unico sistema. A queste hanno fatto seguito architetture distribuite con partizionamento tecnico, dove il sistema è composto da diversi servizi, ciascuno dei quali si occupa di una delle problematiche da affrontare. La problematica principale è l'interoperabilità tra questi servizi che non erano più solamente amministrativi e gestionali ma anche clinici, che riguardano il percorso sanitario del paziente.

Per superare questa problematica alcune aziende hanno effettuato investimenti per l'adozione di sistemi come anagrafe contatti 'Master Patient Index' (MPI): un database centralizzato che contiene tutti i dati anagrafici dei pazienti, in modo da ridurre duplicazioni, disallineamenti e incoerenza nei dati [40]; clinical data repository (CDR) ed enterprise service bus (ESB) adottando di fatto architetture SOA per la necessità di integrare dati provenienti da servizi acquisiti in momenti e con tecnologie differenti, da cui la necessità di un orchestratore che permetta lo scambio e integrazione di dati in un sistema distribuito di servizi come questo [48].

Come dichiarato da una recente riforma del ministero riguardo i sistemi informativi sanitari allo stato dell'arte, essendo questi composti da servizi eterogenei, dove applicazioni 'legacy' e moderne devono interoperare, tipicamente si utilizzano SOA con orchestratore centrale ESB e servizi più recenti utilizzano lo standard FHIR per lo scambio dei dati, mentre la maggior parte dei servizi è ancora implementato secondo architetture tradizionali monolitiche con alcuni progetti in atto per effettuare una transizione verso architetture distribuite moderne a microservizi [13].

Ogni sistema che gestisce o tratta informazioni sui pazienti possiede frequentemente, ad esempio, una propria anagrafica, degli archivi sulle codifiche e i dati strutturati, nonché una serie di archivi con i dati che l'applicazione gestisce. In un'Azienda sanitaria sono dunque presenti sistemi diversi, prodotti da diversi fornitori, molto eterogenei per architettura, tecnologie, funzioni. La definizione e l'utilizzo di standard come HL7 FHIR consente lo scambio di messaggi, per trasmettere dati dal sistema dove questi sono originati (produttore) verso altri sistemi che li adoperano (utente). Un esempio sono i messaggi ADT (Accettazione/Dimissione/Trasferimento), che il sistema di accettazione

ospedaliero invia a tutti i sistemi diagnostici e clinici per consentire loro di aggiornare le proprie anagrafiche e agli utenti di selezionare un paziente senza dovere reintrodurre i dati.

Uno dei limiti riscontrati però, che risulta essere comune ad ogni sistema informativo sanitario considerato, riguarda l'approccio alla progettazione dei servizi riguardo l'interoperabilità, che anche nel caso di applicazioni più recenti basate su architetture a servizi e microservizi, continuano comunque ad essere progettati in modo tradizionale, cioè come isole auto-consistenti, che possiedono all'interno del loro database: un'anagrafe pazienti, una serie di tabelle con le codifiche e archivi con dati operativi necessari, dove l'interoperabilità viene vista come problema esterno al servizio, da delegare ad uno strato esterno dedicato ad esso [13].

3.2 Contesto e obiettivi strategici di AUSL della Romagna

L'Azienda Unica Sanitaria Locale della Romagna (AUSL della Romagna) gestisce un territorio esteso e densamente popolato, rendendone la sua gestione un caso di studio interessante a livello nazionale.

Attualmente gli assistiti si aggirano attorno ad un milione e duecentomila pazienti e l'area territoriale è suddivisa in 8 distretti con 7 presidi ospedalieri di cui 4 ospedali maggiori, gestiti da un numero complessivo di circa 17000 dipendenti, tutti tra loro interconnessi mediante un sistema informativo che ha la necessità di tenere conto dei requisiti dettati dalle complessità che derivano da un sistema sanitario regionale diffuso su un vasto territorio eterogeneo come quello della Romagna.

AUSL della Romagna ha un proprio comparto informatico e collabora con più di 20 fornitori esterni che sono leader del settore sanitario informatico a livello nazionale ed internazionale, ed è attualmente impegnata in un percorso di transizione digitale, che mira al miglioramento dell'infrastruttura informatica a supporto dei processi clinici e amministrativi, con l'obiettivo di efficientare l'erogazione dei servizi, agevolare la comunicazione tra le strutture sanitarie e i cittadini, e semplificare i sistemi di accesso alle cure.

Mira a raggiungere una maggiore dematerializzazione, e un conseguente sviluppo in termini di servizi al cittadino e processi di 'connected care, che riguardano la prevenzione, cura e successivo monitoraggio dei pazienti che avvengono prima, durante e dopo l'accesso al sistema sanitario:

- **attività di pre-cura**: analisi e monitoraggio dei pazienti al di fuori della struttura ospedaliera, con lo scopo di prevenire e rilevare anticipatamente l'insorgere di problematiche legate alla salute del paziente.

- **attività di cura**: la fruizione di tutti i servizi sanitari come visite, esami, ricoveri e terapie
- **attività di post-cura**: tutte le attività che il paziente svolge dopo la cura effettuata nel sistema sanitario, presso il proprio domicilo

Il tutto deve essere supportato da un sistema informatico composto da un insieme integrato di tecnologie, sistemi software, dispositivi smart ed indossabili, record di dati e attori che collaborano tra loro, prendendo la forma di un 'ecosistema informatico' in grado di interconnettere tutte queste tecnologie.

3.3 Sistema informativo di AUSL della Romagna

La gestione dei Sistemi Informativi di un'azienda sanitaria è particolarmente complessa, e si compone di diversi sottosistemi interconnessi, spesso eterogenei in quanto acquisiti nel tempo e in fasi differenti dell'espansione delle pratiche digitali nell'azienda da fornitori differenti. Ciononostante i servizi devono essere ovviamente integrati, per dare luogo ad una visione coerente dei dati relativi alle prestazioni effettuate dal servizio sanitario sia dal punto di vista amministrativo che clinico.

In questa analisi ci si concentrerà sul sistema ospedaliero, come cuore della complessità interna alla gestione dei sistemi informativi che riguardano la cura dei pazienti all'interno delle strutture ospedaliere, lo storico delle prestazioni e la gestione amministrativa delle strutture stesse.

Altri moduli del sistema informativo di AUSL Romagna, che non riguardano il sistema informativo ospedaliero, come ad esempio: screening, CUP, vaccinazioni e logistica, non verranno trattate in questa analisi.

Di seguito è riportata una mappa dei moduli principali che racchiudono la complessità intrinseca del sistema ospedaliero e sono riportate le descrizioni dei ruoli che tali sistemi ricoprono e dei flussi di informazione che li collegano nello svolgersi dei diversi processi clinici.

3.3.1 Moduli del sistema informativo ospedaliero

Il sistema ospedaliero allo stato attuale, è composto da diversi moduli, tra cui:

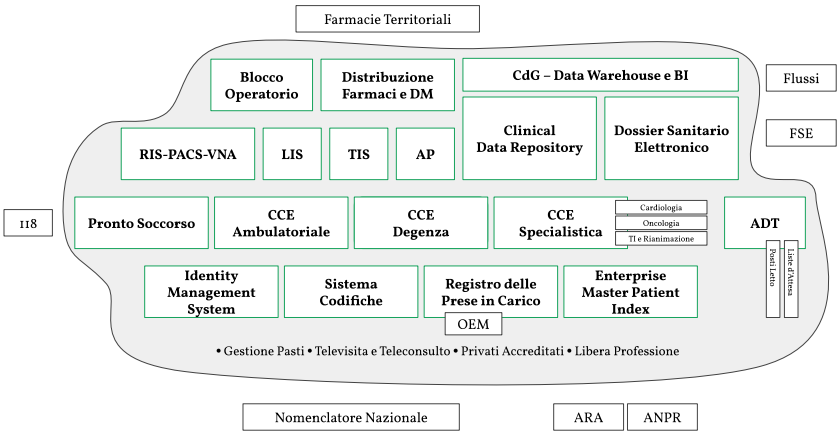

Figura 3.1: Macro elementi del informativo ospedaliero AUSL Romagna

Cartella clinica elettronica

La Cartella Clinica Elettronica (CCE) è il principale collegamento tra i diversi servizi a supporto della gestione ospedaliera. È responsabile del mantenimento dei dati relativi ad un paziente e le prestazioni ricevute all'interno dell'ospedale, indipendentemente dalla durata della permanenza.

Non esiste una unica CCE, ma esistono diversi formati di CCE a seconda del tipo di prestazione erogata, nello specifico:

- **CCE Ambulatoriale**: gestisce le prestazioni ambulatoriali, che sono di durata generalmente breve.
- **CCE Degenza**: gestisce gli accessi ospedalieri, con relative durate e tempistiche, effettuati dai pazienti.
- **CCE Specialistica**: gestisce prestazioni di tipo ambulatoriale o specialistico per contesti tipicamente critici, tra cui cardiologia, oncologia, terapia intensiva e rianimazione.

Pronto Soccorso

Per i pazienti che entrano tramite il percorso emergenziale del pronto soccorso e che poi eventualmente passano negli altri reparti dell'ospedale.

Identity Management System

Riguarda la gestione degli utenti, monitora e permette di effettuare operazioni sui vari sistemi in base all'utente.

Sistema Codifiche

Ogni visita e accesso è regolata a livello nazionale o internazionale e deve essere codificata secondo le regole lì definite.

Ammissione Dimissione Trasferimenti

Il sistema di Ammissione, Dimissione e Trasferimenti (ADT) è il sistema che gestisce l'ingresso e uscita di un paziente nell'ospedale e tra i reparti. Ha visione sui posti letto assegnabili e le liste di attesa. Collegata alla stessa CCE possono esserci diversi passaggi di ADT, se il paziente si sposta tra diversi reparti nel corso della degenza.

Le prese in carico sono legate ad un record ADT, che può averne anche diverse nel corso di una singola ammissione, quando il paziente è ammesso in un reparto, ma allo stesso tempo preso in carico in diversi altri settori dell'ospedale per esami specifici.

Il registro svolge anche la funzione di garantire visibilità temporanea (limitatamente alla presa in carico) dei dati sanitari contenuti nel Clinical Data Repository relativi ad un dato paziente. Questo è essenziale per stabilire la relazione di presa in carico tra paziente e medico, e permettere quindi al medico di visualizzare la storia clinica del paziente quando viene visitato, per una eventuale diagnosi più informata.

Sistemi Specifici per le Prestazioni

Diversi sotto-sistemi riguardano la fornitura di prestazioni specifiche all'interno del sistema ospedaliero. Fanno parte di questo insieme di sotto-sistemi:

- **Radiology Information System (RIS)** per la parte di diagnostica per immagini, combinato con il Picture Archiving and Communication System (PACS). Recentemente sistemi Vendor Neutral Archive (VNA) sono stati adottati per superare i vincoli dei sistemi PACS spesso legati ai formati specifici di determinati macchinari.

- **Laboratory Information System (LIS)** per la richiesta delle analisi di laboratorio e la gestione dei referti.

- **Transfusion Information System (TIS)** per tutto ciò che riguarda le trasfusioni

- **Anatomia Patologica (AP)** per le richieste di analisi istologiche, che riguardano lo studio di tessuti per la rilevazione di problematiche di salute legate ad essi.

Tutti questi sistemi forniscono supporto ai laboratori specifici (ed integrazione con i dispositivi medici utilizzati) e danno la possibilità di condividere il dato digitale sui referti di esami che il paziente può subire durante una degenza.

L'interazione classica parte da una richiesta fatta dalla cartella clinica per un esame, e successivamente la richiesta viene completata con il referto che confluisce nella cartella clinica e poi nel clinical data repository, mentre le copie originali rimangono nei singoli sistemi.

Clinical Data Repository

Rappresenta la base di dati strutturata dove confluiscono tutte le informazioni sulla storia clinica dei pazienti, in modo da mantenere i documenti che sono stati prodotti da tutti gli altri sistemi. La sua visibilità è fortemente controllata al fine di garantire la privacy dei pazienti in base ai consensi definiti.

Dossier Sanitario Elettronico

È una vista, per paziente, sul Clinical Data Repository. Per abilitarlo i pazienti devono dare il consenso all'inserimento dei dati nel dossier, e alla sua condivisione con i medici, che possono consultarlo per investigare la storia clinica del paziente quando hanno una presa in carico per quel paziente.

Fascicolo Sanitario Elettronico

Riguarda tutta la storia clinica del paziente, per i pazienti, che possono condividerlo con i medici di base oppure con quelli specialistici.

Anagrafica Centrale - Enterprise Master Patient Index (eMPI)

L'anagrafica centrale contiene le informazioni anagrafiche di dettaglio di pazienti e dipendenti. Ogni sotto-sistema ha la propria parte di anagrafica locale che è sincronizzata con quella centrale, che viene consultata quando è necessario reperire informazioni specifiche. Ogni anagrafica locale tiene una copia dei dati anagrafici di tutti i pazienti che vedono di volta in volta.

3.4 Analisi dell'architettura allo stato attuale

I diversi moduli appena descritti, possono essere visti con una granularità di un certo tipo, come differenti contesti funzionali che derivano dall'insieme di funzionalità che devono essere garantite dal sistema informativo di AUSL Romagna, devono tra loro comunicare e scambiarsi informazioni, per

cui l'architettura attualmente utilizzata per risolvere questo tipo di problema è l'architettura orientata ai servizi, che fa uso di un ESB per permettere l'interoperabilità tra i vari moduli del sistema informativo (vd. Figura 3.2).

Ciascun modulo internamente utilizza una propria architettura, alcuni di essi utilizzano architetture a microservizi ad eventi seguendo il domain driven design, ed espongono API REST verso l'ESB o sistemi informativi di terze parti, utilizzando standard FHIR ed OpenEHR, altri sono in fase di re-ingegnerizzazione per utilizzare questo tipo di architettura.

3.4.1 Utilizzo degli standard

Attualmente lo standard FHIR è spesso utilizzato solamente come modello per la rappresentazione di concetti e informazioni sanitarie, senza sfruttare appieno le sue potenzialità, soprattutto per quanto riguarda le proprietà non funzionali di:

- **Performance**: in quanto non si utilizza sempre la funzione di riferimenti delle risorse per limitare la dimensione dei payload delle richieste HTTP dei client, rendendo più lento il tempo di risposta alle varie richieste. Il motivo sembra essere la mancanza della possibilità di validazione della latenza impiegata nell'elaborazione di richieste contenenti risorse complete rispetto al caso di risorse FHIR che seguono le regole di conformità e utilizzano collegamenti ad altre risorse richiedendo però un maggior numero di richieste scambiate, per cui si tende ad ipotizzare che seguire le regole di conformità possa essere svantaggioso in termini di performance, ma senza la possibilità, per la mancanza di pattern per la raccolta di metriche e scenari di test, di dimostrare concretamente queste ipotesi, per cui non è attualmente possibile verificare la correttezza ne di una ne dell'altra opzione.

- **Interoperabilità**: tra i servizi dei sistema informativo sanitario, in quanto l'utilizzo di questo standard non sempre segue sempre le regole di conformità viste alla sezione 2.3.1, limitando fortemente l'interoperabilità tra i diversi servizi del sistema informativo, che è invece l'obiettivo di queste regole. Il motivo sembra essere legato alla precedente questione delle performance, in quanto si ipotizza che utilizzare lo standard FHIR solo come modo per rappresentare informazioni sanitarie, senza seguire le regole per l'interoperabilità, possa risultare più performante.

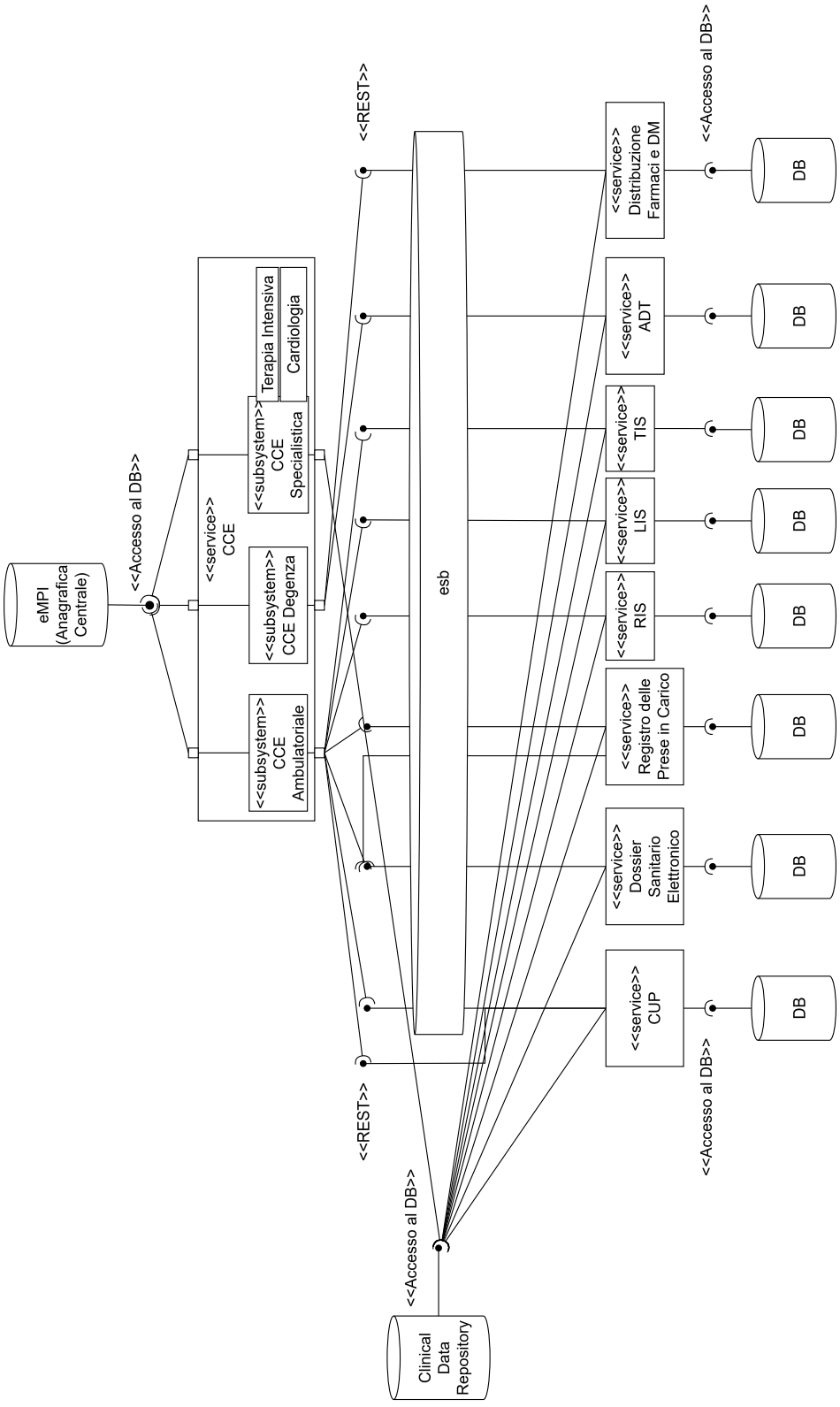

Figura 3.2: Schema C&C dell'architettura SOA attualmente in uso ad AUSL della Romagna

3.4.2 Interoperabilità

Per alcuni servizi è utilizzato un 'anti-corruption layer' (ACL), che dà la possibilità ad ogni sistema di limitare le problematiche dovute alla differente rappresentazione dei dati a causa della dipendenza tra i contesti dei diversi servizi.

