

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI MATEMATICA "TULLIO-LEVI CIVITA"

CORSO DI LAUREA IN INFORMATICA

## Migrazione da un'architettura monolitica a un'architettura a microservizi

*Relatore: Prof. Tullio Vardanega*

*Laureando: Alessia Domua*

*Migrazione da un'architettura Monolitica a un'architettura a Microservizi*  
Tesi di laurea triennale.  
Autore: Alessia Domua  
Copyright: I 2025.

# Sommario

La presente tesi di laurea triennale descrive la mia esperienza di *stage* svolta presso l'azienda SogeaSoft S.r.l. L'attività di *stage* è durata 304 ore totali svolte tra il 23 settembre e il 17 novembre 2024.

L'obiettivo del progetto di *stage* è la migrazione da un'architettura monolitica a un'architettura a microservizi: in particolare, l'azienda richiede un'analisi approfondita del *software* gestionale di SogeaSoft S.r.l. per poi procedere con l'individuazione e l'estrazione dei servizi offerti dal *software* stesso.

La presente relazione si suddivide in quattro capitoli:

- **L'azienda:** presentazione dell'azienda dove ho svolto lo *stage* dal punto di vista delle tecnologie utilizzate e dell'organizzazione interna;
- **Progetto di *stage*:** descrizione del progetto di *stage*, del dominio applicativo, degli obiettivi e delle ragioni che hanno motivato questa particolare scelta;
- **Svolgimento dello *stage*:** descrizione delle attività svolte e dei risultati raggiunti;
- **Valutazione retrospettiva:** analisi retrospettiva del lavoro svolto, con particolare riferimento al prodotto finale e ai processi attuati; valutazione della mia evoluzione professionale e personale al termine dello *stage*.

L'appendice del documento contiene il Glossario dei termini e la lista degli Acronimi.

La relazione adotta i seguenti accorgimenti:

- i termini in lingua diversa dall'italiano sono segnalati in corsivo, ad eccezione dei nomi propri;
- i termini presenti nel Glossario sono caratterizzati da una **G** a pedice.
- ogni figura è accompagnata da una didascalia che ne descrive il contenuto e, nel caso non sia realizzata da me, la fonte da cui è tratta;
- definizioni, regole e affermazioni che non derivano da un ragionamento esplicito nel testo, come ad esempio conclusioni non immediatamente giustificabili, sono accompagnate da un numero ad apice, che indica il riferimento bibliografico da cui provengono.

# Indice

<b>1 Azienda ospitante</b>	<b>7</b>
1.1 SogeaSoft S.r.l . . . . .	7
1.2 Organizzazione aziendale . . . . .	7
1.3 Prodotti di SogeaSoft S.r.l. . . . .	8
1.3.1 Target dell'azienda . . . . .	9
1.4 Modello di sviluppo . . . . .	9
1.5 Organizzazione interna . . . . .	10
1.5.1 Gestione della configurazione . . . . .	12
1.5.2 Gestione dell'informazione . . . . .	12
1.5.3 Processi di formazione . . . . .	13
1.6 Ciclo di vita di un progetto <i>software</i> . . . . .	14
1.6.1 Analisi preliminare e raccolta dei requisiti . . . . .	15
1.6.2 Progettazione . . . . .	16
1.6.3 Implementazione . . . . .	17
1.6.4 Verifica e validazione . . . . .	18
1.6.5 Manutenzione . . . . .	19
1.7 Tecnologie utilizzate . . . . .	20
1.8 L'innovazione in SogeaSoft S.r.l. . . . .	22
<b>2 Progetto di <i>stage</i></b>	<b>24</b>
2.1 Gestione degli <i>stage</i> in SogeaSoft S.r.l. . . . .	24
2.2 Il <i>software</i> SAIonWeb . . . . .	25
2.2.1 Funzionalità generali di SAIonWeb . . . . .	26
2.2.2 L'architettura di SAI . . . . .	27
2.2.3 La migrazione . . . . .	29
2.3 Obiettivi del progetto di <i>stage</i> . . . . .	30
2.3.1 Obiettivi aziendali . . . . .	30
2.3.2 Vincoli . . . . .	31
2.4 Metodo di lavoro . . . . .	32
2.4.1 Pianificazione . . . . .	32
2.4.2 Modello di sviluppo . . . . .	33
2.4.3 Strumenti . . . . .	34
2.4.4 Revisioni di progresso . . . . .	34
2.5 Motivazioni della scelta . . . . .	34
2.5.1 Obiettivi e aspettative personali . . . . .	35
<b>3 Svolgimento dello <i>stage</i></b>	<b>37</b>
3.1 Conoscenza del dominio di applicazione . . . . .	37
3.2 Attività svolte . . . . .	38
3.2.1 Analisi dei requisiti . . . . .	38
3.2.2 Progettazione . . . . .	40
3.2.3 Implementazione . . . . .	44

3.2.4	Verifica e validazione . . . . .	50
3.3	Risultati raggiunti . . . . .	52
3.3.1	Il microservizio . . . . .	52
3.3.2	Risultati quantitativi . . . . .	53
3.4	Sviluppi futuri . . . . .	54
<b>4</b>	<b>Valutazione retrospettiva</b>	<b>55</b>
4.1	Soddisfacimento degli obiettivi di <i>stage</i> . . . . .	55
4.1.1	Competenze acquisite . . . . .	55
4.1.2	Bilancio formativo . . . . .	55
<b>5</b>	<b>Glossario dei termini</b>	<b>56</b>
<b>6</b>	<b>Lista degli acronimi</b>	<b>62</b>

# Elenco delle figure

1.1	Prodotti di SogeaSoft S.r.l. . . . .	8
1.2	Diagramma del modello Scrum . . . . .	10
1.3	Certificazione ISO di SogeaSoft S.r.l. . . . .	11
1.4	Microsoft Azure, Wiki . . . . .	13
1.5	Rendicontazione ore dedicate alla formazione . . . . .	14
1.6	Schema Analisi dei Requisiti nel contesto DDD . . . . .	16
1.7	Schema dell'attività di Progettazione nel contesto DDD . . . . .	17
1.8	Come DDD permea il ciclo di vita del <i>software</i> . . . . .	18
1.9	Visione d'insieme dell'utilizzo di DDD nello sviluppo <i>software</i> . . . . .	20
1.10	Tecnologie utilizzate in SogeaSoft S.r.l. . . . .	21
2.1	Differenza tra PoC, Prototipo, MVP . . . . .	24
2.2	Esempio di un ERP generico . . . . .	25
2.3	Schermata del <i>software</i> SAI di SogeaSoft S.r.l. . . . .	26
2.4	Rappresentazione grafica della suddivisione in microservizi . . . . .	29
2.5	Piano di migrazione di SogeaSoft S.r.l. . . . .	30
2.6	Rappresentazione dell'attività di sviluppo effettiva secondo <i>Agile</i> (Scrum) . . . . .	33
3.1	Modello di dominio preso in esame . . . . .	39
3.2	Progettazione per l'estrazione del microservizio . . . . .	41
3.3	Rappresentazione dell'architettura esagonale . . . . .	42
3.4	Schema del pattern <i>Change Data Capture</i> . . . . .	43
3.5	<i>Product Backlog</i> e delle metriche . . . . .	44
3.6	Struttura del codice utilizzata per l'estrazione del microservizio. . . . .	45
3.7	Definizione dei metodi di <b>Rilevamento</b> . . . . .	45
3.8	Implementazione di <b>ResourceHandler</b> e i rispettivi metodi. . . . .	46
3.9	Rappresentazione di un <i>Data Transfer Object</i> . . . . .	47
3.10	Implementazione dell'API per la comunicazione tra <b>OrdineProduzioneFase</b> e il <i>database</i> monolitico . . . . .	48
3.11	Definizione della classe <b>ProductionOrderDto</b> e aggregati . . . . .	49
3.12	Coda implementata con Debezium e gestita da RabbitMQ. . . . .	50
3.13	Schermata d'esempio di Swagger per la lettura di un dato. . . . .	51
3.14	Schermata iniziale di Swagger per la gestione del microservizio <b>MS_Rilevamento</b>	53