Ogni sistema ha necessariamente una propria codifica, utilizzando uno strato di anti-corruzione gli altri possono comunque continuare a comunicare e scambiarsi informazioni, dove possibile. Ma questo non è applicato a tutti i servizi, rendendo più complessa l'interoperabilità con i sistemi legacy.

3.4.3 Monitoraggio e sicurezza

Due ulteriori elementi di monitor e sicurezza, effettuano controlli sull'ESB e verificano rispettivamente il corretto funzionamento di esso in ogni momento, in modo da non diventare il punto critico dell'intero sistema informativo, ed un controllo sugli accessi, i documenti scambiati e il numero di informazioni in transito, per rilevare o meno la presenza di attacchi informatici in atto.

3.4.4 Cartella clinica elettronica

La cartella clinica presenta un'architettura 'modular monolith' con un partizionamento tecnico che suddivide i diversi contesti in tre categorie: Ambulatoriale, Degenza, Specialistica; dal punto di vista architetturale, per quello che è nella versione consolidata del sistema, la CCE è un unico sistema informativo che contiene una parte software più generica di Ambulatoriale e Degenza, più altre cartelle cliniche software ulteriori (facenti parte della Specialistica), come Cardiologia e Terapia Intensiva. Ciascuna di esse ha una ulteriore suddivisione funzionale a seconda del contesto, ma il sistema complessivo fa riferimento ad un unico database condiviso e presenta quindi una architettura a singolo quanto, che può limitare significativamente le performance [17].

Per alcuni casi, come quello di anagrafiche che sarà discusso in seguito, la raccolta delle informazioni relative ai dati anagrafici dei pazienti attraversa la cartella clinica elettronica, in quanto mezzo di comunicazione con il database di 'eMPI' oppure le informazioni sono richieste mediante viste su database.

Parlando in generale dello strumento software di CCE, questo è caratterizzato da una architettura monolitica che utilizza un paradigma MVC dove però non è sempre garantito il disaccoppiamento tra le tre parti di model, view e controller, per cui si creano situazioni in cui sono presenti dipendenze non necessarie tra i componenti che rendono più complessa l'evoluzione e mantenimento del software stesso.

3.4.5 Integrazione tra CCE Ambulatoriale e gli altri servizi

L'integrazione con gli altri sistemi avviene in diversi modi, in questo caso si prende in considerazione la CCE Ambulatoriale, in quanto la CCE di Degenza è equivalente ad essa con poche differenze a livello architetturale che saranno esposte in seguito. L'integrazione avviene:

- **Mediante API**: come la comunicazione tra CCE Ambulatoriale con il modulo di CUP, che è esterno al sistema informativo ospedaliero e permette l'interrogazione riguardo tutta la parte ambulatoriale, in modo da recuperare tutte le informazioni necessarie per la CCE Ambulatoriale mediante chiamate API REST o RESTful con richieste HTTP che scambiano messaggi HL7, le più recenti supportano FHIR, i messaggi scambiati sono diretti tra i due servizi o mediati da un ESB nel caso in cui ci sia necessità di tradurre la codifica dei messaggi in un formato differente.

- **Viste su DB**: alcuni sistemi comunicano con gli altri esponendo viste materializzate che periodicamente sono lette dagli altri servizi in modo da aggiornare i propri dati. Questo crea un accoppiamento più forte tra i servizi, che hanno una maggiore dipendenza tra loro: le viste sui DB possono essere modificate nel tempo e non essere più utilizzabili da alcuni servizi, a differenza delle API che rimangono tali, eliminando dipendenze di questo tipo e permettendo maggiore indipendenza, rendendo il sistema più facilmente manutenibile ed evolvibile nel tempo.

La ragione dietro al fatto che ci siano queste diverse modalità sembra essere principalmente dovuto alla semplicità di implementazione di una vista su un database rispetto all'implementazione di una API, con lo svantaggio che una API permette un disaccoppiamento più forte e una più bassa dipendenza con le chiamate per la richiesta dei dati, in quanto le viste sui DB possono essere modificate nel tempo e non essere più utilizzabili da alcuni servizi, mentre le API rimangono tali, eliminando dipendenze di questo tipo.

3.4.6 Interazioni e scambio di eventi tra CCE Ambulatoriale ed altri servizi

Le interazioni tra i servizi avvengono mediante eventi, messaggi che contengono cambi di stato e che sono notificati al servizio che necessita di conoscerli. Come per il caso precedente, anche qui si prende in considerazione solamente

la CCE Ambulatoriale, in quanto, a livello architetturale, la CCE di Degenza è equivalente ad essa.

Prendendo in considerazione eventi ed interazioni tra CCE Ambulatoriale ed il CUP, nel momento in cui un paziente prende un appuntamento, viene generata una notifica in formato HL7 riguardo la prenotazione di un appuntamento, che è indirizzata verso la CCE mediante ESB. Nel momento in cui il medico modifica lo stato della visita (es: a stato di erogazione), si genera un evento di cambio di stato della prenotazione verso CUP; se la visita viene modificata da parte del paziente, questo lo fa mediante il servizio CUP che quindi genera un cambio di stato nella prenotazione che è raccolto dalla CCE.

Questa tipologia di comunicazioni mediante notifiche di cambiamenti di stato all'interno del dominio dei servizi, è tipicamente adatta ad uno stile architetturale ad eventi, dove diverse componenti del dominio presentano caratteristiche parzialmente sovrapposte ma con una visione differente delle stesse, per cui l'adozione di una architettura puramente ad eventi, anziché a scambio di messaggi contenenti informazioni modellate come eventi, risulterebbe più vantaggiosa.

3.4.7 Dossier Sanitario

Riguardo altri servizi come il 'registro prese in carico', questo ha il compito di monitorare chi è preso in carico e per conto di chi. Quando un paziente effettua, ad esempio, una visita cardiologica, viene emesso un referto che viene inoltrato e mantenuto nel 'Clinical Data Repository', se il paziente torna per una visita cardiologica di controllo e ed è preso in carico da un altro medico, è necessario che anche il secondo medico possa vedere il referto emesso dal primo; questo è valido solamente se riguarda lo stesso reparto o unità operativa, in questo caso 'cardiologia'; se la seconda visita è di un altro reparto, ad esempio ortopedia, questo non vale, per cui il registro unico delle prese in carico (RUPC) deve tenere traccia di chi può vedere quali documenti. Per questo, il registro prese in carico si deve interfacciare al dossier sanitario perché deve tenere conto del consenso del paziente che potrebbe non voler mostrare alcune informazioni a specifiche persone o gruppi di persone, di cui il registro prese in carico deve tenere traccia.

Quando un medico apre la cartella clinica, il sistema effettua una indagine su tutti gli altri servizi e mostra al medico solamente quello che è autorizzato a vedere; la visione o meno di un dato è discriminata dal consenso fornito dal paziente nel proprio dossier sanitario riguardo la consultazione dei propri documenti, se il medico è abilitato alla visione dei dati, il dossier sanitario dà accesso alla CCE a quel documento, e la cartella clinica può recuperare il dato dal servizio che lo contiene (tutti i documenti sono mantenuti sia dal servizio

che lo genera, sia dal CDR, questo perché per un obbligo di legge ogni servizio è responsabile dei documenti che produce).

3.4.8 Anagrafiche locali e centralizzata

Essendo il sistema di cartella clinica composto da una architettura monolitica modulare a singolo quanto, l'unico database utilizzato risulta contenere diverse tabelle di dati che aumentano significativamente il costo computazionale delle query effettuate per il recupero dei dati: ipotesi che però non può essere validata concretamente per l'attuale mancanza di pattern per la raccolta di metriche, attui a identificare concretamente e dettagliatamente le criticità del sistema informatico.

Ogni servizio del sistema informativo che necessita dei dati anagrafici dei pazienti contiene un database di anagrafica locale, in modo che ogni servizio possa continuare a funzionare anche in caso di malfunzionamenti nel servizio della CCE di anagrafica centralizzato (Enterprise Master Patient Index, eMPI), in modo che la CCE non diventi un collo di bottiglia (vd. Figura 3.2).

L'allineamento tra anagrafiche locali e centralizzata avviene mediante uno scambio di richieste ad intervalli di tempo, per cui l'aggiornamento dei dati anagrafici avviene periodicamente, creando situazioni dove i dati anagrafici non sono garantiti essere sempre gli stessi in ogni parte del sistema informativo. Inoltre, l'aggiornamento dei dati anagrafici in un servizio non corrisponde sempre alla generazione di una notifica per l'allineamento di tutte le anagrafiche del sistema, per cui non è sempre garantita l'unicità del dato, con frequenti disallineamenti dove i pazienti possono risultare avere dati anagrafici differenti a seconda del servizio utilizzato.

3.4.9 CCE di Degenza

La parte di CCE per quanto riguarda la degenza funziona allo stesso modo, a livello architetturale, della CCE Ambulatoriale, l'equivalente per la parte di degenza del CUP in questo caso è il sistema di ammissione e dimissione (ADT), dove la comunicazione avviene mediante API che si scambiano messaggi (tipicamente HL7), perché l'equivalente della visita ambulatoriale, è per la degenza il ricovero, che necessita di processi amministrativi riguardo la dimissione, ammissione ed il trasferimento di un paziente tra i vari reparti e ospedali, con gli stessi meccanismi e le stesse interazioni con l'anagrafe che sono state descritte in precedenza nella controparte ambulatoriale.

La differenza significativa tra le due CCE riguarda principalmente le funzioni che offrono, in quanto la degenza deve tenere traccia delle terapie somministrate, e deve comunicare con il servizio di farmacie riguardo eventuali

interazioni che la terapia può avere, tenendo aperta una sessione di ricovero sia per medici che infermieri, entrambe le categorie di professionisti vedono lo stesso ricovero ma da punti di vista molto diversi, dove il ricovero non può essere modellato con lo stesso modello dei dati e le stesse informazioni, perché lo stesso paziente e le stesse informazioni fanno parte di contesti differenti, lo stesso paziente ricoverato è visto dagli infermieri solo come somministrazione della terapia e pratiche cliniche, mentre dal medico come prescrizione e analisi di ciò che sta succedendo nel ricovero, quindi anche il modello di dati è differente.

Effettuando una analisi dell'applicazione della cartella clinica, questa permette gli utenti di selezionare il reparto che si intende interrogare, e per ciascuno di essi è possibile selezionare un paziente e verificare i dati a riguardo mediante un partizionamento funzionale che separa in diverse categorie le possibili attività da monitorare o elaborare, come la funzionalità di 'Terapia', per monitorare l'andamento della terapia dei pazienti nello specifico reparto considerato, 'Diario integrato', per permettere ai medici di prendere nota di alcune osservazioni sui pazienti in cura, 'Cartella infermieristica' per il monitoraggio di quello che sta succedendo al paziente, ed altre ancora.

Come per il caso precedente si segnala la criticità riguardo le performance, la cui causa è però di difficile individuazione per la mancanza di pattern per la raccolta di metriche, in grado di identificare e quantificare concretamente le criticità del sistema.

3.4.10 Enterprise Service Bus

Una criticità dell'attuale sistema informativo di AUSL della Romagna è il punto di centralizzazione dovuto dal ESB: questo è il mezzo di comunicazione tra tutti i servizi, per cui può rivelarsi collo di bottiglia dell'intero sistema informativo se non sono adottati specifici accorgimenti.

Attualmente questo presenta un'architettura distribuita dove i diversi canali sono servizi che svolgono la funzione atomica di conversione di formato dei dati per permettere l'interoperabilità tra servizi moderni e legacy, dove però i singoli canali possono rivelarsi punto critico per la comunicazione tra i servizi, in quanto ogni comunicazione verso il servizio di cui il canale è responsabile, deve attraversare questo, per cui ciascuno di essi può diventare punto di centralizzazione per le comunicazioni con i servizi di cui è il gestore. Inoltre, essendo tutte le comunicazioni tra i servizi mediate dal ESB, eventuali carichi sostenuti possono presentare rallentamenti significativi, per cui può rivelarsi problematico per le comunicazioni tra i servizi del sistema.

Una soluzione attualmente applicata da alcuni servizi, allo scopo di evitare rallentamenti dovuti al forte carico che può interessare l'ESB, è quella

di utilizzare una comunicazione diretta tra i servizi, evitando completamente quest'ultimo, soluzione che rende però la sua presenza, di fatto, non necessaria, e che aumenta piuttosto la complessità del sistema, rendendo più difficile tracciare le comunicazioni tra i servizi, con impatti significativi sull'evolvibilità del sistema.

Una possibile soluzione per il caso di studio specifico preso in considerazione con questo elaborato, è descritta al Capitolo 4, dove si è discussa una soluzione generale che prevede l'utilizzo di un'architettura distribuita ad eventi che possa garantire una maggiore robustezza e disponibilità alle funzionalità del ESB, rendendolo inoltre unica fonte di verità dei messaggi scambiati dal sistema pur mantenendo una architettura distribuita.

3.4.11 Identificazione delle criticità e validazione dell'architettura attuale e future

La principale criticità rilevata nel caso della cartella clinica elettronica, è l'assenza di utilizzo di pattern per la raccolta di metriche, che sono essenziali per la rilevazione e concretizzazione di criticità nel sistema. Per cui risulta complesso, se non impossibile, l'identificazione delle cause delle problematiche relative al servizio considerato, rendendo di fatto infattibile l'identificazione di soluzioni architetturali concrete che possano garantire il superamento delle principali problematiche del sistema, limitando fortemente ogni tipo di intervento che possa essere effettuato per la risoluzione delle criticità identificate da questa analisi e rendendo complessa l'evoluzione del servizio stesso.

Capitolo 4

Progettazione di un prototipo architetturale esemplificativo per la CCE e sistemi correlati di AUSL della Romagna

Lo scopo di questo elaborato è quello di fornire un esempio di progettazione di una architettura per la cartella clinica elettronica di AUSL della Romagna che sia in linea con i requisiti non-funzionali generali forniti e mostri come superare le criticità messe in luce alla sezione 3.4.

Non essendo in questo momento possibile la raccolta di metriche per la validazione dell'architettura attuale ed eventuali proposte architetturali future (dal punto di vista delle proprietà non-funzionali), il focus di questo elaborato è stato quello di selezionare uno scenario rappresentativo della cartella clinica in modo che possa essere preso d'esempio per mostrare come applicare pattern per la raccolta di metriche, in modo da identificare e quantificare concretamente i punti critici dell'architettura per le future validazioni.

Il risultato è l'implementazione di pattern per misurazione di attributi di qualità, ricavati mediante test automatizzati che ricreano scenari verosimili, che permettano di quantificare il grado con cui sono rispettati i requisiti non-funzionali forniti, oltre alla possibilità di inserire limiti e vincoli da rispettare.

4.1 Selezione degli scenari

Essendo il sistema informativo sanitario di AUSL della Romagna complesso ed eterogeneo, non è stato possibile considerarlo nella sua interezza, per cui è stato scelto come scenario rappresentativo una porzione della cartella

clinica elettronica che fosse sufficientemente generale, in modo da poter essere studiato e analizzato per permettere di applicare gli stessi ragionamenti e conclusioni anche per altri servizi della cartella clinica, declinandoli al caso specifico considerato.

Lo scopo di questo elaborato è quello di mostrare un esempio di architettura per la cartella clinica elettronica che rispetti i requisiti non-funzionali generali forniti e le linee guida per l'applicazione di pattern per la raccolta di metriche (vd. sottosezione 4.3.3) per fornire una base a progettazioni future con cui validare le proprie proposte architetturali ed identificare concretamente i punti critici dell'architettura attuale, per permettere la progettazione di soluzioni architetturali specifiche e poter dimostrare concretamente che siano migliorative a livello di tutte le proprietà non-funzionali richieste.

4.2 Analisi

Come descritto nella sezione 3.4 riguardo l'analisi e criticità dell'attuale sistema informativo sanitario di AUSL della Romagna, la cartella clinica presenta una suddivisione in tre categorie: Ambulatoriale, Degenza, Specialistica; dal punto di vista architetturale, per quello che è nella versione consolidata del sistema, la CCE è un unico sistema informativo che contiene una parte software più generica di Ambulatoriale e Degenza, più altre cartelle cliniche software ulteriori come Cardiologia e Terapia Intensiva.

Per valutare la soluzione da proporre, è necessario conoscere i requisiti che devono essere rispettati, soprattutto non-funzionali, riguardo proprietà che permettono ad esempio una maggiore resilienza, scalabilità e robustezza che l'architettura proposta deve rispettare.

4.2.1 Requisiti funzionali

Non essendo scopo di questo elaborato la definizione di nuove applicazioni e funzionalità per il sistema informativo di AUSL della Romagna, non ci si è concentrati sull'aspetto dei requisiti funzionali, che rimangono quelli già garantiti dai servizi presi in considerazione in questo documento.

4.2.2 Requisiti non-funzionali

Il nuovo design architetturale per la cartella clinica elettronica deve essere in grado di garantire maggiormente, rispetto alla situazione attuale, le seguenti proprietà non-funzionali:

- **Performance**: attualmente la CCE presenta diversi rallentamenti dovuti principalmente al fatto che sia composta da una architettura monolitica modulare con singolo database comune a tutti i moduli, in cui query SQL complesse composte da diversi join impattano significativamente sulle performance;

- **Basso accoppiamento**: è richiesto maggiore isolamento tra i servizi, in modo da permettere l'utilizzo delle singole funzionalità della cartella clinica elettronica in modo indipendente le une dalle altre;

- **Robustezza**: è richiesto che parti specifiche della cartella clinica possano rispettare requisiti non funzionali specifici, come le funzioni relative alla terapia, che ci si aspetta debbano subire carichi di lavoro più alti rispetto ad altre funzionalità, per cui è richiesta un'alta disponibilità, maggiore rispetto ad altre funzionalità dello stesso sistema; per cui è richiesta una soluzione che possa garantire proprietà non-funzionali specifiche a seconda del contesto.

Si aggiungono inoltre ulteriori necessità:

- è richiesta la proposta di soluzione per il tracciamento delle metriche, validazione e identificazione delle criticità, del servizio di cartella clinica elettronica, dal punto di vista architetturale;

- è richiesta una proposta di soluzione, a livello architetturale, che permetta di tracciare ed analizzare le interazioni degli utenti con la cartella clinica elettronica;

4.2.3 Quality Attributes

Non essendo attualmente disponibile un sistema per la raccolta delle metriche, non è stato possibile quantificare il grado con cui sono garantite le varie proprietà non funzionali fornite, per cui non è possibile effettuare valutazioni in merito.

Per questo motivo, l'obiettivo di questo elaborato è diventato quello di fornire un esempio su come ottenere questo tipo di misurazioni, piuttosto che proporre una soluzione architetturale ad-hoc per questo caso di studio; in quanto le informazioni attualmente consultabili risultano essere di carattere generale e non adatte ad una soluzione concreta che possa effettivamente essere migliorativa.

A seguito di questo elaborato sarà possibile misurare attributi di qualità concreti, che mostrino il grado con cui sono attualmente rispettate le

proprietà non-funzionali dell'architettura attuale, inoltre, gli stessi test automatizzati possono essere usati per la validazione e comparazione di proposte architetturali future (vd. sottosezione 4.3.3).

4.2.4 Dominio

Essendo la Cartella Clinica Elettronica di AUSL della Romagna composta da diverse funzionalità, presenta un dominio complesso, composto principalmente dalle tre macro categorie di 'CCE Ambulatoriale', 'CCE di Degenza', e 'CCE Specialistica', ciascuna di esse presenta a sua volta una suddivisione funzionale in base alle attività che devono essere svolte per ciascuna di esse.

A titolo esemplificativo, si riporta una parte di dominio della cartella clinica elettronica che verrà presa in considerazione per questo progetto riguardo terapia, diario clinico e anamnesi pregressa, scelti apposta in quanto sufficientemente rappresentativi degli scenari che possono essere rilevati in questo ambito.

Per quanto riguarda l'interazione tra la cartella clinica elettronica e servizi terzi, si è presa in considerazione la questione che riguarda i dati anagrafici dei pazienti, che sono contenuti nel 'eMPI', con cui attualmente comunica la cartella clinica elettronica, per cui i servizi terzi devono appoggiarsi ad essa; ed il 'dossier sanitario', che contiene documenti relativi alle informazioni del paziente riguardo la sua storia clinica, informazioni generate durante eventi sanitari come ricoveri, visite specialistiche e accessi al pronto soccorso.

Nonostante questo sia un unico sistema e il dominio presenti caratteristiche comuni, a seconda del contesto a cui deve essere applicato, il ruolo e modello di dati degli stessi oggetti del dominio, ha caratteristiche differenti, per cui un partizionamento in diversi contesti renderebbe più modulare il sistema, creando una specializzazione della cartella clinica sulla base dell'applicazione che deve essere effettuata, utilizzando un modello di dati specifico a seconda del contesto in cui si trovano; a differenza dell'architettura monolitica attuale che utilizza lo stesso modello di dati per tutti i diversi contesti in cui può essere applicata, senza effettuare disaccoppiamento tra i le diverse casistiche in cui può essere utilizzata, che è invece richiesto da applicazioni con una logica 'patient centered', dove la logica del sistema ruota attorno alle specifiche esigenze del paziente a seconda del contesto sanitario in cui si trova.

Considerando lo scenario rappresentativo considerato in questa analisi, si sono rilevati diversi partizionamenti che separano le informazioni relative al paziente a seconda del tipo di contesto, identificando funzionalità atomiche di 'terapia', 'diario clinico' e 'anamnesi pregressa' dove, secondo la logica 'patient centered', ciascuno di questi servizi conterrà informazioni relative al paziente, ma da un punto di vista differente a seconda del contesto in cui si trova, in

modo che ciascun servizio abbia una visione sul paziente incentrata solo sul contesto a cui appartiene:

- **Diario Clinico**: contesto che rappresenta una funzionalità per prendere note sui pazienti di volta in volta che fanno visite e danno indicazioni su come sta andando il ricovero e lo stato del paziente, ogni volta che deve essere somministrato un farmaco o definita una terapia è richiesto che venga effettuato un controllo per accertare l'assenza di allergie o conflitti con patologie pregresse. Inoltre, a seguito di un generico intervento, questo deve essere notificato al servizio di anamnesi in quanto potrebbe rendere non più compatibile la somministrazione farmacologica attualmente prescritta in terapia. Questo può essere ulteriormente partizionato in due servizi di diario medico e infermieristico.

- **Terapia**: informazioni relative al paziente, riguardo la definizione e aggiornamento della terapia per il paziente, ogni volta che deve essere effettuata una somministrazione di un farmaco, deve essere verificato se possa essere in conflitto con alcune allergie o patologie del paziente.

- **Anamnesi pregressa**: informazioni relative al paziente, riguardo eventuali allergie o patologie recenti, nel contesto attuale. Verifica l'eventuale presenza di incompatibilità con i farmaci prescritti sul diario e terapia.

Inoltre, sempre considerando la precedente analisi del dominio per quanto riguarda le interazioni tra cartella clinica elettronica e servizi terzi, la cartella clinica elettronica deve poter accedere a tutte le informazioni relative alla storia clinica del paziente, per verificare eventuali incompatibilità dei trattamenti di terapia con le patologie del paziente, oltre ai dati anagrafici, necessari per il rintracciamento del paziente:

- **Dossier Sanitario**: contenente le informazioni sul paziente relative alla lettura di informazioni riguardo la storia pregressa del paziente, come le patologie, deve poter comunicare alla cartella clinica elettronica eventuali nuove patologie per la sospensione delle terapie già pianificate non più compatibili con le nuove patologie identificate. Inoltre, tutte le informazioni lette devono essere verificate per controllare di essere in linea con i requisiti sulla privacy dettati dal paziente prima di essere letti dal personale;

- **Dati anagrafici**: informazioni sul paziente relative ai dati anagrafici, come aggiunta o modifica di informazioni anagrafiche. Eventuali aggiornamenti dei dati anagrafici devono essere notificati a tutti i servizi terzi, in modo da essere sempre aggiornati all'ultima versione dei dati possibile.

4.3 Design

Le informazioni riportate in questa sezione sono solo a livello architetturale, per cui sono agnostiche riguardo implementazioni e tecnologie specifiche. Per altre informazioni più specifiche riguardo l'implementazione proposta per questo caso di studi, si rimanda alla sezione 4.4.

Di seguito si riporta l'esempio architetturale proposto, da considerare come scenario rappresentativo che possa essere utilizzato come riferimento per l'effettiva progettazione di architetture per il caso specifico di AUSL della Romagna. Non essendo possibile effettuare una raccolta di attributi di qualità dall'architettura attuale, queste conclusioni sono da considerarsi come linee guida per una futura implementazione concreta ed efficace di architetture che sono generalmente valide per garantire i requisiti non-funzionali generali forniti in questo caso di studio; per cui deve essere considerata solo come esempio da cui declinare soluzioni concrete e validi per questo caso di studio concreto.

Data l'analisi effettuata in precedenza e le informazioni attualmente disponibili, si propone una soluzione architetturale distribuita e ad eventi per il caso della cartella clinica elettronica e sistemi correlati.

4.3.1 Architettura per la Cartella Clinica Elettronica

La CCE potrebbe essere utilizzata come unico sistema, se sufficientemente modulare dal punto di vista architetturale, perché necessita di caratteristiche che permettano alle funzionalità della cartella clinica di essere il più possibile indipendenti tra loro, in modo da ridurre l'accoppiamento.

Per questo motivo, per quanto riguarda l'architettura per la Cartella Clinica Elettronica, si propone un'architettura a microservizi, che separi le diverse funzionalità della cartella in più microservizi tra loro il più possibile indipendenti, in modo da aumentare il disaccoppiamento tra i vari servizi e permettere l'utilizzo di ciascuna funzione in modo indipendente dalle altre.

Una transizione da una architettura monolitica ad una distribuita a microservizi che separi nettamente diversi aspetti, è in linea con la proprietà nonfunzionale di garantire che le diverse funzionalità in cui è attualmente suddivisa la cartella clinica possano continuare a funzionare in maniera indipendente l'una dalle altre, quindi fare in modo che, ciascuna di queste funzionalità possa continuare ad eseguire nonostante eventuali lavori di manutenzione o malfunzionamenti riguardanti alcuni servizi specifici, senza dipendere dagli altri, in modo da ridurre l'accoppiamento tra i vari servizi e permettere che ciascuno di essi possa continuare a funzionare autonomamente. Per questo motivo si propone un partizionamento, che isoli il più possibile i vari contesti del dominio

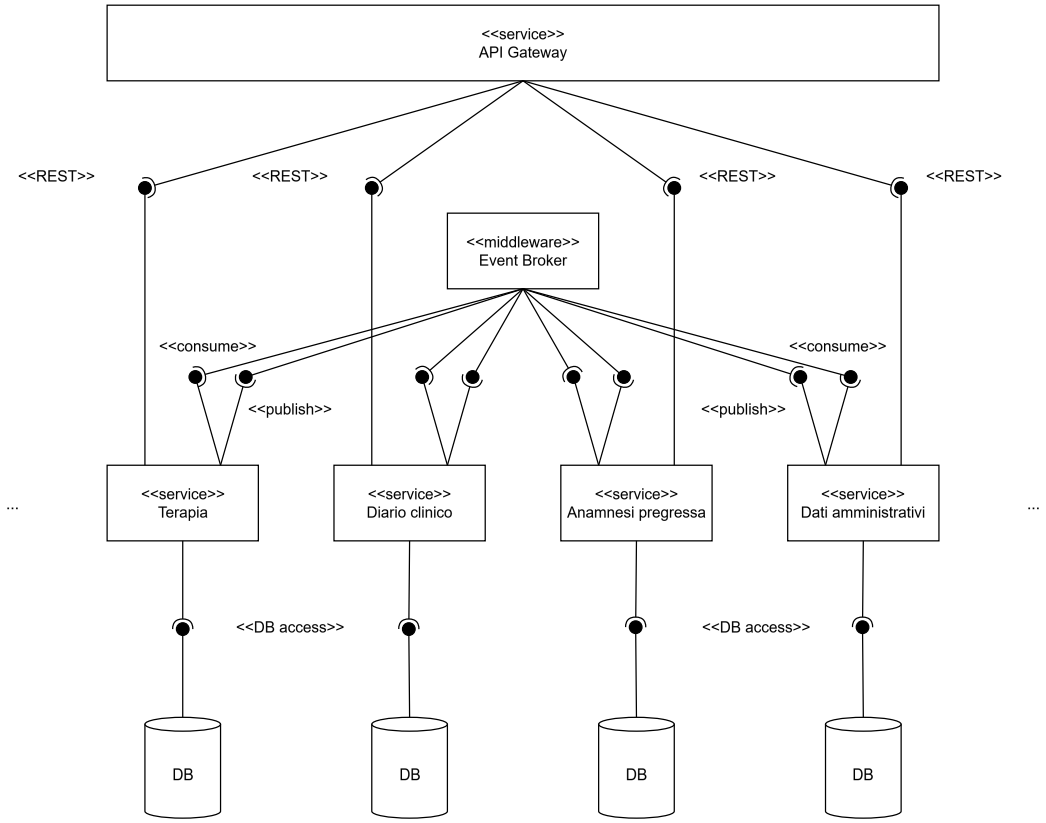

Figura 4.1: Schema C&C di soluzione, esemplificativo, per la CCE

della cartella clinica, come quello mostrato in Figura 4.1, dove ciascuno delle attuali suddivisioni farà parte di un microservizio a sé stante.

Inoltre, un'architettura a microservizi rispetterebbe il requisito non funzionale di permettere che ciascun servizio della cartella clinica possa avere una progettazione che garantisca requisiti non funzionali specifici, validi solamente per il servizio considerato. Permettendo di intervenire in maniera indipendente su proprietà come 'performance' e 'scalabilità', entrambe proprietà che possono essere facilmente raggiunte applicando specifici pattern e utilizzando, a livello di deployment, meccanismi di scaling orizzontale automatizzato, proprietà che è perfettamente garantita da architetture a microservizi, a rispetto di altre architetture che non sono in grado di garantire con la stessa efficacia.

Per quanto riguarda il requisito della maggiori prestazioni, non è stato possibile identificare e quantificare la fonte delle criticità, dati che possono essere raccolti mediante pattern di metriche e validazione mediante scenari; per cui si consiglia l'applicazione di pattern per miglioramento delle prestazioni, come CQRS [74], che però sono efficaci solamente se è rilevata una forte asimmetria tra il numero di richieste di lettura e scrittura, ed è tollerata una consistenza

di tipo 'eventual'.

Visti i requisiti non funzionali attuali per la CCE, le considerazioni effettuate nelle precedenti sezioni e una analisi della letteratura dove si prendevano in considerazione tematiche relative alle architetture software e come sono state utilizzate in aziende con caratteristiche simili ad AUSL Romagna, una architettura a microservizi potrebbe in grado di superare le maggiori limitazioni transizioni da architetture monolitiche a microservizi per server FHIR hanno mostrato netti miglioramenti in termini di proprietà non-funzionali, in linea con il requisito di maggiore autonomia tra le funzionalità della CCE fornito per questo caso di studio [67] [82] [8] [7] [81].

In architetture a microservizi come quella qui proposta a titolo esemplificativo (vd. Figura 4.1), ciascun servizio è il più possibile indipendente dagli altri, ha una singola responsabilità e svolge una sola funzionalità atomica, per cui è garantita la sua corretta funzione anche in caso di mancato funzionamento di altri servizi: ciascun servizio ha una propria base informativa, necessaria a garantire il proprio funzionamento, e la comunicazione con altri servizi per il reperimento di ulteriori informazioni è effettuato mediante scambio di messaggi (od eventi, nel caso di architetture ad eventi) e ciascun microservizio espone una API, tipicamente REST secondo regole RESTful, con cui potersi interfacciare con l'esterno (vd. sottosezione 1.2.2 e sottosezione 1.2.3).

Per questo caso di studio specifico, dopo una analisi del dominio (vd. sottosezione 4.2.4), si riporta una soluzione di architettura a microservizi ad eventi per cui ciascun microservizio ha una propria base informativa che garantisce il proprio corretto funzionamento in modo indipendente dagli altri, e gli eventi generati da altri servizi sono raccolti da ogni microservizio richiedente in modo da aggiornare la propria base informativa ed eventualmente eseguire le ulteriori operazioni necessarie. Questo avviene mediante Event Broker, che scambia eventi tra i microservizi, seguendo le regole per l'interoperabilità in vigore per lo standard considerato, in modo che ciascuno di essi sia sempre aggiornato sui dati da cui dipende dagli altri per operare in maniera più autonoma possibile.

La scelta di utilizzare un'architettura ad eventi è dovuta al requisito non funzionale di garantire basso accoppiamento e quindi maggiore autonomia dei servizi, in modo che ciascuno di essi possa continuare ad eseguire le proprie funzioni in maniera il più possibile indipendente dagli altri, e architetture ad eventi permettono un maggior livello di indipendenza, mediante comunicazioni asincrone 'publish-subscribe' di eventi rispetto a comunicazioni di tipo 'request-response' a scambio di messaggi: dove l'elaborazione delle informazioni può essere richiesta a servizi esterni per cui si forma un maggiore livello di accoppiamento.