# Elenco delle tabelle

2.1	Descrizione degli obiettivi aziendali per il progetto di <i>stage</i> . . . . .	31
2.2	Ripartizione delle ore in base alle attività . . . . .	33
2.3	Tabella degli obiettivi personali . . . . .	36
3.1	Tabella dei requisiti individuati . . . . .	40

# 1 Azienda ospitante

## 1.1 SogeaSoft S.r.l

SogeaSoft S.r.l. è un'azienda di sviluppo *software* fondata nel 1980 a Treviso, con l'obiettivo di progettare e realizzare soluzioni a supporto dei processi aziendali, servendo diverse realtà nel Triveneto. Oggi, SogeaSoft S.r.l. rappresenta una filiale di Bluenext S.r.l., azienda di consulenza informatica fondata nel 2012 a Rimini, attiva su tutto il territorio nazionale. Inizialmente, l'azienda acquisitrice si concentrava esclusivamente sullo sviluppo di un *software* per ottimizzare il settore commerciale delle imprese, ma negli ultimi anni ha ampliato il proprio raggio d'azione, esplorando nuovi settori.

## 1.2 Organizzazione aziendale

La Figura 1.1 mostra come SogeaSoft S.r.l. sia strutturata in diverse aree operative, ciascuna dedicata alla gestione e allo sviluppo dei prodotti e servizi descritti nei punti seguenti:

- **Sistema Aziendale Integrato - SAI:** si occupa dello sviluppo del *software* gestionale *Enterprise Resource Planning<sub>G</sub>* (*ERP<sub>G</sub>*), in particolare rappresenta la libreria di base e gestisce la contabilità.
- **SAICon:** sviluppa soluzioni verticali per il settore delle confezioni (abbigliamento) e calzaturiero, integrate con l'*ERP SAI*.
- **SAIOnWeb:** si concentra su applicativi correlati agli *ERP*, ma anche su soluzioni autonome come *Customer Relationship Management<sub>G</sub>* (*CRM<sub>G</sub>*), *Supplier Relationship Management<sub>G</sub>* (*SRM<sub>G</sub>*), raccolta ordini e *After Sales Service<sub>G</sub>* (*ASS<sub>G</sub>*), ossia una serie di strumenti per ottimizzare le relazioni con i clienti e con i fornitori. Inoltre, sviluppa la nuova architettura per la migrazione del gestionale.
- **SAIPro:** fornisce applicativi per la pianificazione e il controllo della produzione, destinati alle aziende manifatturiere.
- **BI:** Sviluppa soluzioni per la *Business Intelligence<sub>G</sub>* (*BI<sub>G</sub>*), focalizzandosi sull'analisi dei dati aziendali per ottimizzare le *performance* e supportare decisioni strategiche più informate e mirate.
- **CS (Customer Service<sub>G</sub>):** offre supporto ai clienti prima, durante e dopo l'acquisto tramite un ufficio dedicato a SAI e un altro dedicato a SAICon, per garantire assistenza costante.
- **Ufficio sistematico:** si occupa della gestione dell'infrastruttura *hardware*, sia interna all'azienda che per i clienti che richiedono supporto, oltre a fornire assistenza ai *team* di sviluppo.

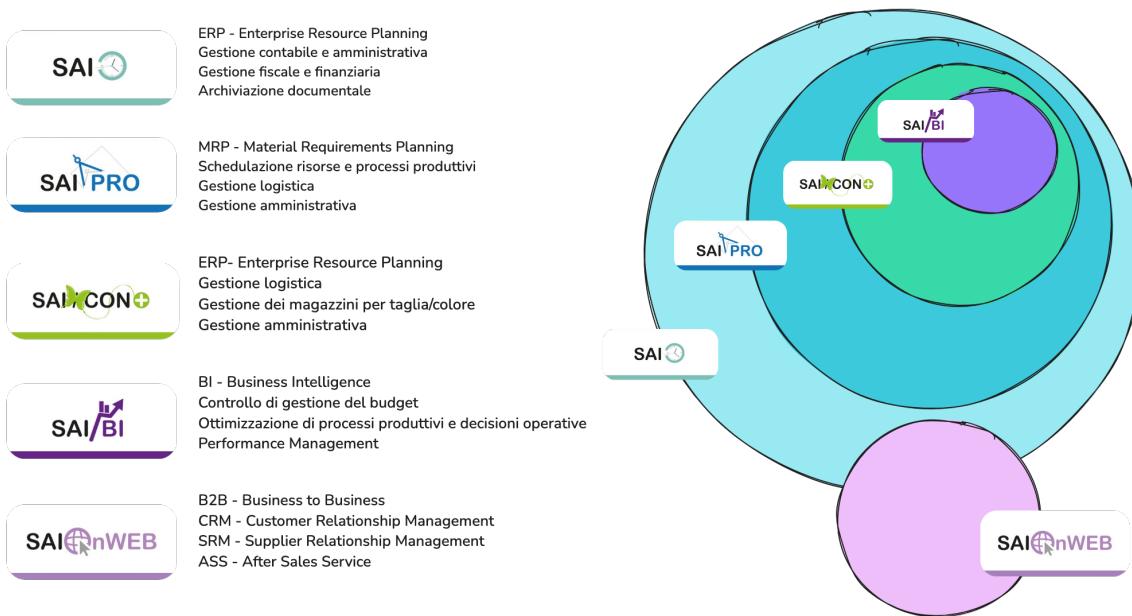


Figura 1.1: Panoramica dei prodotti e servizi offerti da SogeaSoft S.r.l.

SAIonWeb è stato il tema principale del mio *stage*. Il nome SAIonWeb era inizialmente legato al progetto originale, focalizzato esclusivamente su applicativi utilizzabili tramite il *web*. Tuttavia, con l’evoluzione dell’architettura del sistema, il nome è diventato fuorviante. Attualmente il progetto si occupa di sviluppare un’infrastruttura più moderna e flessibile, che possa eventualmente supportare e gradualmente sostituire il gestionale SAI. Questo processo prevede la collaborazione di membri provenienti da diverse aree, tra cui il *team* di SAIcon e quello di SAIPro, che lavorano congiuntamente per integrare i vari componenti del sistema.