Inoltre, l'idea dell'utilizzo di un'architettura ad eventi è rafforzata dal precedente studio sul dominio, in cui si poteva individuare la possibilità di au-

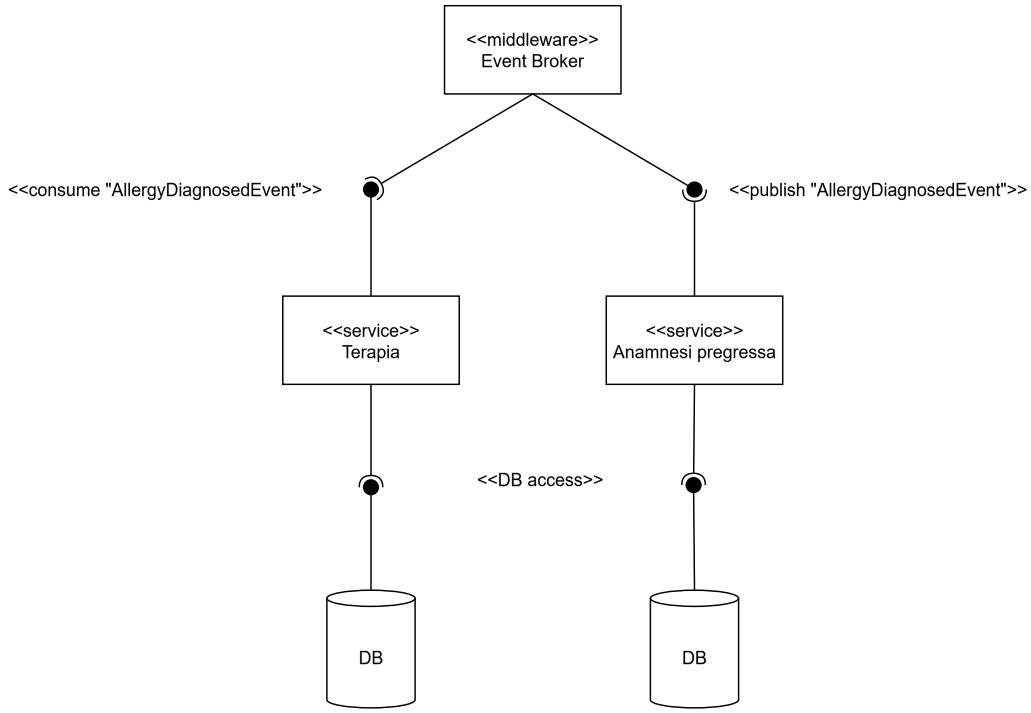

Figura 4.2: Schema C&C per la CCE specifico riguardo gli eventi per la sospensione automatizzata delle terapie

tomatizzare il processo di identificazione di allergie ed incompatibilità con i farmaci, a seguito dell'aggiornamento di esse da parte del servizio di anamnesi e la sospensione immediata ed automatizzata delle terapie già pianificate nel caso di rilevazione di nuove patologie identificate dal dossier sanitario per il paziente (vd. Figura 4.2). Possibilità che può essere garantita da architetture ad eventi, dove lo scopo è quello di poter garantire un'elaborazione delle informazioni in tempo reale e in modo completamente asincrono, a fronte di ogni cambiamento degli elementi di dominio (vd. sottosezione 1.2.3). Il tutto seguendo una logica 'patient centered' dove ogni servizio può essere notificato di eventuali cambi di stato del paziente riguardo i contesti clinici considerati.

Esponendo tutti i servizi una API REST, rimane sempre possibile lo scambio di informazioni di tipo 'richiesta-risposta' tra i servizi, per cui lo scambio di eventi asincrono rende questi reattivi, automatizzando alcuni processi che attualmente vengono effettuati in maniera manuale, ma rimane sempre possibile la richiesta di informazioni tra servizi per la raccolta di dati contenuti in servizi terzi.

Per quanto riguarda le altre funzionalità della cartella clinica elettronica è possibile ripetere gli stessi ragionamenti qui effettuati e applicarli allo stesso modo a ciascuna di esse, creando per ognuna un servizio, e seguendo gli stessi ragionamenti che hanno portato alla creazione di questi ma declinandoli al contesto specifico.

4.3.2 Integrazione tra CCE e servizi esterni

Riguardo l'integrazione tra CCE e i servizi esterni, si propone una architettura ad eventi in cui i cambi di stato all'interno di un sistema generano eventi che sono raccolti dal ESB e inoltrati mediante i rispettivi canali verso il servizio richiedente.

Attualmente la comunicazione avviene per scambio di messaggi che notificano i cambiamenti di stato che avvengono tra i vari servizi (come se fossero eventi), per cui si propone l'utilizzo di una architettura ad eventi (EDA, sottosezione 1.2.3) in cui cambi di stato che avvengono già attualmente all'interno della cartella clinica possono essere raccolti, mediante comunicazioni di tipo publish-subscribe, dal middleware e inviato attraverso i canali utilizzati per le comunicazioni tra servizi esterni ed inoltrati verso i sottoscritti.

Questo tipo di soluzione è rafforzata dal precedente studio sul dominio dove si è identificata la possibilità di rappresentare come eventi i diversi cambi di stato degli elementi che riguardavano lo scenario esemplificativo di anagrafiche e dossier sanitario, dove la comunicazione tra la cartella clinica elettronica e i sistemi correlati viene utilizzata per lo scambio di informazioni sul cambiamento di stato dei dati anagrafici e patologie, rispettivamente.

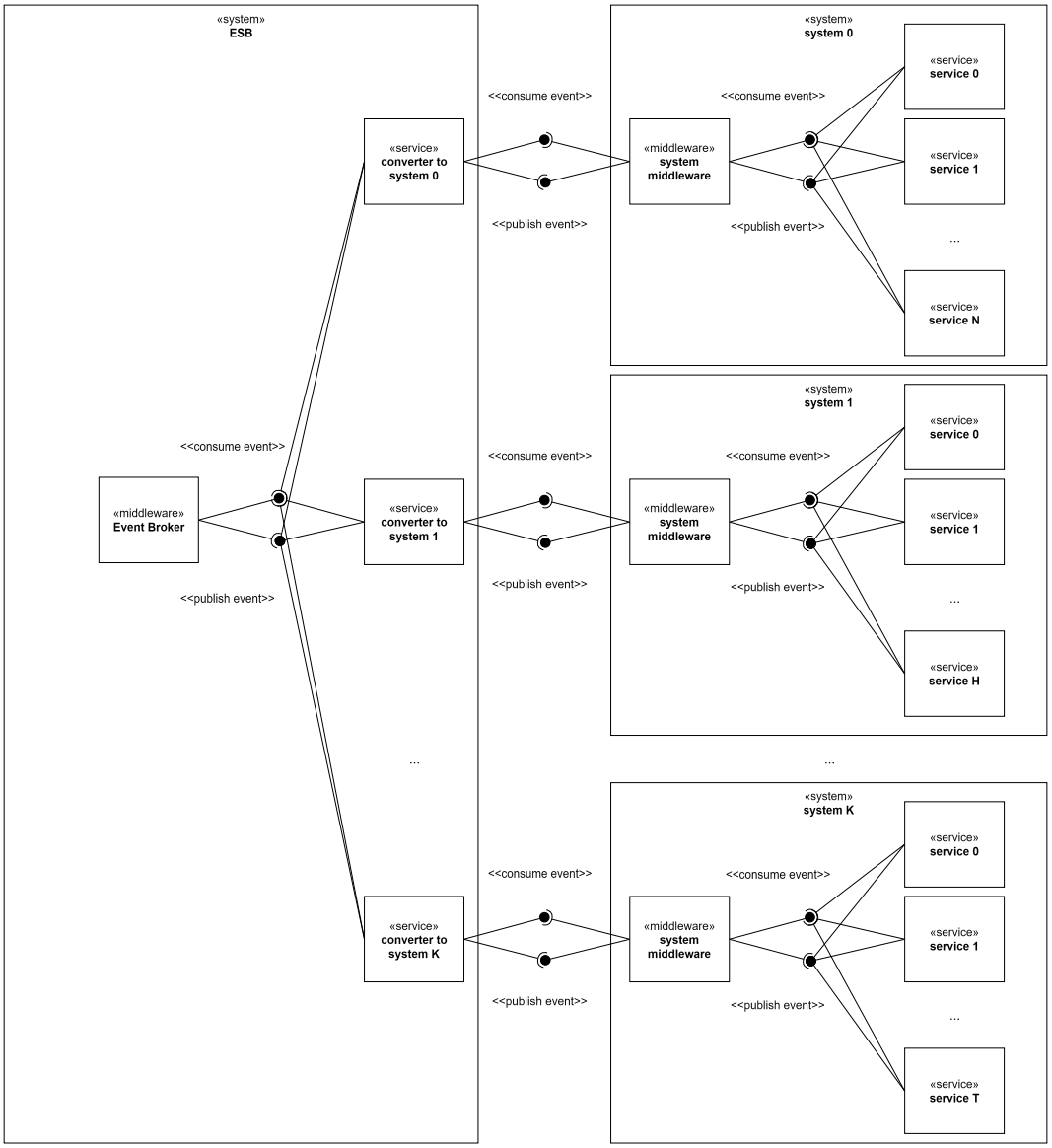

Figura 4.3: Schema C&C di soluzione per l'integrazione tra ESB e servizi terzi

Per utilizzare questo tipo di architetture evitando disallineamenti e duplicazioni dei dati scambiati tramite eventi e raccolti nei database dei servizi considerati, è necessario definire una unica fonte di verità, cioè un database che possa mantenere traccia di tutti gli eventi scambiati e permettere l'eventuale riallineamento con i servizi. Inoltre, per questo caso di studi, essendo il sistema di AUSL della Romagna complesso ed eterogeneo, è necessario garantire l'interoperabilità tra i diversi sistemi.

Per mantenere l'interoperabilità tra i servizi già presenti e rendere il ESB l'unica fonte di verità al posto del 'eMPI', si suggerisce una architettura per ESB decentralizzata in cui si utilizza un middleware di tipo 'Event Broker' decentralizzato e i vari servizi che effettuano la traduzione da un formato ad un altro (già presenti nel ESB), consumerebbero mediante sottoscrizione gli eventi generati dal broker in modo da inoltrarli al sistema a cui sono associati e convertiti nel formato atteso. Prendendo nello specifico il caso dei disallineamenti tra le anagrafiche dei vari servizi, sarebbe possibile riallinearli richiedendo tutti gli eventi di creazione o aggiornamento di esse che sono stati scambiati mediante event broker da uno specifico istante di tempo in poi, in modo da rendere il 'event store' l'unica fonte di verità e garantire la consistenza dei dati tra i diversi servizi [6], questo è mostrato in Figura 4.3) dove è riportato un generico design per l'interoperabilità tra servizi. In questo modo è possibile ridurre l'effetto di collo di bottiglia che in attualmente si crea nel servizio di cartella clinica, che è l'unico servizio che può accedere alle informazioni contenute nel 'eMPI', in quanto tutte le comunicazioni per quanto riguarda le informazioni anagrafiche sono contenute in una differente fonte di verità, che non è più accessibile solamente via cartella clinica elettronica o viste su database, ma da qualunque servizio necessiti di queste informazioni, comunicandole al broker.

Un approccio mediante 'event broker' decentralizzato e microservizi per i canali responsabili di effettuare conversioni di formato, permette una maggiore disponibilità del servizio di ESB, evitando centralizzazioni che lo rendono collo di bottiglia dell'intero sistema informativo, in quanto l'utilizzo di repliche per le partizioni dei dati contenuti nel 'event broker' garantisce alta disponibilità e robustezza, mentre l'utilizzo di microservizi, contenenti funzionalità atomiche di conversione di formato dei dati abilita la possibilità di scalare orizzontalmente, per permettere alta disponibilità e resilienza anche a fronte di un grande numero di richieste, evitando rallentamenti che, dato il ruolo del ESB di mediatore tra tutti i servizi del sistema informativo, impatterebbero sull'interezza del sistema.

Integrazione specifica tra CCE e gli altri servizi

Considerando quanto mostrato precedentemente in maniera più generale alla Figura 4.3 e Figura 4.1, l'interazione nello specifico tra cartella clinica elettronica e il ESB avverrebbe come riportato in Figura 4.4, dove le comunicazioni tra i microservizi della cartella clinica avverrebbero mediante broker, e quelle con gli altri sistemi mediante ESB attraverso il servizio di conversione apposito per il sistema di cartella clinica e mediante sistema di trasmissione dati REST via API Gateway per l'autenticazione.

Ciascun microservizio è un server autonomo per cui i dati contenuti in ciascun database sono necessari al solo funzionamento del microservizio stesso, la comunicazione tra essi avviene mediante broker, mentre le comunicazioni con sistemi esterni o l'utente avviene mediante ESB ed API REST.

4.3.3 Pattern per la raccolta di metriche e validazione delle architetture

I pattern per la raccolta di metriche sono utilizzati per la rilevazione di dati tra cui metriche a livello infrastrutturale, come la CPU, memoria, utilizzo del disco, fino ad applicazioni ad un livello di astrazione più alto, come il numero di richieste ricevute dai servizi, la latenza in risposta ad esse, il tipo di richieste ricevute (lettura o scrittura) e sono raccolte da un servizio di metriche che elabora le informazioni ricevute e fornisce degli strumenti per la visualizzazione in tempo reale dell'evoluzione del sistema e notifica di eventuali criticità.

Il modello riportato in Figura 4.5 è definito 'pull model', ed è un modello dove le metriche sono raccolte dai singoli servizi, che espongono una API, utilizzata dal servizio di elaborazione delle metriche per la raccolta e analisi periodica delle informazioni rilevate dai singoli servizi.

Le metriche sono dati identificabili attraverso un 'nome' che rappresenta il tipo di informazione che si sta tracciando, un 'valore' che indica la quantità misurata e l'istante di tempo in cui la misurazione è stata effettuata.

Una volta raccolte possono essere richieste dai client, e a seconda della tecnologia considerata, si possono visualizzare grafici ed effettuare query che monitorano i dati in tempo reale, fornendo anche eventuali funzionalità di notifica di eventi critici.

Questi pattern possono anche essere utilizzati per la validazione delle architetture software, per la rilevazione e concretizzazione di attributi di qualità, ricreando scenari di test automatici e verosimili con cui monitorare il comportamento dell'architettura a fronte di uno stimolo esterno.

Sono disponibili diversi tipi di scenari per la rilevazione di attributi di qualità, tra cui scenari per la validazione della 'disponibilità', 'sicurezza', 'u-

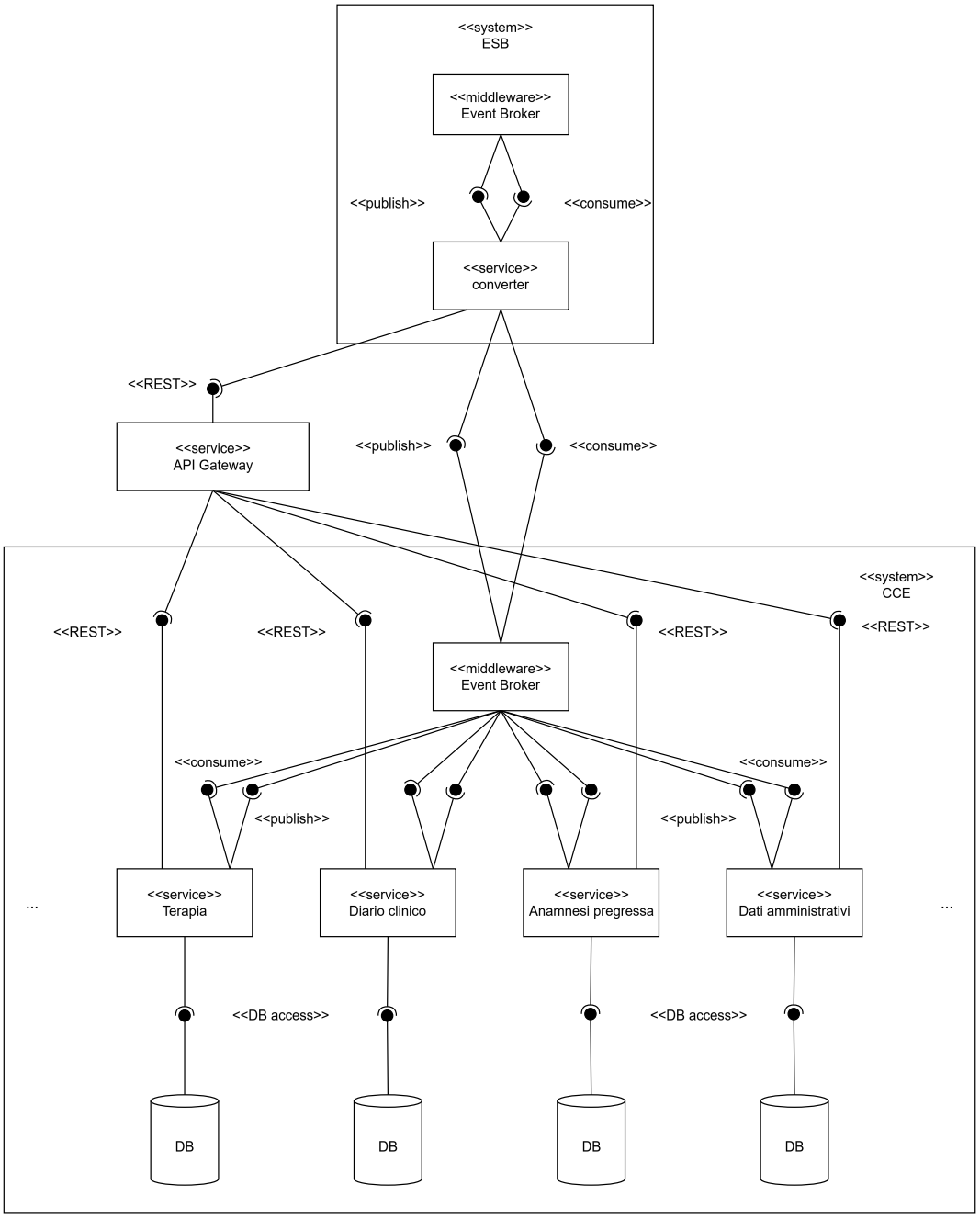

Figura 4.4: Schema C&C di soluzione specifico tra CCE e ESB

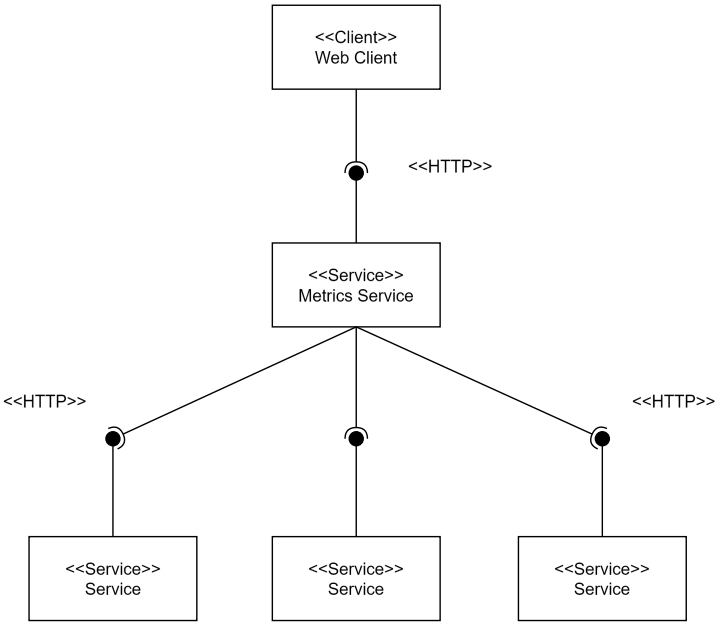

Figura 4.5: Schema C&C di pattern per la raccolta di metriche

sabilità' e 'performance' [5], ma a livello generale, gli step per la creazione degli scenari prevedono l'identificazione di una 'sorgente', fonte dello 'stimolo' che è raccolto dal componente o sistema considerato per il test, che causa una 'risposta' misurabile e visualizzabile mediante metriche. In questo modo è possibile verificare punti critici dell'architettura in maniera precisa, concretizzabile e ripetibile. I risultati così ottenuti possono essere utilizzati per validare e comparare le proposte architetturali con quelle attuali, a livello di attributi di qualità garantiti. Inoltre, possono essere anche definiti limiti e vincoli sulla quantità tollerabile dei valori misurati.

Un esempio di utilizzo è quello della valutazione delle prestazioni, in cui si può creare uno scenario che simula una situazione reale con cui valutare il comportamento del sistema a fronte di un numero molto alto di richieste ricevute in un breve intervallo di tempo, misurando il numero di richieste ricevuto, quelle che sono fallite o che hanno avuto successo e la latenza impiegata, in seguito alla misurazione è possibile trarre conclusioni come ad esempio valutare il 95° percentile di latenza di risposta alle richieste o imporre un vincolo sulla quantità massima tollerabile.

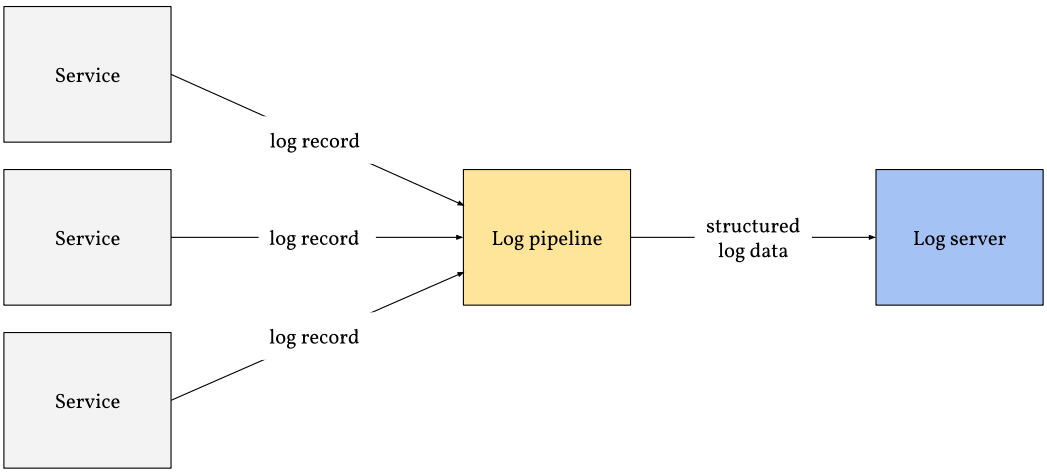

Figura 4.6: Esempio di pattern per la raccolta di log generico

4.3.4 Pattern per la raccolta di log e tracciamento delle interazioni dell'utente

Per quanto riguarda il requisito del tracciamento delle interazioni dell'utente con la cartella clinica elettronica per effettuare elaborazioni e analisi degli intenti, si propone una soluzione che utilizza pattern per la raccolta di log [74], per effettuare audit logging, in modo che ciascuna interazione degli utenti possa essere tracciata tramite log (di cui un campo debba obbligatoriamente essere l'identificativo dell'utente), e ciascuno di essi venga raccolto da un servizio di centralizzazione che effettui l'aggregazione di ciascuno di essi e possa inoltrare i dati strutturati ad un ulteriore servizio che possa essere interrogato per una successiva fase di analisi ed elaborazione degli intenti.

Ciascun servizio raccoglie tutte le informazioni inoltrate su standard output e le inoltra ad un servizio di aggregazione, che raccoglie tutti i dati, li trasforma e inoltra i dati strutturati ad un servizio che permette l'interrogazione di essi per effettuare un'analisi ed elaborazione delle informazioni. Le richieste per l'elaborazione delle informazioni possono avvenire via interfaccia grafica, per cui sono di semplice utilizzo e comprensione (vd. Figura 4.6).

Questa analisi è ad un livello di astrazione architetturale per cui indipendente dalle tecnologie utilizzate, ma in letteratura è possibile trovare esempi di applicazioni concrete di questi pattern in ambito sanitario [4], che utilizzano lo stack di tecnologie open source 'ELK' [75] per l'implementazione di questi pattern, mostrando performance maggiori rispetto a soluzioni commerciali

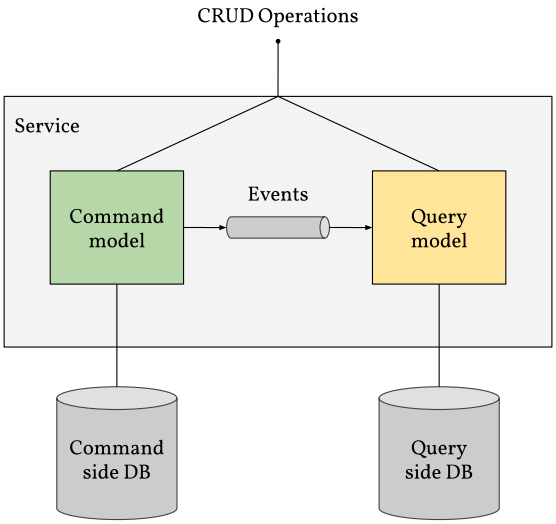

Figura 4.7: Esempio di pattern CQRS generico

[84].

Alternativamente, è possibile applicare pattern come 'event sourcing', che permettono di tracciare la variazione di stato negli oggetti di dominio, associando a ciascun evento un istante di tempo, in modo da poter essere raccolti (ad esempio mediante event broker), e visualizzati nello stesso ordine in cui sono stati generati, con lo scopo di ricreare l'intera sequenza di eventi, in ordine cronologico, per effettuare, tra le possibili applicazioni, anche un'analisi degli intenti [74].

4.3.5 Pattern per l'incremento delle performance

Si riporta di seguito un pattern per l'incremento delle performance che può essere applicato efficacemente se sono rispettati diversi requisiti:

Nel caso in cui il numero di richieste nei servizi fosse elevato, si identificasse questo come punto critico per l'incremento della latenza e conseguenti rallentamenti del sistema, si rilevasse una significativa asimmetria tra il numero di richieste di lettura e scrittura, con le prime molto maggiori delle seconde, e fosse considerata tollerabile una consistenza di tipo 'eventual': sarebbe possibile applicare il pattern CQRS dove necessario (vd. Figura 4.7), pattern pensato per incrementare le performance dei servizi che consiste nella separazione in due parti dei tipi di operazioni che possono essere effettuate (command, query)

e gestirle da due differenti servizi con database ottimizzati rispettivamente uno per letture (query) e l'altro per le scritture (write).

I dati contenuti in entrambi i database saranno mantenuti allineati mediante uno scambio di eventi che porterà i dati relativi alla scrittura verso la controparte utilizzata per la lettura, riducendo la latenza per le richieste di lettura ma garantendo una consistenza di tipo 'eventual', dove non necessariamente le letture effettuate dopo una scrittura sullo stesso dato saranno consistenti, ma è garantito che prima o poi lo diventeranno [74].

4.4 Implementazione

Un prototipo di Cartella Clinica Elettronica è stato sviluppato per validare la soluzione architetturale proposta e fornire delle linee guida per l'implementazione complessiva della stessa. Il progetto è visualizzabile al link in biografia [16].

Di seguito una discussione sull'implementazione del servizio di terapia nel prototipo della Cartella Clinica Elettronica.

L'architettura utilizzata per l'implementazione del servizio è di tipo 'layered' [72], composta da 3 layers ('Domain', 'Application' e 'Infrastructure'), utilizzando una 'hexagonal architecture', dove sono state definite interfacce come connettori utilizzati per l'implementazione di adattatori per lo scambio di informazioni tra le parti della struttura del servizio [11]. Il rispetto delle dipendenze tra i vari strati dell'architettura sono stati verificati mediante test unitari delle dipendenze tra le parti.

I servizi sono stati implementati secondo Domain Driven Design, dove sono stati definiti gli elementi del dominio ed oggetti per la loro elaborazione per ciascuno di essi. Prendendo in considerazione il servizio di terapia, questo ha come elementi del suo dominio la pianificazione delle terapie ed eventuali conflitti con esse, queste sono state modellate mediante le risorse FHIR di 'CarePlan' e 'AllergyIntolerance', risorse che hanno riferimenti ad altre, come 'Patient', che identifica univocamente all'interno del sistema il paziente che si sta prendendo in considerazione.

Questo servizio deve permettere la lettura e scrittura di terapie per i propri pazienti, esponendo API per effettuare le operazioni, per cui sono esposti, secondo le regole RESTful definite nella documentazione FHIR [23] [34] e descritte alla sezione 2.3.1, diversi endpoint che permettono l'esecuzione di ogni operazione necessaria ad effettuare le elaborazioni richieste. Alla Figura 4.8 il diagramma delle sequenze per quanto riguarda la lettura di terapie, la scrittura avviene in modo analogo.

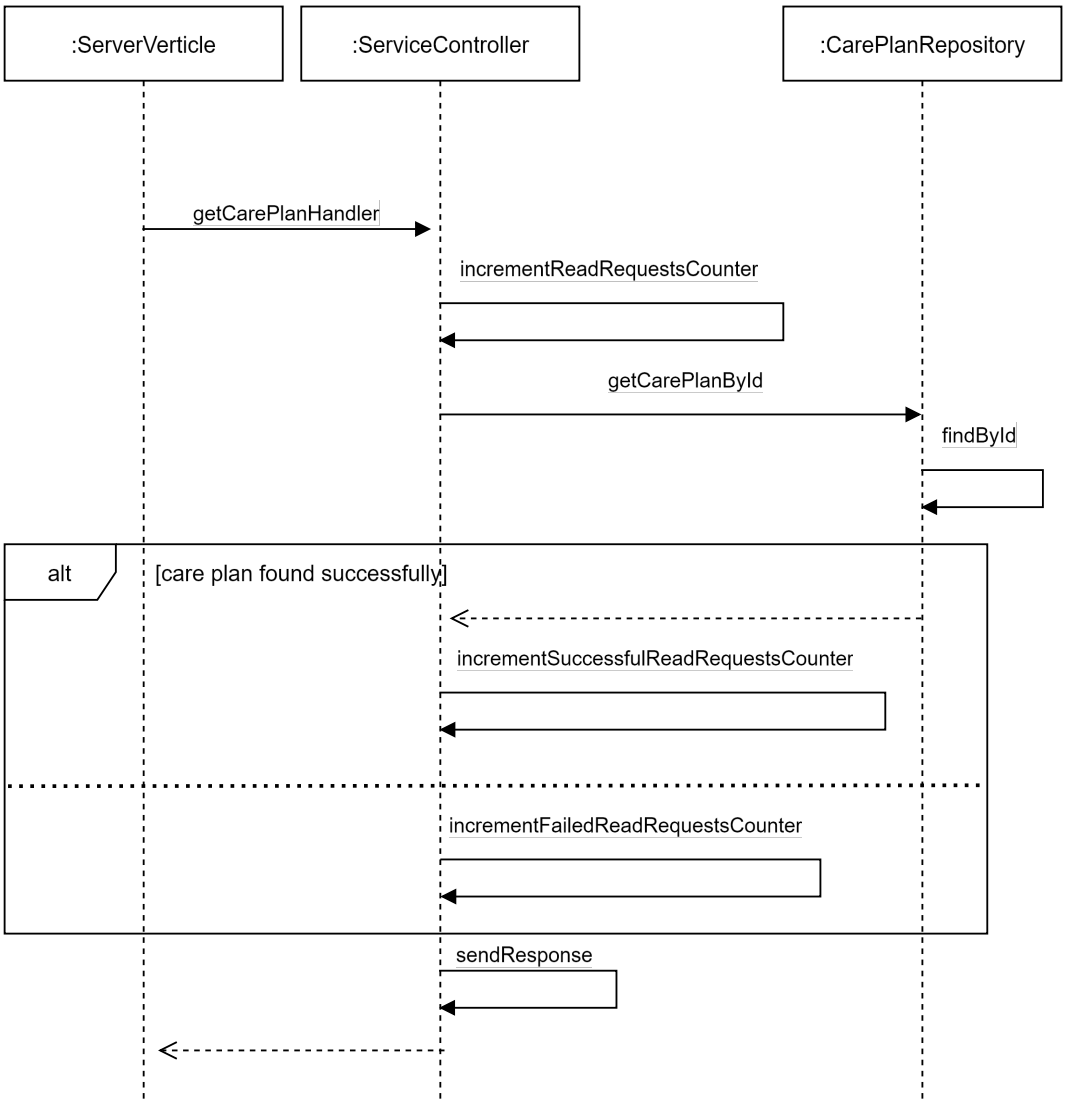

Figura 4.8: Diagramma delle sequenze per la lettura dei piani di terapia

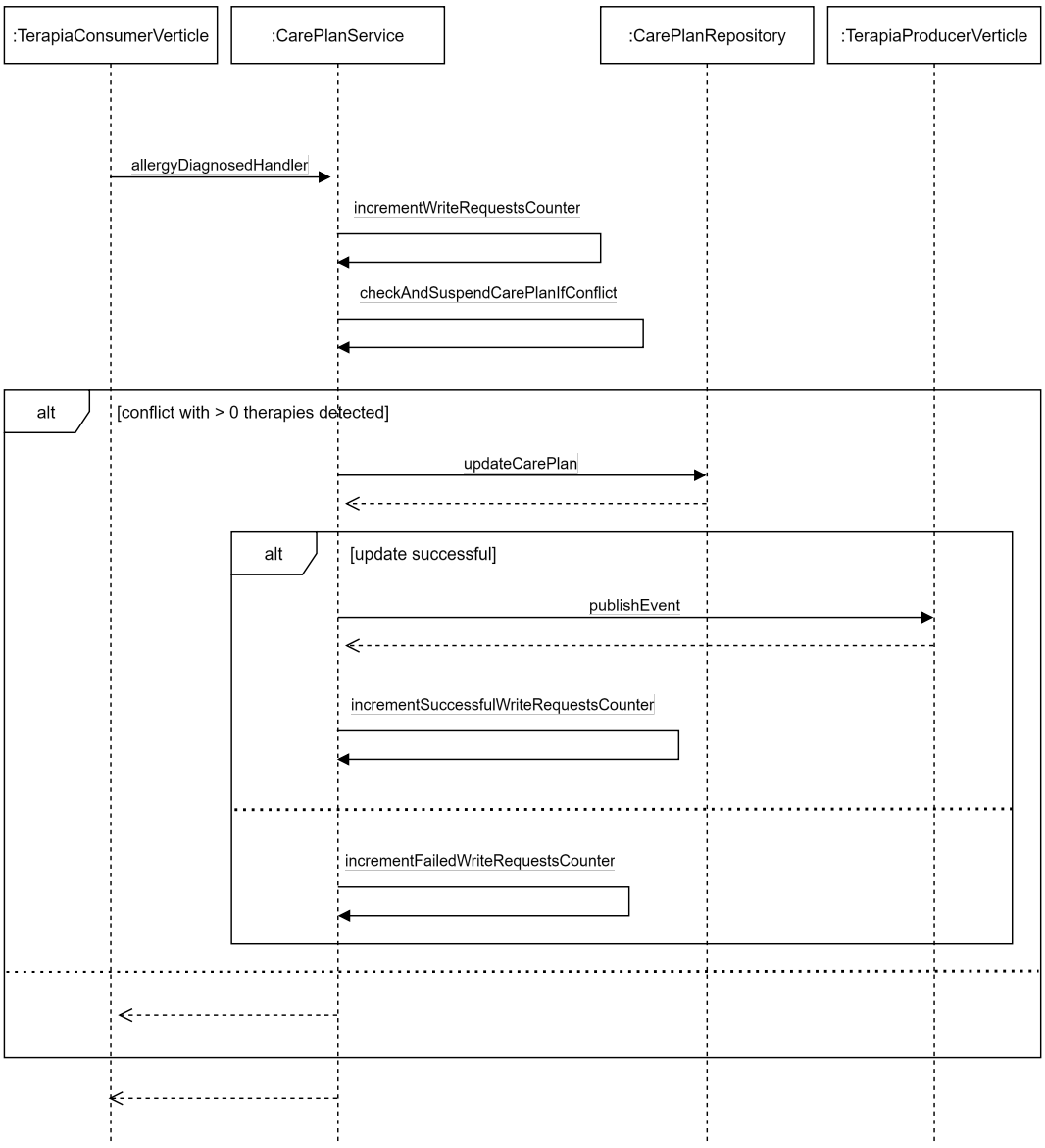

Figura 4.9: Diagramma delle sequenze per la sospensione di una terapia a fronte della rilevazione di un evento di conflitto con una nuova patologia

Lo scambio di eventi tra i servizi avviene tramite 'event broker', come definito in fase di design alla sezione 4.3, alla Figura 4.9 un esempio di come avviene l'elaborazione di un evento da parte del servizio di terapia una volta raccolto un evento di rilevazione di una nuova allergia da parte del servizio di 'anamnesi-pregressa'.