Come si può osservare in Figura 1.1, i principali prodotti di SogeaSoft S.r.l. sono strettamente interconnessi, il che si riflette nell’organizzazione dei *team* all’interno dell’azienda. Sebbene esista una divisione indicativa basata sul tipo di prodotto sviluppato, i ruoli risultano essere sfumati e coprono più aree di competenza. Le persone che lavorano su ciascun prodotto fanno riferimento a un *Product Owner*, che può rappresentare più *team* in base alle necessità. Durante il mio *stage* ho avuto modo di interagire con:

- **Product Owner:** è la figura responsabile della definizione delle caratteristiche di un prodotto, della gestione delle priorità e della comunicazione con il *team* di sviluppo, al fine di garantire che il prodotto finale soddisfi le esigenze degli utenti e gli obiettivi aziendali;
- **Team Leader:** gestisce il *team* di sviluppo. In SogeaSoft S.r.l. questa figura può coincidere con il *Product Owner* e/o con il *Scrum Master* (approfondito nella Sezione 1.4);
- **Sviluppatore:** si occupa di progettazione, sviluppo e *testing* del *software*. In base al grado di esperienza contribuisce anche all’analisi e all’assistenza ai clienti.

### 1.3 Prodotti di SogeaSoft S.r.l.

Il prodotto principale di SogeaSoft S.r.l. è un *software* ERP denominato SAI. Costituisce la base per tutti gli altri prodotti sviluppati dall’azienda infatti fanno affidamento diretto su SAI. Nasce come *software* per la gestione della contabilità, poi ampliato con il tempo e adattato

a realtà manifatturiere; ciò ha portato la necessità di introdurre ulteriori funzionalità. Nel contesto del mio stage ho potuto approfondire SAIONWeb e SAIPro che sviluppano le seguenti funzionalità:

- **Material Requirements Planning (MRP)**: pianificazione per determinare i materiali necessari per la produzione, ottimizzando i tempi di approvvigionamento e garantendo la disponibilità delle componenti richieste;
- **Master Production Scheduling (MPS)**: piano di produzione dettagliato per garantire il rispetto degli impegni con i clienti, ottimizzando l'utilizzo delle risorse e coordinando la produzione con la domanda prevista;
- **Capacity Requirements Planning (CRP)**: analisi delle capacità produttive disponibili per verificare che siano adeguate a soddisfare i requisiti stabiliti dal piano di produzione;
- **Finite Capacity Scheduling (FCS)**: pianificazione dettagliata che considera i limiti effettivi delle risorse aziendali, ottimizzando l'allocazione e la sequenza delle attività produttive per massimizzare l'efficienza.

### 1.3.1 Target dell'azienda

SogeaSoft S.r.l. si rivolge principalmente alle Piccole e Medie Imprese (PMI), che spesso presentano la necessità di digitalizzare i propri processi aziendali, richiedendo al contempo un supporto tecnico affidabile e costante.

Le PMI che rappresentano il *target* principale dell'azienda operano prevalentemente nel settore manifatturiero e sono interessate a ottimizzare i propri flussi operativi. Questi includono la gestione e il monitoraggio delle *performance* del personale, la regolazione e l'automazione dei processi logistici, e l'implementazione di soluzioni che migliorino la pianificazione, la produzione e il controllo dei costi.

## 1.4 Modello di sviluppo

SogeaSoft S.r.l. ha adottato un modello di sviluppo software basato sul *framework<sub>G</sub>* Scrum, una metodologia ispirata ai principi dello sviluppo *Agile*. Questo approccio mira a ottimizzare il processo di lavoro rendendolo modulare, adattivo e in grado di rispondere rapidamente ai cambiamenti e alle necessità del cliente e alla complessità delle commesse.

Alla base di Scrum vi è il concetto di *User Story*, una descrizione ad alto livello delle funzionalità attese dal cliente o individuate dal *Product Owner*. Le *User Story*, raccolte nel *Product Backlog*, costituiscono l'insieme di attività prioritarie che guidano lo sviluppo. Questo elenco viene costantemente aggiornato per riflettere nuove necessità o modificare le priorità.

Dopo essere state definite e raffinate, le *User Story* guidano l'organizzazione e la pianificazione dello sviluppo, che avviene attraverso cicli iterativi chiamati *Sprint*. Questo approccio consente di garantire un miglior controllo dei processi e un allineamento costante tra il *team* di sviluppo e gli obiettivi aziendali.

Le principali ceremonie Scrum adottate sono visibili in un diagramma nella Figura 1.2, meglio descritte in seguito:

- **Sprint Planning**: all'inizio di ogni *Sprint*, il gruppo di lavoro partecipa a una sessione di pianificazione per selezionare gli elementi del *backlog* da sviluppare. Gli obiettivi principali dello *Sprint* vengono definiti insieme allo *Sprint Goal*, che rappresenta il risultato principale atteso;

- **Daily Scrum:** ogni giorno, il gruppo di sviluppo e lo *Scrum Master* partecipano a un incontro breve, durante il quale si coordinano le attività e si identificano eventuali ostacoli. Questo garantisce un allineamento costante del gruppo verso gli obiettivi dello *Sprint*;
- **Sprint Review:** alla fine dello *Sprint*, gli incrementi di prodotto sviluppati vengono presentati. Sebbene in teoria questa fase preveda il coinvolgimento degli *stakeholder<sub>G</sub>*, nel caso di SogeaSoft S.r.l. il ruolo di collegamento con il cliente finale viene svolto dal *Product Owner*, il quale raccoglie *feedback* e li traduce in aggiornamenti per il *backlog*;
- **Sprint Retrospective:** questo momento di riflessione sul lavoro svolto è utilizzato dal gruppo di lavoro per identificare opportunità di miglioramento nel processo di sviluppo. Tuttavia, data la dimensione contenuta dei *team* e la consolidata organizzazione del lavoro, questa cerimonia viene svolta solo quando necessario.
- **Raffinamento del Backlog:** un'altra attività fondamentale è quella del Raffinamento, che si tiene con cadenza regolare per preparare gli elementi del *backlog* per i successivi *Sprint*. Durante queste sessioni, il gruppo di lavoro collabora per suddividere le funzionalità più complesse in elementi più piccoli, chiarire i requisiti e definire criteri di accettazione. Questo processo garantisce che gli elementi siano chiari e pronti per essere inclusi nel prossimo *Sprint Planning*. La responsabilità principale di questa attività ricade sul *Product Owner*, con il supporto del *team* di sviluppo.

In sintesi, il modello di sviluppo di SogeaSoft S.r.l. si basa su una gestione iterativa e collaborativa, che consente di rispondere, in un modo che si avvicina al modello *Agile*, alle richieste dei clienti e di mantenere un flusso di lavoro efficace e focalizzato sugli obiettivi aziendali.

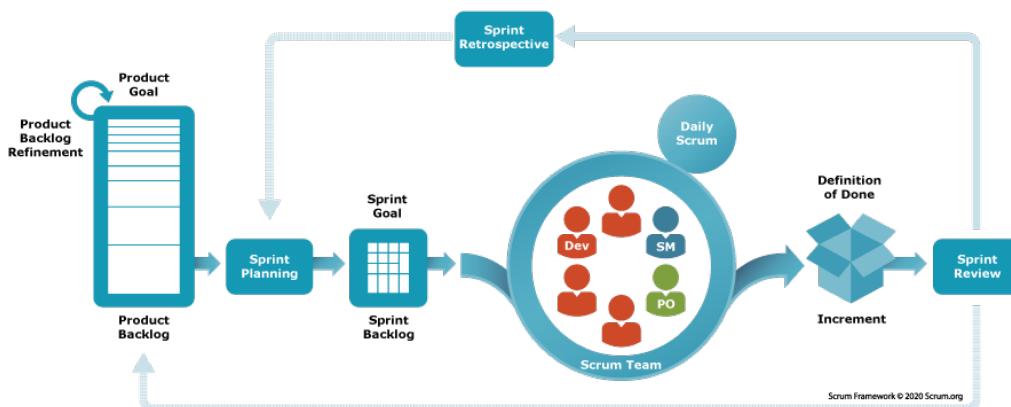


Figura 1.2: Diagramma del flusso di lavoro del modello Scrum, basato sulla filosofia *Agile*.  
Fonte: Documentazione Scrum (*ultimo accesso 3/03/2025*)

## 1.5 Organizzazione interna

SogeaSoft S.r.l. implementa i principi della norma **UNI EN ISO 9001:2015** per garantire un sistema di gestione della qualità dei prodotti efficace.<sup>1</sup>

<sup>1</sup>Fonte: <https://sogeasoft.com/p/iso9001> (*ultimo accesso 3/03/2025*)

Pur non essendo specificamente pensata per lo sviluppo *software*, la norma trova applicazione nell’organizzazione grazie all’adozione di pratiche volte a standardizzare i processi e migliorare l’efficienza operativa. L’azienda basa la propria struttura sull’approccio per processi, suddividendo il lavoro in unità specializzate (come SAI, SAIPro e SAICon) e adottando metodologie *Agile* per la pianificazione e gestione delle attività, ispirandosi allo standard ISO/IEC/IEEE 12207 (*Systems and software engineering - Software life cycle processes*).

Anche i principi dell'approccio per processi e del metodo Plan-Do-Check-Act<sub>G</sub> (*PDCAG*) sono applicati in tutte le fasi operative. Il PDCA si concretizza attraverso la pianificazione accurata delle attività (*Plan*), la loro esecuzione (*Do*), l'analisi dei risultati ottenuti (*Check*) e l'implementazione di eventuali azioni correttive o migliorative (*Act*).

Un esempio dell'applicazione del PDCA è riscontrabile nel modello di sviluppo Scrum, adottato da SogeaSoft S.r.l. per garantire iterazioni rapide e flessibili. Durante gli *Sprint*, il *backlog* viene pianificato e raffinato (*Plan*), le attività vengono eseguite secondo le priorità stabilite (*Do*), e i risultati sono presentati e analizzati attraverso la *Sprint Review* e la *Retrospective* (*Check*). Eventuali miglioramenti vengono quindi incorporati negli *Sprint* successivi (*Act*).

La Figura 1.3 mostra il certificato UNI EN ISO 9001:2015 ottenuto dall'azienda, che conferma l'impegno di SogeaSoft S.r.l. nel mantenere elevati standard di qualità nei propri processi.



Figura 1.3: Certificazione ISO ottenuta da SogeaSoft S.r.l. Fonte: Documentazione pubblica di SogeaSoft S.r.l. (*ultimo accesso 26/02/2025*)

Nelle sezioni successive analizzerò alcuni aspetti fondamentali della gestione del ciclo di vita del *software* all'interno di SogeaSoft S.r.l., con particolare attenzione ai processi di supporto che garantiscono l'efficienza e la qualità dello sviluppo. In particolare approfondirò i processi di cui ho avuto esperienza diretta, come la **Gestione della configurazione** (Sezione 1.5.1),

essenziale per tracciare e controllare le modifiche al codice e alle risorse del sistema; la **Gestione dell'informazione** (Sezione 1.5.2), che disciplina la documentazione e la condivisione delle conoscenze all'interno dell'azienda; e la **Gestione delle risorse umane** (Sezione 1.5.3), fondamentale per il coordinamento dei *team* e la formazione del personale.

### 1.5.1 Gestione della configurazione

Lo scopo del processo di Gestione della configurazione è stabilire e mantenere l'integrità di tutti gli *output* identificati di un progetto o processo e renderli disponibili alle parti interessate.<sup>2</sup>

Nello specifico, un sistema di controllo è importante nell'evoluzione delle funzionalità del *software*, del codice e della documentazione associata perché essi costituiscono gli *output* in questione. Ciò permette di mantenere traccia delle modifiche e garantire che ogni versione sia controllata, riproducibile e coerente con i requisiti stabiliti, facilitando così la gestione delle evoluzioni del *software* e la collaborazione tra i membri del *team*.

In SogeaSoft S.r.l. il controllo delle modifiche al codice avviene attraverso un *Version Control System<sub>G</sub>* (*VCS<sub>G</sub>*), che traccia l'evoluzione del *software* in modo strutturato. Le versioni del codice vengono archiviate all'interno di *repository<sub>G</sub>*, strutture dati appositamente organizzate per gestire i cambiamenti. Queste *repository* si basano su tre principali tipologie di *branch<sub>G</sub>*, ossia rami di sviluppo:

- **master**: è il *branch* principale e stabile, contenente il codice pronto per la produzione. Ogni versione ufficiale del *software* viene rilasciata a partire da questo *branch*;
- **develop**: è il *branch* destinato allo sviluppo continuo, in cui confluiscono le nuove funzionalità e le modifiche prima di essere integrate nel *master*. Rappresenta una versione stabile ma non definitiva del *software*;
- **release**: sono *branch* temporanei creati a partire dal *develop* per preparare un rilascio specifico.

La documentazione viene gestita all'interno di un'area dedicata dell'ambiente di sviluppo adottato (Sezione 1.5.2) da SogeaSoft S.r.l., denominata Wiki. Questo sistema organizza le informazioni in base ad argomenti, capitoli e finalità d'uso, garantendo una struttura relativamente chiara. Inoltre, ogni modifica è accompagnata da un *timestamp* e dall'identificativo dell'autore, permettendo un tracciamento preciso delle revisioni.

### 1.5.2 Gestione dell'informazione

Lo scopo del Processo di Gestione delle Informazioni è garantire alle parti designate l'accesso a informazioni pertinenti, tempestive, complete e valide durante e, ove opportuno, dopo il ciclo di vita del sistema.<sup>3</sup>

SogeaSoft S.r.l. impiega la piattaforma Microsoft Azure sia per la scrittura del codice sia per la gestione della documentazione aziendale. Questo strumento collaborativo facilita l'integrazione con diversi sistemi di supporto ai processi di Gestione della Configurazione, Progettazione, Implementazione e Manutenzione. In particolare, per la documentazione, SogeaSoft S.r.l. utilizza le Wiki (Figura 1.4), che consentono di organizzare e strutturare le informazioni in modo sistematico.

---

<sup>2</sup>ISO/IEC/IEEE 12207:2008, Configuration Management process, par. 6.3.5.

<sup>3</sup>ISO/IEC/IEEE 12207:2008, Information Management process, par. 6.3.6.