Una volta rilevata un'allergia, il servizio di 'anamnesi-pregressa' pubblica un evento che è raccolto dal 'event broker', archiviato sul suo database e inoltrato ai sottoscritti, tra cui il servizio di 'terapia' preso in considerazione per questa analisi.

Una volta raccolto l'evento, viene verificata o meno la presenza di conflitti con le terapie attualmente pianificate, ed in caso di rilevamento, il campo relativo allo stato della terapia viene aggiornato in modo da identificare la terapia come 'sospesa' a causa delle nuove patologie identificate nel paziente.

Questo genera a sua volta un evento, pubblicato dal servizio di terapia, per notificare che un piano di terapia che era prima pianificata ha cambiato stato ed è stata sospesa, in modo da poter essere raccolta da altri servizi ed agire in modo reattivo a questo cambiamento per generare ulteriori elaborazioni in altre parti del sistema informativo, come può essere un servizio per la gestione degli aspetti amministrativi ed eventuali rimborsi automatici per i pazienti, oppure il servizio di CUP per la riprogrammazione automatizzata di visite e controlli legati ai nuovi cambiamenti causati dalla sospensione della terapia e dalla rilevazione delle patologie, automatizzando diversi processi, non solo all'interno della cartella clinica elettronica, ma relativi al sistema informativo nel suo complesso, riducendo i tempi di attesa, eliminando l'eventuale errore umano e semplificando operazioni che vengono attualmente effettuate manualmente.

Questi servizi sono stati descritti in questa sezione in quanto esemplificativi dell'interezza del prototipo, gli stessi ragionamenti e meccanismi mostrati in questi esempi sono stati applicati anche negli altri servizi, allo stesso modo di come sono stati descritti in questa sezione, ed ulteriori servizi possono essere implementati secondo queste linee guida generali.

4.4.1 Esempio di raccolta delle metriche e monitoraggio del sistema complessivo

La raccolta di metriche del sistema permette di effettuare query per l'analisi delle informazioni, come può essere la latenza media o al 95° percentile Figura 4.10, per verificare il tempo di risposta di ciascun servizio a fronte delle richieste ricevute, oppure rilevare il numero di richieste che hanno avuto successo o sono fallite Figura 4.11, ed il numero di richieste di lettura e scrittura entro un certo intervallo di tempo Figura 4.12 Figura 4.13, in modo da valutare l'efficacia dell'applicazione di pattern come CQRS [74] a specifici servizi;

inoltre, possono essere utilizzati per notificare eventuali anomalie rilevate dal sistema.

4.4.2 Tecnologie utilizzate

Sono state utilizzate diverse tecnologie per l'implementazione del prototipo, di seguito un elenco e motivazioni che hanno portato alla scelta delle principali:

- **Kotlin / Java**: linguaggi della JVM (versione utilizzata: 21), la scelta di utilizzare questi linguaggi è stata soprattutto per la grande quantità di librerie e framework disponibili per questa piattaforma, e per poter utilizzare la libreria 'Hapi' [88] che implementa lo standard FHIR per la JVM. Un'altra possibile opzione è 'Go' [18], linguaggio che permette di raggiungere tipicamente performance molto più alte rispetto ai linguaggi della JVM con un impatto sulle risorse molto più ridotto, utilizzato di frequente per lo sviluppo di microservizi, anche in ambito sanitario, come dimostrato nei precedenti capitoli dopo un'approfondita analisi della letteratura; ma attualmente non gode dello stesso supporto riguardo librerie e framework open source di libero utilizzo come invece accade nel caso della JVM, per cui si è preferito utilizzare quella piattaforma.

- **Hapi**: implementazione per la JVM dello standard FHIR [88] [87];

- **Vert.x**: framework per la JVM utilizzato per lo sviluppo dei server per quanto riguarda i microservizi ed API-Gateway [14].

- **Prometheus**: tecnologia open source utilizzata per effettuare il monitoraggio dei sistemi e valutazione mediante metriche. Assieme ad altre tecnologie come 'Grafana' [20], con cui può collaborare, sono tra le più utilizzate nelle architetture per quanto riguarda i pattern di osservabilità [47], anche in ambito sanitario [86]. È possibile monitorare in tempo reale diverse informazioni, tra cui il numero di richieste effettuate, quante hanno dato esito positivo o negativo, e monitorarne il tipo, ad esempio per tracciare il numero di richieste di lettura o scrittura che vengono effettuate verso il servizio considerato, per valutare o meno criticità e possibili soluzioni mirate, oppure la validazione e concretizzazione mediante scenari di attributi di qualità; Inoltre in letteratura è possibile trovare articoli che dimostrano come utilizzare questa tecnologia per la rilevazione automatica e preventiva di anomalie [49].

- **Kubernetes**: tecnologia open source inizialmente sviluppata da Google, utilizzata per i deploy; molto efficace riguardo i microservizi, per via le sue funzionalità come 'autoscale', che permette di effettuare scaling

Figura 4.10: Esempio di tracciamento del 95° percentile del tempo richiesto dai servizi ad elaborare ogni richiesta ricevuta negli ultimi 5 minuti, via prometheus GUI

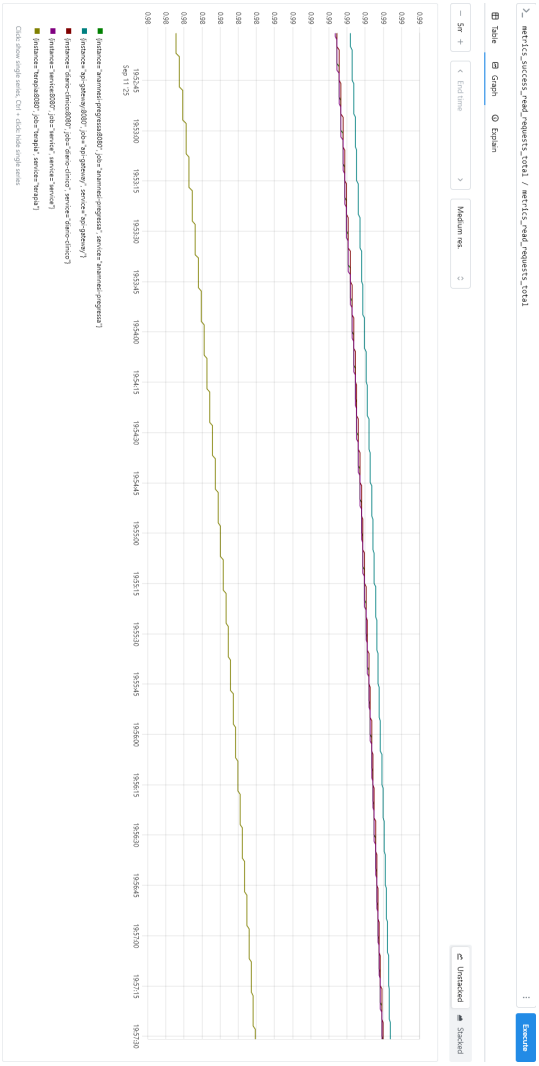

Figura 4.11: Esempio di tracciamento del rapporto tra numero di richieste di lettura che hanno avuto successo rispetto al totale, via prometheus GUI

Figura 4.12: Esempio di tracciamento del rapporto tra numero di richieste di lettura rispetto al totale per ciascun servizio, via prometheus GUI

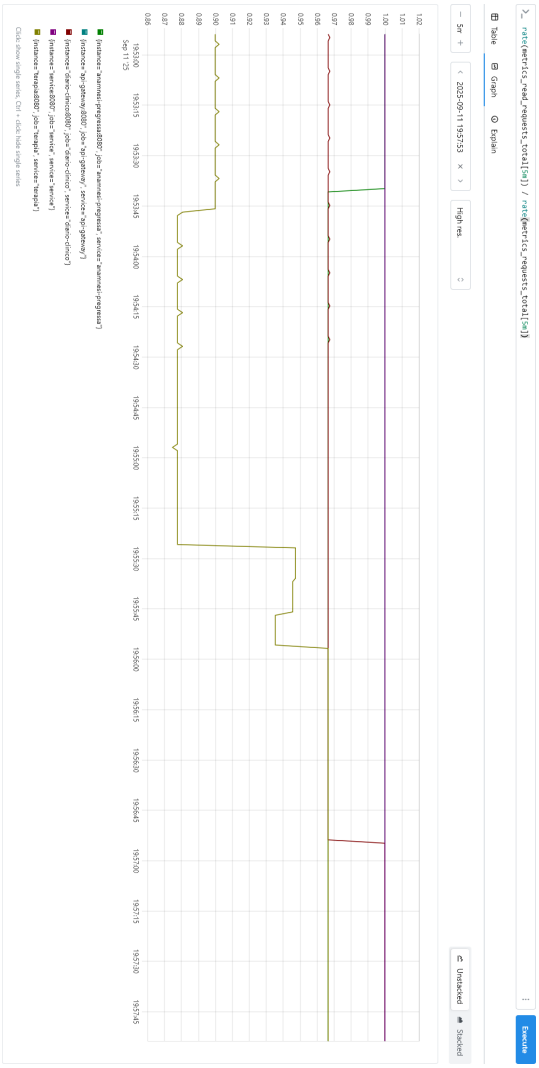

Figura 4.13: Esempio di tracciamento del rapporto tra numero di richieste di lettura rispetto al totale per ciascun servizio negli ultimi 5 minuti, via prometheus GUI

orizzontale automatico in base all'utilizzo delle risorse, capacità chiave che migliora la robustezza e disponibilità dei microservizi [45] e che non può essere garantita da architetture con granularità più alte o di diverso tipo.

- **Kafka**: tecnologia inizialmente sviluppata da Linkedin, poi diventata open source, molto utilizzata soprattutto in architetture a microservizi, anche in ambito sanitario [77], per mantenere una copia di tutti i messaggi scambiati tra i microservizi e su richiesta ripescarli in qualsiasi momento in modo da eventualmente riallineare database oppure anche solo analizzare in maniera sequenziale i dati che sono scambiati tra i servizi [43] [80].

- **JUnit 5**: tecnologia utilizzata per effettuare test automatici e ripetibili [41].

- **Mockk**: tecnologia utilizzata per effettuare test integrativi mediante test double [51].

Altre tecnologie prese in considerazione sono state:

- **Synthea**: strumento per la generazione di dati sanitari sintetici in FHIR [85], utilizzabili per scopi di testing e validazione; in questo prototipo non è stata utilizzata in quanto genera solamente risorse 'Patient' che per questo prototipo non sono state utilizzate per intero ma solamente come riferimenti inseriti in altri tipi di risorse, ma considerando differenti scenari di utilizzo della cartella clinica elettronica rispetto a quelli considerati in questo caso di studio, può rivelarsi efficace.

- **Gatling**: tecnologia sviluppata in 'Scala' [78], utilizzando framework 'Akka' [46], per effettuare test non funzionali di valutazione del comportamento di un sistema a fronte della ricezione di un grande numero di richieste da un numero variabile di client [83] [12], permette anche la generazione di report per mostrare il risultato dei test una volta completati, ma per questo progetto è stato utilizzato 'Prometheus' per cui non è stato necessario.

4.5 Validazione della soluzione proposta

La validazione della soluzione proposta è avvenuta mediante test ripetibili che ricreano situazioni verosimili, definendo un ambiente di esecuzione che replica scenari reali, in modo da verificare il comportamento del sistema a

fronte di situazioni critiche, per valutarne performance e limiti, come spiegato in sottosezione 4.3.3.

4.5.1 Testing e valutazioni sperimentali

Per effettuare testing di validazione riguardo i requisiti non-funzionali, i limiti dell'architettura proposta e una comparazione con le precedenti architetture, è possibile definire dei 'quality attribute scenarios' [5], scenari ripetibili che permettono di creare un ambiente specifico dove misurare, e valutare mediante metriche, il comportamento del sistema a seguito di alcuni stimoli esterni.

Sono stati creati scenari di test in cui misurare, mediante attributi di qualità, le proprietà non funzionali garantite, come il tempo di risposta dei sistemi in funzione di un numero di richieste crescente, misurato sia come media che 95° percentile, o la disponibilità dei servizi a fronte di un fallimento simulato, per valutare il tempo di recupero del servizio in seguito ad una situazione critica, questo per simulare uno scenario in cui si ha urgenza di utilizzare un servizio, che può essere ad esempio 'terapia', ma che per un fallimento non è disponibile, e si ha la necessità che torni operativo autonomamente entro un lasso di tempo il più breve possibile.

Il prototipo sviluppato è predisposto per effettuare i test su ogni servizio, avviando ad ogni test tutti i servizi del sistema, il monitoraggio dei risultati può essere effettuato su ciascun servizio individualmente, in modo da valutare proprietà e necessità specifiche per ciascuno, ed identificare criticità e possibilità di miglioramento adatte a ciascun servizio. Nei test riportati di seguito si riportano scenari relativi al solo servizio di 'terapia', in quanto le proprietà non-funzionali richieste da questo servizio sono più restrittive rispetto agli altri, e richiede di poter gestire carichi di lavoro più alti rispetto alla media, mantenendo un'alta robustezza e disponibilità, resilienza e tempi di elaborazione delle richieste ridotta, ma l'interfaccia grafica utilizzata è predisposta per la visione dei risultati di ciascun servizio del sistema.

Non essendo attualmente disponibile un sistema di raccolta di metriche per la Cartella Clinica Elettronica in uso ad AUSL della Romagna, non è stato possibile effettuare un paragone tra la soluzione architetturale qui proposta e quella attualmente utilizzata, per cui i risultati e grafici mostrati sono solo relativi ai risultati ottenuti da questa proposta architetturale.

I valori esatti dei risultati dei test possono variare a seconda della macchina utilizzata per effettuarli, per cui si riportano di seguito le caratteristiche del computer e la versione del software utilizzato per effettuare queste validazioni, in modo da renderle perfettamente ripetibili:

- **Processore**: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 2208 Mhz, 6 core, 12 processori logici
- **Memoria**: RAM 16,0 GB
- **Versione Prototipo CCE**: la versione del prototipo di CCE implementato per questo elaborato ed utilizzato per i test è stata la: v1.7.2.

Definizione e validazione mediante uno scenario di carico sostenuto crescente

La tecnologia utilizzata per effettuare il deployment dei servizi per il test è stata 'kubernetes', in quanto permette di definire un 'horizontal pod autoscaler' in grado di effettuare scaling di tipo orizzontale per i pod considerati, ed essendo necessario per il servizio di terapia ricreare uno scenario di carico sostenuto, per garantire un'alta disponibilità del servizio si è utilizzata questa funzionalità in modo da valutare il comportamento del sistema a fronte di un grande numero di richieste.

Al Codice 4.1, il file di configurazione utilizzato per definire l'autoscaler relativo al servizio di terapia utilizzato per i test.

Il test definito prevede la creazione di uno scenario dove il sistema riceve un grande numero di richieste concorrentemente, provenienti da diversi utenti, ad intervalli regolari di 30 secondi, con un numero di richieste crescente, partendo da 50 e raggiungendo le 500 richieste, incrementando ogni volta il numero di richieste di 50 rispetto al caso precedente.

La scelta di effettuare test di questo tipo è stata intrapresa dopo un'analisi della letteratura dove sono stati effettuati test simili per la validazione di architetture a microservizi, utilizzando lo standard FHIR [81]. Intervalli di tempo di 30 secondi tra i gruppi di richieste sono stati definiti per dare modo al sistema di rispondere a tutte le richieste prima di inoltrarne un nuovo gruppo; l'utilizzo di un numero crescente di richieste è stato effettuato per mostrare il comportamento del sistema a fronte di diverse tipologie di carico e mostrare l'efficacia di proprietà come la scalabilità orizzontale che viene applicata autonomamente al superamento di una certa soglia di utilizzo delle risorse dedicate al sistema.

I risultati ottenuti sono stati raccolti dal servizio di metriche relativo ai dati sui pod di 'kubernetes' e relativo ai servizi mediante 'Prometheus'. Lo scopo è quello di valutare il comportamento del sistema a fronte di ciascun picco di richieste.