The screenshot shows a Microsoft Azure DevOps Wiki page titled "Welcome to Azure DevOps Wiki". The page has a header with "Close" and "Save" buttons. Below the header is a toolbar with various icons. The main content area contains several sections:

- # Introduction**  
TODO: Give a short introduction of your project. Let this section explain the objectives or the motivation behind this project.
- # Getting Started**  
TODO: Guide users through getting your code up and running on their own system. In this section you can talk about:
  1. Installation process
  2. Software dependencies
  3. Latest releases
  4. API references
- # Build and Test**  
TODO: Describe and show how to build your code and run the tests.
- # Contribute**  
TODO: Explain how other users and developers can contribute to make your code better.

At the bottom of the page, there is a note: "If you want to learn more about creating good readme files then refer the following [guidelines] (<https://docs.microsoft.com/en-us/azure/devops/repos/git/create-a-readme?view=azure-devops>). You can also seek inspiration from the below readme files:" followed by a list of links:

- [ASP.NET Core] (<https://github.com/aspnet/Home>)
- [Visual Studio Code] (<https://github.com/Microsoft/vscode>)
- [Chakra Core] (<https://github.com/Microsoft/ChakraCore>)

Figura 1.4: Esempio di Wiki in Microsoft Azure. Fonte: Documentazione Microsoft Azure (*ultimo accesso 1/03/2025*)

Durante il mio *stage*, ho osservato come, data l’interoperabilità del *software* aziendale, risulti fondamentale che la documentazione associata sia formalmente strutturata in documenti accessibili a tutti i membri dei *team*. Tale documentazione segue una gerarchia ben definita, comprendente la registrazione di incontri, requisiti, analisi e scelte progettuali, tutte adeguatamente descritte e motivate.

Tuttavia, questo approccio non è sempre stato adottato in maniera sistematica. Nelle versioni più datate del *software*, la documentazione risultava spesso incompleta o assente, costringendo gli sviluppatori a ricorrere al *reverse engineering* per comprendere il funzionamento del codice. Questa criticità ha generato una forte dipendenza dalle conoscenze dei singoli sviluppatori coinvolti nel processo iniziale, rendendo più complesso l’apporto di modifiche e aggiornamenti successivi.

Da qui deriva la necessità di SogeaSoft S.r.l. di abbandonare gradualmente il vecchio sistema in favore di un nuovo sistema più documentato, decentralizzato e flessibile.

### 1.5.3 Processi di formazione

La gestione delle risorse umane rappresenta un elemento fondamentale per garantire all’organizzazione le competenze necessarie in linea con le esigenze aziendali. Questo processo assicura la disponibilità di personale qualificato e con esperienza, in grado di svolgere le attività del ciclo di vita del *software* e contribuire al raggiungimento degli obiettivi aziendali, di progetto e del cliente.<sup>4</sup>

Durante il mio *stage* ho avuto modo di analizzare le modalità adottate da SogeaSoft S.r.l. per la formazione dei propri dipendenti. L’azienda implementa un approccio articolato, combinando diverse metodologie formative per garantire un apprendimento efficace e continuo. In particolare, le strategie impiegate comprendono:

- 1. Lezioni in presenza:** nel caso in cui l’azienda intenda introdurre nuove tecnologie o apportare modifiche significative ai *software* in uso, vengono organizzati corsi di formazione condotti da esperti del settore;
- 2. Autoapprendimento:** successivamente alle lezioni frontali, i dipendenti sono incoraggiati a integrare e approfondire autonomamente le conoscenze acquisite. Tale processo

<sup>4</sup>ISO/IEC/IEEE 12207:2008, Human Resource Management process, par. 6.3.4.

avviene attraverso la consultazione di libri tecnici, la visione di videolezioni su piattaforme online gratuite e l'analisi di progetti preesistenti affini;

3. **Peer programming:** questa pratica collaborativa prevede il coinvolgimento di due o più programmatori nella scrittura del codice, promuovendo la condivisione di competenze e il miglioramento della qualità del *software*.

Nel corso del mio stage ho applicato prevalentemente le metodologie di **autoapprendimento** e ***peer programming***. Come si può osservare in Figura 1.5, nelle prime settimane la ricerca autonoma di informazioni e il confronto diretto con colleghi più esperti hanno occupato gran parte della mia giornata lavorativa. Tuttavia, con il progressivo consolidamento delle competenze acquisite, dopo circa dieci giorni lavorativi ho potuto ridurre significativamente il tempo dedicato allo studio individuale, limitando il ricorso al *peer programming* ai soli casi in cui si presentassero difficoltà specifiche nello sviluppo del *software*.

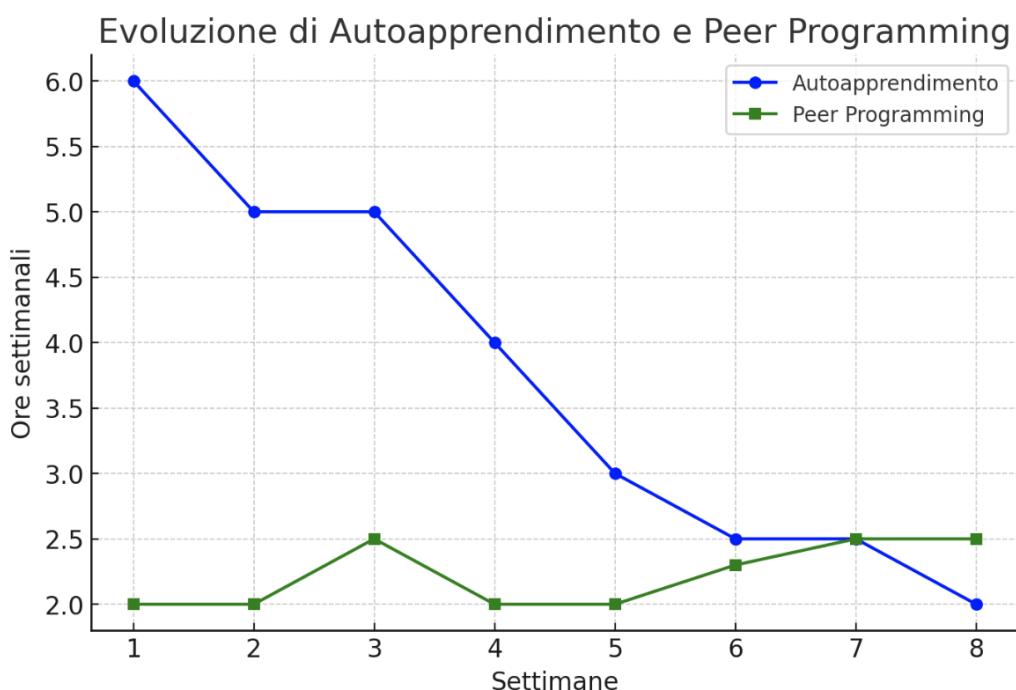


Figura 1.5: Rendiconto ore dedicate ai processi di formazione durante il mio *stage*

## 1.6 Ciclo di vita di un progetto *software*

Il ciclo di vita di un progetto *software* rappresenta l'insieme delle fasi attraverso cui un sistema viene ideato, sviluppato, verificato e mantenuto nel tempo.<sup>5</sup> Nell'ambito dello sviluppo *Agile*, tale processo assume una natura iterativa e incrementale, consentendo una maggiore flessibilità nell'adattamento ai requisiti in evoluzione e nell'ottimizzazione continua del prodotto.

Lo standard ISO/IEC/IEEE 12207 fornisce un quadro di riferimento formale per la gestione del ciclo di vita del *software*, delineando attività strutturate per la pianificazione, lo sviluppo e la manutenzione. Le sezioni successive approfondiranno le principali fasi del ciclo di vita del software nel contesto aziendale di SogeaSoft S.r.l., con particolare riferimento all'esperienza maturata durante il mio *stage*.

---

<sup>5</sup>I. Sommerville, Software Engineering, 10<sup>th</sup> ed., Boston, MA, USA: Pearson, 2015.

### 1.6.1 Analisi preliminare e raccolta dei requisiti

L'analisi dei bisogni degli *stakeholder* e la raccolta dei requisiti rappresentano fasi fondamentali nel ciclo di vita di un progetto *software*, poiché definiscono le basi su cui verrà sviluppato il sistema.<sup>6</sup> Secondo lo standard ISO/IEC/IEEE 12207, tali attività rientrano nel processo di gestione dei requisiti e hanno l'obiettivo di identificare, documentare e validare le esigenze delle parti interessate, garantendo che il *software* finale sia allineato alle aspettative degli utenti.

Gli *stakeholder* di un progetto *software* possono includere clienti, utenti finali, *team* di sviluppo, responsabili di prodotto e altre figure coinvolte nel ciclo di vita del sistema. L'analisi dei bisogni si concentra sull'identificazione delle problematiche esistenti, sulle necessità operative e sugli obiettivi strategici che il *software* deve supportare. Tale attività si avvale di tecniche come interviste, *workshop*, questionari e l'osservazione diretta dei processi aziendali.

Nel contesto dello sviluppo *Agile*, questa fase è dinamica e continua: i bisogni vengono esplorati progressivamente, attraverso interazioni frequenti con gli *stakeholder*. Per favorire la collaborazione costante tra le parti, il *Product Owner* agisce come intermediario tra il *team* di sviluppo e le parti interessate, assicurando che le priorità del prodotto riflettano i reali bisogni dell'azienda.

Una volta analizzati i bisogni, si procede con la definizione e la formalizzazione dei requisiti, ovvero le specifiche funzionali e non funzionali che il *software* dovrà soddisfare. I requisiti funzionali descrivono le capacità e le operazioni del sistema, mentre quelli non funzionali riguardano aspetti come prestazioni, sicurezza, usabilità e scalabilità.<sup>7</sup>

Nello specifico SogeaSoft S.r.l. applica il concetto di *Domain – Driven Design* ( $DDD_G$ ), un approccio che permea l'intero ciclo di vita dello sviluppo *software*, contribuendo a modellare il dominio in maniera coerente e a garantire che il sistema sviluppato risponda in modo preciso e scalabile alle esigenze precedentemente identificate.<sup>8</sup> Una visione d'insieme dell'approccio si può osservare nella Figura 1.8. Nel contesto dell'analisi dei requisiti l'introduzione di DDD comporta diversi passaggi chiave (visualizzabili anche nella Figura 1.6):

- **Collaborazione con gli esperti del dominio (*domain experts*<sub>G</sub>)**: l'elemento distintivo di DDD è il coinvolgimento continuo degli esperti del dominio, che sono coloro che possiedono una conoscenza approfondita del contesto in cui il sistema deve operare. Durante l'analisi dei requisiti, il *team* di sviluppo e gli esperti di dominio lavorano a stretto contatto per assicurarsi che i requisiti non siano solo funzionali, ma riflettano una comprensione profonda delle dinamiche del *business*.
- **Creazione di un linguaggio comune (*Ubiquitous Language*<sub>G</sub>)**: uno degli aspetti centrali di DDD è l'adozione di un linguaggio comune, che consente a tutte le parti coinvolte nel progetto (sia tecniche che non) di discutere in modo chiaro e preciso i concetti chiave del dominio. Questo linguaggio deve essere utilizzato sin dalla fase di analisi dei requisiti per evitare ambiguità e garantire che tutti i requisiti siano ben definiti.
- **Definizione di *Bounded Context*<sub>G</sub>**: un altro concetto fondamentale di DDD è la divisione del dominio in *Bounded Contexts*. Durante l'analisi dei requisiti, i *team* identificano e definiscono questi contesti limitati, ciascuno con il proprio modello di dominio, che può evolvere in modo indipendente dagli altri. Questo aiuta a evitare conflitti tra requisiti di diverse parti del sistema e facilita la gestione della complessità, permettendo di concentrarsi su una parte specifica del dominio alla volta.

<sup>6</sup>ISO/IEC/IEEE 12207:2008, Stakeholder Requirements Definition Process, par. 6.4.1.

<sup>7</sup>ISO/IEC/IEEE 12207:2008, System Requirements Analysis Process, par 6.4.2.

<sup>8</sup>E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003

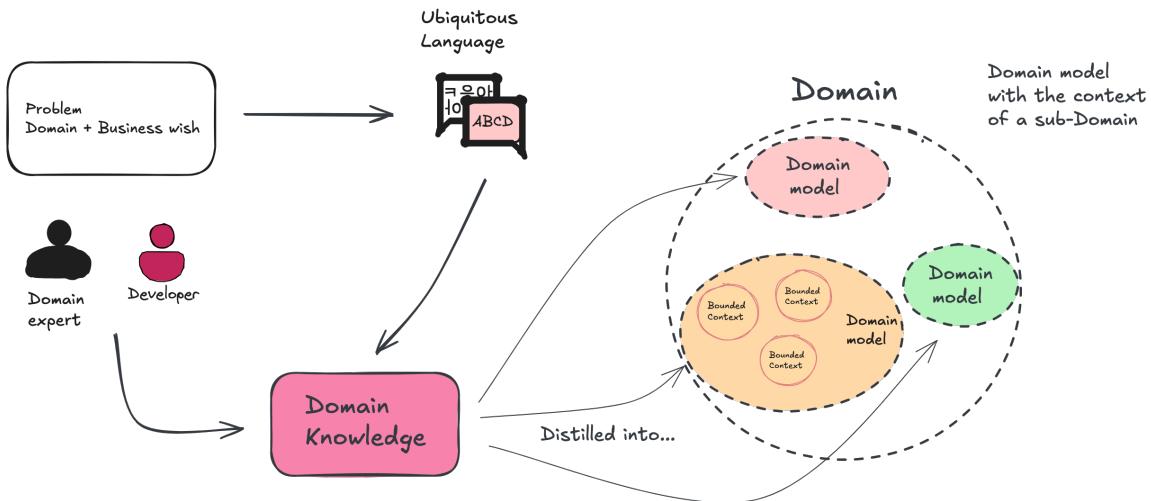


Figura 1.6: Visualizzazione del processo di Analisi dei Requisiti nel contesto DDD.

In conclusione, l'integrazione del *Domain-Driven Design* nell'analisi dei requisiti consente di sviluppare un modello del dominio solido e coerente, facilitando la comunicazione tra gli *stakeholder* e il *team* di sviluppo. L'adozione di un linguaggio comune e la definizione di *Bounded Contexts* contribuiscono a ridurre le ambiguità e a migliorare l'allineamento tra le esigenze di business e le soluzioni *software*.