Opzionalmente è possibile definire il test come test automatizzato e ripetibile, ricreando uno scenario ed eseguendo i pod necessari a valutarlo, oltre che aggiungere vincoli per il suo superamento, come riportato al Codice 4.2, dove,

```yaml
# terapia-hpa.yaml (horizontal pod autoscaler for 'terapia')
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: terapia-hpa
  namespace: monitoring-app
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: terapia
  minReplicas: 1  # minimum num. of replicas to scale down to
  maxReplicas: 20  # maximum num. of replicas to scale up to
  metrics:
    # scale based on CPU utilization
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          # 2/3 of provided % CPU is required to scale up
          averageUtilization: 66
```

Codice 4.1: File 'yaml' di configurazione 'kubernetes' per la realizzazione di un 'horizontal pod autoscaler' relativo al servizio di 'terapia', per la creazione dello scenario di validazione

```kotlin
@Test
@DisplayName("Escalating spike load test")
@Timeout(30 * 60) // 30 minutes timeout
fun performanceEvaluationSustainedAverageLoad() {
    setUpEnvironment(k8sFiles)
    val results = escalatingSpikeTest(
        // number of iterations and requests multiplier
        10, 50,
        // endpoints and data to be sent
        "/CarePlan", carePlanTest, "/CarePlan/002"
    )

    // Log the results
    results.logSummary()

    // Optional constraints can be added here to pass the
    // test. All data is fetched through Prometheus's API.
    assertAll(
        { assertTrue(
            results.p95ResponseTime < 2500,
            "95th percentile response time must be < 2500 ms"
            )
        },
        { assertTrue(
            results.replicaNumber > 1,
            "Should have scaled horizontally"
            )
        },
    )
}
```

Codice 4.2: Test con carico crescente nel tempo

dopo un'iniziale richiesta di POST, contenente i dati relativi ad un 'CarePlan' di esempio (piano di terapia definito come risorsa FHIR), sono stati effettuati 10 gruppi di richieste di lettura GET sulla stessa risorsa, ad intervalli regolari di 30 secondi dove ciascun gruppo di richieste di lettura era di 50 richieste superiore al precedente, partendo da 50 e raggiungendo il valore finale di 500.

I risultati possono essere raccolti tramite la API esposta da Prometheus, effettuando query utilizzando il linguaggio di interrogazione definito da Prometheus stesso, e i risultati possono essere raccolti ed elaborati dal linguaggio utilizzato per i test. In questo caso il test è considerato superato se il tempo di risposta al 95° percentile di tutte le richieste inviate non ha superato i 2500 millisecondi e il servizio di terapia ha effettuato scaling di tipo orizzontale, in modo da verificare la corretta generazione di repliche e 'load balancing' tra esse simulando uno scenario reale in cui il servizio di terapia riceve un grande numero concorrente di richieste da gestire, mantenendo comunque un'alta disponibilità e tempi di risposta ridotti. È possibile definire altri vincoli, relativi ad esempio al numero di richieste che hanno avuto risposta positiva, definendo ulteriori query da inoltrare al endpoint di Prometheus.

Per monitorare il comportamento del sistema durante il test, è possibile visualizzare tramite interfaccia grafica i dati raccolti da 'Prometheus'. I risultati ottenuti dalla macchina utilizzata per effettuare i test sono discussi di seguito.

Al termine del test il servizio di terapia ha effettuato scaling orizzontale fino a raggiungere 6 repliche, per far fronte al numero sempre crescente di richieste. Questo ha permesso di ridurre gradualmente la latenza media (vd. Figura 4.14), nonostante il numero sempre crescente di elaborazioni da eseguire. La latenza media è passata dal valore iniziale di 1.2 secondi, per far fronte a 50 richieste di lettura e utilizzando un singolo pod, ad un valore finale di 0.2 secondi, per far fronte a 500 richieste di lettura e utilizzando 6 pod.

Il valore di latenza al 95° percentile (vd. Figura 4.15), misurato ad ogni intervallo di 30 secondi dove veniva inoltrato un nuovo gruppo di richieste, è passato dai 2000 millisecondi iniziali, per far fronte a 50 richieste di lettura ed utilizzando un singolo pod, ad un valore finale di circa 1000 millisecondi per far fronte a 500 richieste di lettura ed utilizzando 6 pod di replica.

Definizione e validazione del tempo di recupero mediante uno scenario di fallimento di un servizio

Un altro test definito per valutare la resilienza dei servizi, è quello del fallimento di un servizio, considerando il caso in cui non sia stato fatto scaling orizzontale, e quindi sia presente un singolo pod, è possibile valutare il tempo necessario al servizio a tornare in funzione autonomamente.

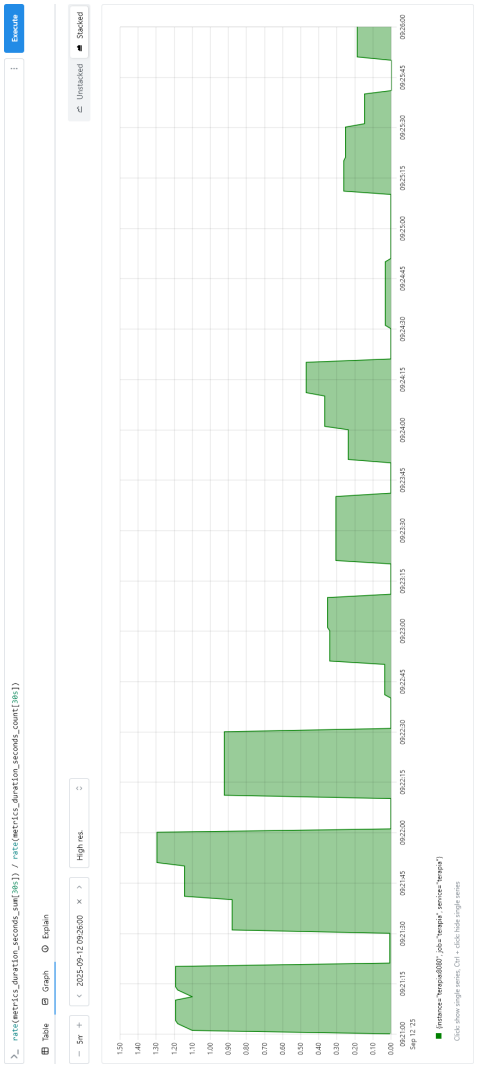

Figura 4.14: Validazione: latenza media, calcolata ogni 30 secondi, a fronte di un numero crescente di richieste da 50 a 500

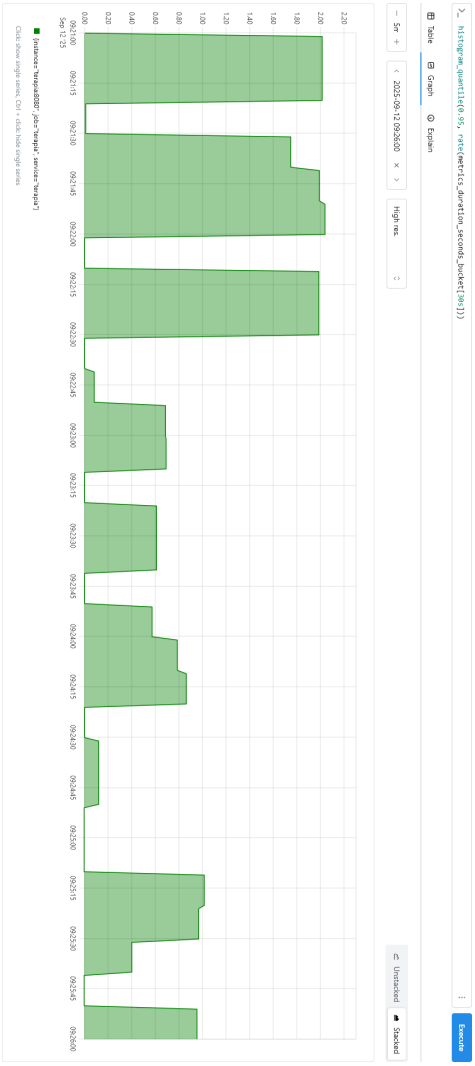

Figura 4.15: Validazione: latenza al 95° percentile, calcolata ogni 30 secondi, a fronte di un numero crescente di richieste da 50 a 500

Questo scenario è pensato per simulare il caso in cui un paziente necessiti di un trattamento urgente e il servizio smetta improvvisamente di funzionare a causa un fallimento, il servizio deve tornare operativo entro il minor tempo possibile, per permettere al personale di intervenire il prima possibile sul paziente che necessita urgentemente di trattamenti.

Il test è automatizzato e consiste nell'inizializzazione dei servizi del sistema, generando diversi pod, e mandando continuamente segnali di 'health check' per controllare se i servizi sono correttamente in funzione. È possibile definire un intervallo di tempo entro il quale periodicamente inoltrare richieste di 'health check', per questo test è stato utilizzato un intervallo di 1000 millisecondi, per cui, ad ogni secondo, ci viene verificato o meno se il servizio considerato è in funzione.

```
@Test
@DisplayName("Test recovery time for 'terapia' service")
@Timeout(30 * 60) // 30 minutes timeout
fun testTerapiaRecoveryTimeUponFailingWithSinglePod() {
    val result = executeRecoveryTestWithSingleFailure(
        testName = "Terapia Pod Failure",
        initialReplicas = 1,
        testDurationSeconds = 90,
        healthCheckIntervalMs = 1000,
        failureDelaySeconds = 15,
        healthCheckEndpoint = healthEndpoint,
        serviceToKill = "terapia",
        host = "http://localhost:31082",
    )

    result.logSummary()

    // should recover within 15000 milliseconds on average
    result.assertAverageRecoveryTime(15000)
}
```

Codice 4.3: Test per valutare il tempo impiegato da un servizio a tornare operativo dopo un fallimento

Un servizio viene interrotto improvvisamente dopo 15 secondi dall'inizio del test, come nel caso riportato al Codice 4.3, dove il servizio di 'terapia' viene interrotto per valutare il tempo impiegato a tornare operativo; ma è stato effettuato un test per ciascun servizio del sistema.

Il pod del servizio di terapia è reattivo, ed impostato per tornare operativo non appena viene interrotto, per cui viene immediatamente ripristinato e non appena pronto ad elaborare le richieste ricomincia a rispondere correttamente ai segnali di 'health check'. Intanto viene misurato il tempo trascorso dall'ultimo segnale di 'health check' che ha avuto successo prima del primo fallimento, fino al primo successo dopo che il servizio è tornato operativo, in modo valutare con una sensibilità di $\pm$ 1000 millisecondi il tempo impiegato dal servizio a tornare operativo.

```
09:27:46.632 INFO  = TEST SUMMARY FOR 'Terapia Pod Failure' =
09:27:46.642 INFO  Test duration: 89 seconds
09:27:46.645 INFO  Total health checks: 88
09:27:46.646 INFO  Successful health checks: 79
09:27:46.652 INFO  Average failure duration: 9,11s
09:27:46.655 INFO  ========================================
```

Codice 4.4: Risultato del test utilizzato per valutare il tempo impiegato da un servizio a tornare operativo dopo un fallimento

Il risultato è stato di circa 9 secondi, per cui è possibile garantire, considerando la macchina utilizzata per effettuare questi test, che il servizio torni operativo autonomamente entro questa quantità di tempo, come mostrato dai log al Codice 4.4.

Valori di tempo molto ridotti, come quello qui ottenuto, non è possibile garantirli con architetture monolitiche, in quanto essendo il sistema una singola unità di cui viene effettuato il deploy, il tempo necessario per permettere al servizio di tornare operativo è di gran lunga superiore rispetto al caso dei microservizi dove ogni servizio rappresenta una singola unità funzionale del sistema; inoltre, in caso di guasti, non verrebbe interessato l'intero sistema ma solamente il microservizio considerato, permettendo a tutte le altre funzionalità del sistema di restare operative.

Inoltre, il broker utilizzato per lo scambio di eventi tra i servizi garantisce che gli eventi pubblicati mentre i servizi non erano in funzione vengano recuperati una volta tornati operativi, quindi garantendo un'alta tolleranza ai guasti.

Simulazione di attacchi DOS e DDOS per l'identificazione del carico massimo gestibile

Test come quelli utilizzati per effettuare questa validazione possono essere inoltre utilizzati, per scopi difensivi, per simulare attacchi di tipo 'denial of ser-

vice' o 'distributed denial of service' (DOS o DDOS), che inoltrano un grande numero di richieste al servizio target con lo scopo di impedirgli di elaborare correttamente tutte le richieste, e renderlo inaccessibile.

Effettuando simulazioni come quella qui riportata è possibile valutare quale sia il punto di rottura del sistema, punto oltre il quale il servizio non riesce più a reggere correttamente il carico di richieste ricevuto.

Monitoraggio in contesti reali

Le stesse tecnologie utilizzate per effettuare questa validazione possono essere utilizzate anche in contesti reali, quindi monitorando in tempo reale l'evoluzione dei microservizi del sistema e permettendo l'interrogazione mediante query degli stessi, mostrando grafici in tempo reale della situazione del sistema complessivo con scopo di monitoraggio (compresi malfunzionamenti e notifica di anomalie).

4.6 Analisi dell'architettura proposta

Considerando il caso di studio di questo elaborato, si riportano diverse proprietà non-funzionali che sarebbero maggiormente garantite nel caso di transizione a questo tipo di architettura per la cartella clinica elettronica:

- **Modularità**: una architettura di questo tipo permetterebbe una maggiore flessibilità al cambiamento e modularità, per cui è necessario che i servizi siano costruiti come blocchi modulari, sotto forma di API per permettere la comunicazione tra i vari servizi garantendo un maggiore disaccoppiamento tra essi [3]. È possibile utilizzare le regole di conformità FHIR riguardo architettura RESTful in modo da garantire che ogni servizio utilizzi una struttura conforme allo standard e definita in modo comune agli altri [23]. Questo rispetta il requisito mostrano in precedenza di maggiore autonomia tra le varie funzionalità della cartella clinica.

- **Scalabilità**: utilizzare una architettura a microservizi, piuttosto che una monolitica, permetterebbe una maggiore scalabilità, in quanto essendo i servizi definiti con un livello granularità fine e modulare, è possibile aggiungere facilmente risorse al solo servizio in cui sono richieste, anziché all'intero ecosistema delle applicazioni, rendendo più semplice la scalabilità verticale che è invece più limitata nel caso dell'architettura monolitica attuale [3] (vd. Figura 3.2); utilizzare server con architetture a microservizi permetterebbe inoltre di effettuare scaling orizzontale

dove invece non sarebbe possibile applicarlo con la stessa efficacia per architetture monolitiche [81] [82], per cui se necessario è possibile adottare meccanismi di 'load balancing' e 'horizontal scaling' per distribuire le richieste in maniera uniforme tra questi servizi, inoltre è sempre possibile effettuare 'vertical scaling' per assegnare un maggior numero di risorse al servizio, quando necessario [82].

- **Disaccoppiamento**: questo tipo di architetture permette una riduzione dell'accoppiamento tra le varie funzionalità proposte, dopo aver effettuato un'analisi del dominio per lo scenario rappresentativo considerato, sono state individuate diverse funzionalità atomiche della cartella clinica che sono il più possibile isolate tra loro, in modo da garantire che le singole funzionalità della cartella clinica possano continuare ad operare anche in caso di partizionamenti della rete.

Le principali limitazioni riguardano il ruolo del API Gateway e broker che possono diventare eventuali colli di bottiglia, in quanto tutte le comunicazioni tra servizi devono attraversare questi, inoltre alcuni servizi necessitano di requisiti non funzionali specifici, come la robustezza, e ci si aspetta che alcuni di essi avranno un numero di richieste più alto rispetto agli altri servizi ed asimmetrico (in termini di differenza tra letture e scritture).

Per mantenere una alta disponibilità del servizio di API Gateway è necessario applicare meccanismi di 'scaling' e 'load balancing', requisiti che si possono ottenere facilmente utilizzando strumenti di deploy che permettono di ottenere queste proprietà automaticamente, mentre per quanto riguarda il broker è possibile considerare l'utilizzo di broker decentralizzati.

Inoltre, l'utilizzo di soluzioni distribuite non permettono di raggiungere performance superiori rispetto a quelle monolitiche, dove l'intero sistema ha un deploy su un singolo nodo come un'unica unità, in quanto nelle architetture distribuite, utilizzando diversi nodi dove la memoria tra i sistemi non è condivisa, è necessario scambiare informazioni tra sistemi localizzati in differenti punti dello spazio, per cui la latenza dovuta alla rete è logicamente superiore rispetto alle comunicazioni intra-processo.

Eventualmente, se il carico da gestire da parte dei singoli servizi fosse particolarmente elevato e si rilevasse una significativa asimmetria tra il numero di richieste di lettura e scrittura, con le prime molto maggiori delle seconde, se fosse considerata tollerabile una consistenza di tipo 'eventual' sarebbe possibile applicare il pattern CQRS descritto in sottosezione 4.3.5. Pattern pensato per essere applicato ai singoli servizi dell'architettura, permettendo quindi l'aggiunta di proprietà non funzionali specifiche a seconda del servizio considerato, in modo da intervenire in maniera isolata sulle singole funzionalità che le necessitano, lasciando le altre invariate.

Nello specifico, questo pattern permette di incrementare le performance effettuando una separazione delle operazioni in modo da essere gestite da due differenti servizi, che si occupano ciascuno di una parte delle operazioni complessive del servizio di partenza. Pattern che si è rivelato efficace in casi d'uso concreti sempre in ambito sanitario [71].

Conclusioni

L'obiettivo di questo di questo elaborato è quello di effettuare un'analisi dello stato dell'arte delle architetture software in ambito sanitario ed individuare criticità e margini di miglioramento per il caso di AUSL della Romagna, dove sono state proposte soluzioni architetturali distribuite a microservizi e ad eventi per superare le principali limitazioni attuali e riportare allo stato dell'arte il sistema informativo sanitario utilizzato. Al fine di validare le soluzioni proposte, è stato implementato un prototipo di cartella clinica elettronica, con cui si è dimostrata l'effettiva efficacia delle soluzioni architetturali fornite riguardo i requisiti non funzionali moderni, mediante test automatizzati e ripetibili per la misurazione concreta degli attributi di qualità garantiti.