## 1.6.2 Progettazione

Il processo di Progettazione *software* rappresenta una fase centrale del ciclo di vita dello sviluppo, in cui vengono definite l'architettura, i componenti e le interazioni del sistema al fine di garantire un'implementazione efficiente e conforme ai requisiti precedentemente raccolti.<sup>9</sup>

Come previsto da Scrum (Sezione 1.4), io e il *team* di sviluppo abbiamo iterato le attività di analisi e progettazione, con l'obiettivo di raggiungere delle soluzioni che potessero soddisfare i bisogni del cliente.

In particolare, con l'approccio del *Domain-Driven Design*, introdotto nella Sezione 1.6.1, abbiamo attuato le seguenti attività visibili anche in Figura 1.7:

- **Modellazione del dominio:** dopo aver raccolto i requisiti, la fase di progettazione è dove l'approccio DDD inizia a entrare in gioco più attivamente. Utilizzando il linguaggio comune e i concetti emersi durante l'analisi dei requisiti, io e il *team* di sviluppo abbiamo costruito un modello di dominio dettagliato. In questa fase abbiamo definito gli *aggregati<sub>G</sub>*, le *entità<sub>G</sub>*, i *value objects<sub>G</sub>* e le funzioni di dominio, tutti elementi di progettazione nel contesto DDD.
- **Definizione degli eventuali microservizi<sub>G</sub>:** data la natura del progetto di *stage*, l'approccio DDD ci ha aiutati a individuare delle unità autonome grazie ai *Bounded Contexts* precedentemente identificati. Questo processo verrà approfondito nella Sezione 2.
- **Individuazione di pattern<sub>G</sub> di progettazione:** durante la fase di progettazione, l'approccio DDD impiega diversi *pattern* architettonici e di *design* per affrontare le problematiche comuni che emergono nella costruzione di sistemi complessi.<sup>10</sup> Questi *pattern* ci hanno permesso di trovare modi per gestire in modo efficace la comunicazione tra le varie parti del sistema per poi poterle implementare nella fase successiva.

<sup>9</sup>ISO/IEC/IEEE 12207:2008, System Architectural Design Process, par 6.4.3.

<sup>10</sup>E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003

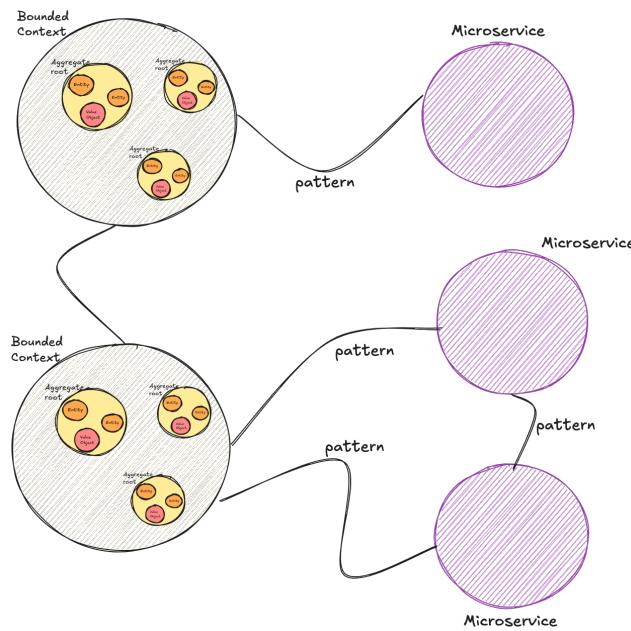


Figura 1.7: Schema dell’attività di Progettazione nel contesto DDD

### 1.6.3 Implementazione

L’attività di Implementazione costituisce una fase essenziale nel ciclo di vita del *software*, in cui la soluzione progettata viene tradotta in codice eseguibile, rendendola concretamente fruibile dagli utenti finali. Secondo lo standard ISO/IEC/IEEE 12207, l’implementazione rientra nei processi primari di sviluppo e comprende la scrittura, la verifica e l’integrazione del codice, garantendo che il prodotto software soddisfi i requisiti definiti nelle fasi precedenti.<sup>11</sup>

Per la gestione e il monitoraggio delle attività di codifica, l’azienda SogeaSoft S.r.l. utilizza Microsoft Azure, una piattaforma che integra un *Issue Tracking System<sub>G</sub>* (*ITS<sub>G</sub>*). Questo strumento consente di gestire e tracciare le attività di sviluppo, offrendo una visione olistica del progetto sia dal punto di vista gestionale che implementativo.

Come è possibile osservare nella Figura 1.8, nel contesto del *Domain-Driven Design* (DDD) l’implementazione segue principi mirati a garantire la coerenza tra il modello concettuale del dominio e la struttura del codice. In particolare, la fase di sviluppo prevede la codifica del modello di dominio. Quest’ultimo, elaborato durante la fase di Progettazione (Sezione 1.6.2), viene tradotto in codice attraverso l’implementazione delle componenti individuate nella fase precedente. Tali componenti vengono strutturate in modo da rispettare le regole di business definite in precedenza, utilizzando il linguaggio comune (*Ubiquitous Language*) condiviso tra gli *stakeholder* per assicurare consistenza semantica.<sup>12</sup>

All’interno di SogeaSoft S.r.l., l’attività di sviluppo è organizzata mediante un sistema di *task assignment*, in cui ogni attività (detta anche *issue*) viene assegnata a uno sviluppatore specifico. Questo approccio favorisce un’elevata *ownership* del lavoro, responsabilizzando il singolo sviluppatore e ottimizzando la gestione delle risorse.

A seconda del livello di urgenza e priorità, il Team Leader può collocare l’attività di implementazione all’interno del *Backlog* specifico dello *Sprint*, qualora si tratti di una lavorazione prioritaria, oppure nel *Product Backlog*, in attesa di essere raffinata e pianificata nei successivi incontri di Raffinamento del *backlog*.

<sup>11</sup>ISO/IEC/IEEE 12207:2008, Software Implementation Processes, par.7.1.