Il presente lavoro si colloca in un contesto in cui la digitalizzazione sanitaria sta attraversando una fase di trasformazione profonda, la soluzione proposta è in linea con le architetture allo stato dell'arte e il moderno approccio 'patient-centered' dove la logica del sistema ruota attorno alle specifiche esigenze del paziente a seconda del contesto sanitario in cui si trova, e presenta una base per l'automatizzazione di diversi processi che vengono attualmente eseguiti in maniera manuale, come la rilevazione e sospensione automatica di terapie già pianificate a seguito dell'identificazione di incompatibilità con eventuali nuove patologie rilevate nel paziente, per cui molteplici possibili sviluppi futuri riguardano la possibilità di permettere, ai diversi componenti del sistema informativo di AUSL della Romagna, la raccolta di eventi generati da altri, allo scopo di automatizzare diversi processi sulla base di quanto mostrato in questo elaborato.

Ringraziamenti

Ringrazio il professor Alessandro Ricci ed i dottori Samuele Burattini ed Angelo Croatti per il supporto costante e per essermi sempre stati vicini in questa esperienza di tesi.

Bibliografia

[1] Eneimi Allwell-Brown. A comparative analysis of hl7 fhir and openehr for electronic aggregation, exchange and reuse of patient data in acute care. *Tukholma: Karolinska Institutet. Viitattu*, 30:2020, 2016.

[2] Amira Rezk Aya Gamal, Sherif Barakat. Standardized electronic health record data modeling and persistence: A comparative review. *Journal of Biomedical Informatics, Volume 114*, 2021.

[3] Prasenjit Banerjee. System integration, from middleware to apis. *International Journal of Computer Trends and Technology*, 72:46–52, 03 2024.

[4] Itamir Barroca Filho, Silvio Costa Sampaio, João Carlos A Tenório, Edvaldo Vasconcelos de C Filho, Matheus Estevam de C Pessoa, Ramon S Malaquias, and Pedro Arthur Fernades. Development of a health dashboard for an electronic health record system. In *2020 20th International Conference on Computational Science and Its Applications (ICCSA)*, pages 16–22. IEEE, 2020.

[5] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, MA, 4 edition, 2022.

[6] Ali Bayramcavus, M Cagri Kaya, and Ali H Dogru. Interoperability of microservice-based systems. In *2021 13th International Conference on Electrical and Electronics Engineering (ELECO)*, pages 594–598. IEEE, 2021.

[7] Giovani Nícolas Bettoni, Thafarel Camargo Lobo, Cecília Dias Flores, Bruno Gomes Tavares dos Santos, and Filipe Santana Da Silva. Application of hl7 fhir in a microservice architecture for patient navigation on registration and appointments. In *2021 IEEE/ACM 3rd International Workshop on Software Engineering for Healthcare (SEH)*, pages 44–51. IEEE, 2021.

[8] Huriviades Calderon-Gomez, Luis Mendoza-Pitti, Miguel Vargas-Lombardo, Jose Manuel Gomez-Pulido, Jose Luis Castillo-Sequera, Jose Sanz-Moreno, and Gloria Sencion. Telemonitoring system for infectious disease prediction in elderly people based on a novel microservice architecture. *IEEE access*, 8:118340–118354, 2020.

[9] D.A. Chappell. *Enterprise Service Bus*. O'Reilly Series. O'Reilly Media, Incorporated, 2004.

[10] Mario Ciampi, Mario Sicuranza, and Stefano Silvestri. A privacy-preserving and standard-based architecture for secondary use of clinical data. *Information*, 13(2), 2022.

[11] Alistair Cockburn. Hexagonal architecture (ports and adapters), 2005. Accessed: 2025-09-15.

[12] Gatling Corp. Gatling: Load testing designed for devops and ci/cd, 2025.

[13] Ministero della Salute, Consiglio Superiore di Sanità, Sessione LII (2019-2022) Sezione I, Prof. Bruno Dallapiccola, Dr. Stefano Moriconi, Gruppo di lavoro "Proposta per lo schema di Riforma dei Sistemi Informativi Sanitari", Prof. Paolo Vineis, Prof. Franco Locatelli, Dott. Giovanni Leonardi, Prof.ssa Paola Di Giulio, Prof. Claudio Cobelli, Dott. Luca De Angelis, Prof. Giancarlo Blangiardo, Dott.ssa Lucia Bisceglia, Dott. Claudio Caccia, Giorgio Cangioli, Prof.ssa Flavia Carle, Prof. Andrea Cavalli, Prof.ssa Giusella Finocchiaro, Dott. Danilo Fusco, Dott.ssa Enrica Massella Ducci Teri, Prof. Gianluca Mazzini, Dott.ssa Serena Battilomo, and Dott.ssa Claudia Biffoli. CSS Sezione I – Proposta per lo schema di Riforma dei Sistemi Informativi Sanitari. Rapporto tecnico, Ministero della Salute, Consiglio Superiore di Sanità, Roma, 2022. Coordinatori: Prof. Paolo Vineis, Prof. Franco Locatelli. Segretario: Dr. Stefano Moriconi.

[14] Eclipse Foundation. Eclipse vert.x, 2025.

[15] Evans. Domain-driven design: Tacking complexity in the heart of software. *Addison-Wesley Professional:*, 01 2004.

[16] Marco Fontana. Ausl romagna microservizi cce proposta di progetto. `https://github.com/MarcoFontana48/AUSL-Romagna-microservizi-CCE-proposta-di-progetto`, 2025.

[17] Neal Ford, Rebecca Parsons, Patrick Kua, and Pramod Sadalage. *Building evolutionary architectures*. " O'Reilly Media, Inc.", 2022.

[18] Google LLC. The go programming language (golang), 2025.

[19] Konrad Gos and Wojciech Zabierowski. The comparison of microservice and monolithic architecture. pages 150–153, 04 2020.

[20] Grafana Labs. Grafana: The open observability platform, 2025.

[21] Richard Hill, Dharmendra Shadija, and Mehran Rezai. Enabling community health care with microservices. In *Proceedings of the 15th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA) and the 16th IEEE International Conference on Ubiquitous Computing and Communications (IUCC)*, pages 1444–1450. IEEE, 2018.

[22] HL7 International. Fhir: Bundle resource type. `https://www.hl7.org/fhir/bundle.html`, 2023. Consultato ad aprile 2025.

[23] HL7 International. Fhir: Conformance. `https://build.fhir.org/conformance-rules.html`, 2023. Consultato ad aprile 2025.

[24] HL7 International. Fhir: documents. `https://build.fhir.org/documents.html`, 2023. Consultato ad aprile 2025.

[25] HL7 International. Fhir: documents, content. `https://build.fhir.org/documents.html#content`, 2023. Consultato ad aprile 2025.

[26] HL7 International. Fhir: esecuzione di operazioni. `https://build.fhir.org/operations.html#executing`, 2023. Consultato ad aprile 2025.

[27] HL7 International. Fhir: Fast healthcare interoperability resources. `https://www.hl7.org/fhir/`, 2023. Consultato ad aprile 2025.

[28] HL7 International. Fhir: messaging. `https://build.fhir.org/messaging.html`, 2023. Consultato ad aprile 2025.

[29] HL7 International. Fhir: Operationdefinition. `https://build.fhir.org/operationdefinition.html`, 2023. Consultato ad aprile 2025.

[30] HL7 International. Fhir: Overview. `https://www.hl7.org/fhir/overview.html`, 2023. Consultato ad aprile 2025.

[31] HL7 International. Fhir: Overview for sofware architects. `https://www.hl7.org/fhir/overview-arch.html`, 2023. Consultato ad aprile 2025.

[32] HL7 International. Fhir: request message example. `https://build.fhir.org/message-request-link.json.html`, 2023. Consultato ad aprile 2025.

[33] HL7 International. Fhir: response message example. https://build.fhir.org/message-response-link.json.html, 2023. Consultato ad aprile 2025.

[34] HL7 International. Fhir: Restful api. https://build.fhir.org/http.html, 2023. Consultato ad aprile 2025.

[35] HL7 International. Fhir: storage. https://build.fhir.org/storage.html, 2023. Consultato ad aprile 2025.

[36] HL7 International. Homepage di hl7 international. https://www.hl7.org/, 2023. Consultato ad aprile 2025.

[37] Marek Horváth, Vladyslav Sakhnenko, and Filip Gurbál'. Comparison of scalability and performance in microservices and monolithic architectures. *2024 IEEE 17th International Scientific Conference on Informatics (Informatics)*, pages 82–87, 2024.

[38] International Organization for Standardization. Iso standard 20269. https://www.iso.org/standard/20269.html, 1994. Consultato ad aprile 2025.

[39] D. Wanta M. Midura J. Kryszyn, W. Smolik and P. Wróblewski. Comparison of openehr and hl7 fhir standards. *International Journal of Electronics and Telecommunications, vol. 69, Art. no. 1*, 2023.

[40] W. G Prabath Jayatissa, Vajira H W Dissanayake, and Roshan Hewapathirane. Review on master patient index, 2018.

[41] JUnit Team. JUnit 5: The next generation of JUnit, 2024. Version 5.x.

[42] Dipanker Jyoti and Jonathan A. Hutcherson. Salesforce integration architecture. In *Salesforce Architect's Handbook*, pages 185–221. Apress, Berkeley, CA, 2021.

[43] Narkhede N. Kreps, J. and J. Rao. Kafka: A distributed messaging system for log processing. 2011.

[44] Jacek Kryszyn, Waldemar T Smolik, Damian Wanta, Mateusz Midura, and Przemysław Wróblewski. Comparison of openehr and hl7 fhir standards. *International Journal of Electronics and Telecommunications*, pages 47–52, 2023.

[45] Kubernetes Authors. Kubernetes: Production-grade container orchestration. https://kubernetes.io, 2025. Consultato a maggio 2025.

[46] Lightbend, Inc. Akka: Build powerful reactive, concurrent, distributed systems, 2025.

[47] Bhanuprakash Madupati. Observability in microservices architectures: Leveraging logging, metrics, and distributed tracing in large-scale systems. *European Journal of Advances in Engineering and Technology*, 10(11):24–31, 2023.

[48] Vice Presidente AISIS (Associazione Italiana Sistemi Informativi in Sanità) membro Comitato Scientifico ASSD Marco Foracchia, CIO AUSL Reggio Emilia. Sistemi informativi sanitari: una evoluzione verso l'esterno del recinto... 2022.

[49] Octavian Mart, Catalin Negru, Florin Pop, and Aniello Castiglione. Observability in kubernetes cluster: Automatic anomalies detection using prometheus. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 565–570. IEEE, 2020.

[50] N. Mateus-Coelho, Maria Cruz-Cunha, and L. Gonzaga. Security in microservices architectures. *Procedia Computer Science*, 181(2019):1225–1236, 2021.

[51] MockK Team. Mockk: mocking library for Kotlin, 2024. Version 1.13.x.

[52] OpenEHR. Openehr: Architecture overview. `https://specifications.openehr.org/releases/BASE/development/architecture_overview.html`, 2021. Consultato ad aprile 2025.

[53] OpenEHR. Openehr: Archetypes and templates. `https://specifications.openehr.org/releases/1.0.1/html/architecture/overview/Output/archetyping.html`, 2024. Consultato ad aprile 2025.

[54] OpenEHR. Openehr: Archetypes query language. `https://specifications.openehr.org/releases/QUERY/latest/AQL.html`, 2024. Consultato ad aprile 2025.

[55] OpenEHR. Openehr: Archetypes query language. `https://specifications.openehr.org/releases/BASE/latest/architecture_overview.html#_separation_of_responsibilities`, 2024. Consultato ad aprile 2025.

[56] OpenEHR. Openehr: Architecture overview. https://specifications.openehr.org/releases/AM/latest/Overview.html#_archetype_technology_overview, 2024. Consultato ad aprile 2025.

[57] OpenEHR. Openehr: Compositions. https://specifications.openehr.org/releases/RM/latest/ehr.html#_compositions, 2024. Consultato ad aprile 2025.

[58] OpenEHR. Openehr: Design. https://specifications.openehr.org/releases/BASE/Release-1.0.3/architecture_overview.html#_design_of_the_openehr_ehr, 2024. Consultato ad aprile 2025.

[59] OpenEHR. Openehr: Entry. https://specifications.openehr.org/releases/RM/latest/ehr.html#_entry_package, 2024. Consultato ad aprile 2025.

[60] OpenEHR. Openehr: Folder structure example. https://specifications.openehr.org/releases/CNF/development/platform_test_schedule.html#_tests_of_reference_folder_structure, 2024. Consultato ad aprile 2025.

[61] OpenEHR. Openehr: Home. https://openehr.ch/, 2024. Consultato ad aprile 2025.

[62] OpenEHR. Openehr: Information model (rm). https://specifications.openehr.org/releases/RM/Release-1.1.0/ehr.html#_the_ehr_information_model, 2024. Consultato ad aprile 2025.

[63] OpenEHR. Openehr: Instruction state machine. https://specifications.openehr.org/releases/RM/latest/ehr.html#_the_standard_instruction_state_machine_ism, 2024. Consultato ad aprile 2025.

[64] OpenEHR. Openehr: the ehr. https://specifications.openehr.org/releases/BASE/Release-1.0.3/architecture_overview.html#_the_ehr, 2024. Consultato ad aprile 2025.

[65] OpenEHR. Openehr: the ehr package information model. https://specifications.openehr.org/releases/RM/latest/ehr.html#_ehr_package, 2024. Consultato ad aprile 2025.

[66] OpenEHR. Openehr: Validation. https://specifications.openehr.org/releases/BASE/Release-1.0.3/architecture_overview.html#_validation_during_data_capture, 2024. Consultato ad aprile 2025.

[67] Kofi Osei-Tutu and Yeong-Tae Song. A microservices enterprise architecture for healthcare information exchange (hie) in developing countries. In *2021 10th International Congress on Advanced Applied Informatics (IIAI-AAI)*, pages 762–767, 2021.

[68] Gunjan Pathak and Monika Singh. A review of cloud microservices architecture for modern applications. In *2023 World Conference on Communication & Computing (WCONF)*, pages 1–7. IEEE, 2023.

[69] Miguel Pedrera-Jiménez, Noelia García-Barrio, Santiago Frid, David Moner, Diego Boscá-Tomás, Raimundo Lozano-Rubí, Dipak Kalra, Thomas Beale, Adolfo Muñoz-Carrero, and Pablo Serrano-Balazote. Can openehr, iso 13606, and hl7 fhir work together? an agnostic approach for the selection and application of electronic health record standards to the next-generation health data spaces. *J Med Internet Res*, 25:e48702, Dec 2023.

[70] Nuno Pimenta, António Chaves, Regina Sousa, António Abelha, and Hugo Peixoto. Interoperability of clinical data through fhir: A review. *Procedia Computer Science*, 220:856–861, 2023. The 14th International Conference on Ambient Systems, Networks and Technologies Networks (ANT) and The 6th International Conference on Emerging Data and Industry 4.0 (EDI40).

[71] Petar Rajkovic. Using cqrs pattern for improving performances in medical information systems. 2013.

[72] M. Richards and N. Ford. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, 2020.

[73] Mark Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, Sebastopol, CA, 2016.

[74] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[75] Gurpreet S Sachdeva. Practical elk stack. *Practical ELK Stack. Apress*, 2017.

[76] Nada Salaheddin and Nuredin Ahmed. Microservices vs. monolithic architectures [the differential structure between two architectures]. *MINAR International Journal of Applied Sciences and Technology*, 4:484–490, 10 2022.

[77] Rui Fernando Carvas dos Santos. *Microservices architectures in healthcare with Apache Kafka*. PhD thesis, 2022.

[78] Scala Center. The scala programming language, 2025.

[79] Dharmendra Shadija, Mehran Rezai, and Richard Hill. Towards an understanding of microservices, 2017. Consultato a maggio 2025.

[80] Gwen Shapira, Todd Palino, Rajini Sivaram, and Krit Petty. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, Sebastopol, CA, 2 edition, 2021.

[81] Fahim Shariar Shoumik, Md. Ibna Masum Millat Talukder, Ahmed Imtiaz Jami, Neeaz Wahed Protik, and Md. Moinul Hoque. Scalable micro-service based approach to fhir server with golang and no-sql. In *2017 20th International Conference of Computer and Information Technology (ICCIT)*, pages 1–6, Dhaka, Bangladesh, 2017. IEEE.

[82] Fahim Shariar Shoumik, Md Ibna Masum Millat Talukder, Ahmed Imtiaz Jami, Neeaz Wahed Protik, and Md Moinul Hoque. Scalable micro-service based approach to fhir server with golang and no-sql. In *2017 20th International Conference of Computer and Information Technology (ICCIT)*, pages 1–6. IEEE, 2017.

[83] Siddhant Shrivastava and S Prapulla. Comprehensive review of load testing tools. *International Research Journal of Engineering and Technology*, 7(3392-3395):43, 2020.

[84] Sung Jun Son and Youngmi Kwon. Performance of elk stack and commercial system in security log analysis. In *2017 IEEE 13th Malaysia international conference on communications (MICC)*, pages 187–190. IEEE, 2017.

[85] Synthea Team. Synthea: Synthetic patient generation. `https://syntheticheal th.github.io/synthea/#home`. Consultato a luglio 2025.

[86] James Turnbull. *Monitoring with Prometheus*. Turnbull Press, 2018.

[87] University Health Network. Hapi fhir - java api for hl7 fhir clients and servers. `https://github.com/hapifhir/hapi-fhir`. Consultato ad aprile 2025.

[88] University Health Network. Hapi fhir website. `https://hapifhir.io/`. Consultato ad aprile 2025.

[89] Min L. Wang R. Wang, L. Archetype relational mapping - a practical openehr persistence solution. *BMC Med Inform Decis Mak 15, 88*, 2015.

[90] M. Zagra and W. Mazzucco. Sistema informativo sanitario (sis). In F. Vitale and M. Zagra, editors, *Igiene, Epidemiologia ed Organizzazione Sanitaria orientati per problemi*, pages 373–382. 2012.