<sup>12</sup>E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003

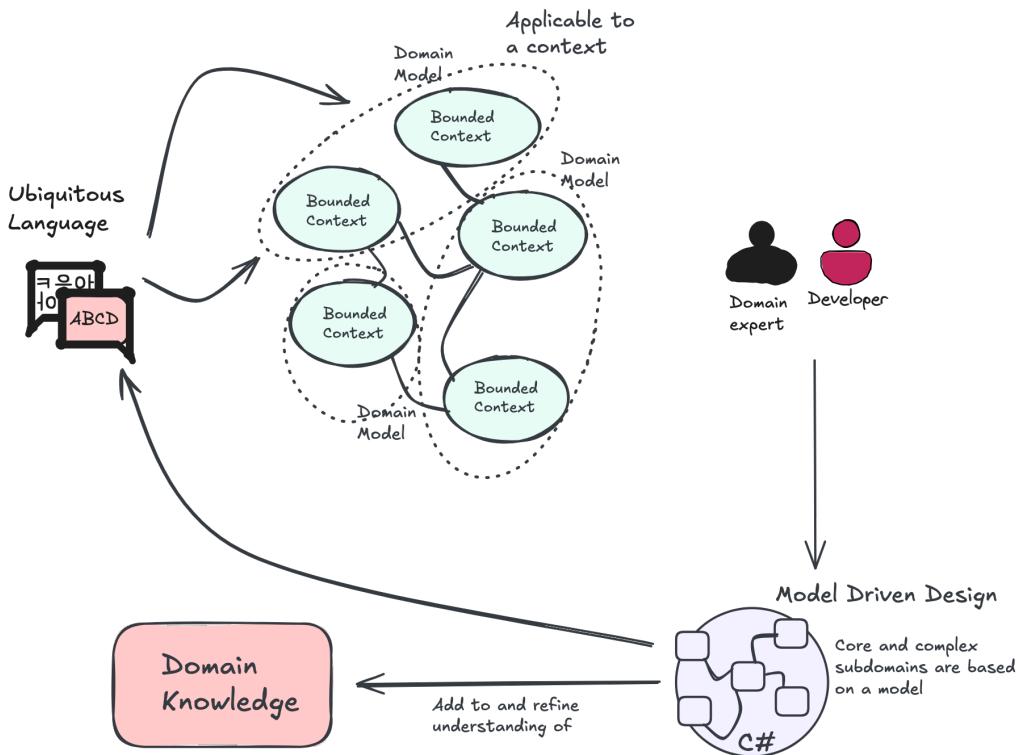


Figura 1.8: Come DDD permea il ciclo di vita di un *software*.

Lo sviluppo del codice avviene secondo una strategia di *Version Control* strutturata: per ogni attività assegnata, viene creato un *branch* dedicato a partire dal ramo di sviluppo principale (*develop*). Tale *branch* viene nominato seguendo una convenzione specifica, includendo parole chiave come "*feature<sub>G</sub>*" (nuova funzionalità) o "*bug<sub>G</sub>-fix*" (correzione di errori), seguite da una breve descrizione dell'attività, al fine di garantire una chiara identificazione e tracciabilità delle modifiche.

Una volta completata la fase di codifica, il codice prodotto è sottoposto a un processo di Verifica e Validazione (Sezione 1.6.4), il quale si concretizza nell'apertura di una *Pull Request<sub>G</sub>* (*PR<sub>G</sub>*). Tramite la PR, lo sviluppatore richiede una revisione formale delle modifiche da parte del *Team Leader* o di un altro membro del *team* con pari competenze. Solo a seguito della validazione, il codice viene integrato nella *codebase* principale attraverso un'operazione di *merge* nel *repository* del progetto, garantendo così il mantenimento della qualità del *software* e la coerenza dell'intero sistema.

#### 1.6.4 Verifica e validazione

Il processo di Verifica ha l'obiettivo di fornire evidenza oggettiva della capacità del *software* di soddisfare i requisiti e le caratteristiche definite.<sup>13</sup> Durante il mio periodo di *stage*, ho avuto l'opportunità di assistere da vicino all'esecuzione delle attività legate a questo processo.

La verifica del *software* si svolge in più fasi, partendo dalle singole unità di codice fino ad estendersi al sistema nel suo complesso. Le principali tipologie di *test* utilizzate sono le seguenti:

- **Test di unità:** verificano le singole unità di codice, ovvero porzioni atomiche ed eseguibili che espongono un comportamento specifico. Nello specifico ho avuto occasione di sviluppare *test* specifici, progettati per verificare il soddisfacimento dei requisiti identificati;

<sup>13</sup>ISO/IEC/IEEE 12207:2008 Software Verification Process, par 7.2.4.

- **Test di integrazione:** questi *test* verificano il comportamento e l’interazione tra le diverse parti del sistema, assicurandosi che il *software* rispetti i requisiti funzionali previsti;
- **Test di sistema:** questi test mirano a verificare la conformità del *software* nel suo insieme, considerando le dipendenze tra le varie componenti.

In generale SogeaSoft S.r.l. adotta un processo di Verifica strutturato, seppur con una metodologia flessibile. La scrittura di *test* non è obbligatoria per la presentazione e la validazione del *software*. Tuttavia, quando presenti, i test vengono sviluppati direttamente all’interno dell’ambiente Microsoft Azure, sfruttando le sue *pipeline*<sub>G</sub>. In questo ambiente, il codice viene compilato e i *test* vengono eseguiti automaticamente. Questi test vengono eseguiti sulle *Pull Requests* (PR) nei *branch* designati, garantendo che le modifiche apportate non introducano errori nel sistema.

Il processo di Validazione invece fornisce evidenza oggettiva sulla capacità del *software* di soddisfare le aspettative e i bisogni del committente.<sup>14</sup>

In SogeaSoft S.r.l. la validazione delle PR è un processo collaborativo che coinvolge più sviluppatori: i colleghi svolgono una revisione del codice e, salvo casi particolari, la validazione richiede l’approvazione di due revisori che non siano gli sviluppatori stessi delle modifiche. Inoltre, secondo il principio del *Domain-Driven Design* (DDD), la validazione dovrebbe coinvolgere anche il *domain expert*, poiché è colui che possiede una conoscenza approfondita del dominio applicativo e rappresenta il principale fruitore del servizio.

Nello specifico, la validazione dei risultati delle attività da me svolte è avvenuta attraverso incontri dedicati, in cui il lavoro completato è stato presentato e discusso con l’intero *team*, durante due *Sprint Review*. Questo approccio ha consentito non solo di rendere visibile a tutti, inclusi il *Product Owner* e i membri del *team* di sviluppo, il progresso e la qualità del lavoro, ma anche di allineare le attività agli obiettivi e ai requisiti stabiliti. Inoltre, tali incontri hanno offerto l’opportunità di identificare e risolvere eventuali problematiche emerse, assicurando che il progetto proseguisse in maniera coerente e conforme alle aspettative degli *stakeholder*.

### 1.6.5 Manutenzione

Lo scopo del processo di Manutenzione del *software* è fornire un supporto a un prodotto *software* già consegnato. Questo processo include una serie di attività destinate a mantenere il software operativo e adeguato alle esigenze in continua evoluzione degli utenti e dell’ambiente. L’obiettivo principale è assicurare che il prodotto mantenga la sua funzionalità, affidabilità e performance nel tempo, minimizzando i costi associati dal momento della distribuzione fino al suo ritiro.<sup>15</sup>

Il *Domain-Driven Design* (DDD) permea anche questa fase del ciclo di vita del *software*. Dopo la messa in produzione, il modello di dominio potrebbe evolversi in risposta all’emergere di nuove esigenze o informazioni. Grazie alla natura iterativa e incrementale di DDD, il modello di dominio può essere continuamente aggiornato e migliorato attraverso il *feedback* degli utenti o la scoperta di nuove dimensioni del dominio, attuando una manutenzione adattiva. Inoltre, DDD facilita l’adattamento ai cambiamenti delle esigenze aziendali, mantenendo il sistema allineato con gli obiettivi del *business*, applicando una manutenzione preventiva. Durante la fase di manutenzione, è possibile rivedere i *Bounded Contexts* e le comunicazioni tra i modelli di dominio, migliorando così la separazione delle preoccupazioni e garantendo che il *software* continui a rispondere in modo adeguato alle evoluzioni del contesto operativo e strategico. Tale processo è rappresentato nel suo insieme nella Figura 1.9 nella pagina successiva:

---

<sup>14</sup>ISO/IEC/IEEE 12207:2008 Software Validation Process, par 7.2.5.

<sup>15</sup>ISO/IEC/IEEE 12207:2008, Software Maintenance Process, par 6.4.10.

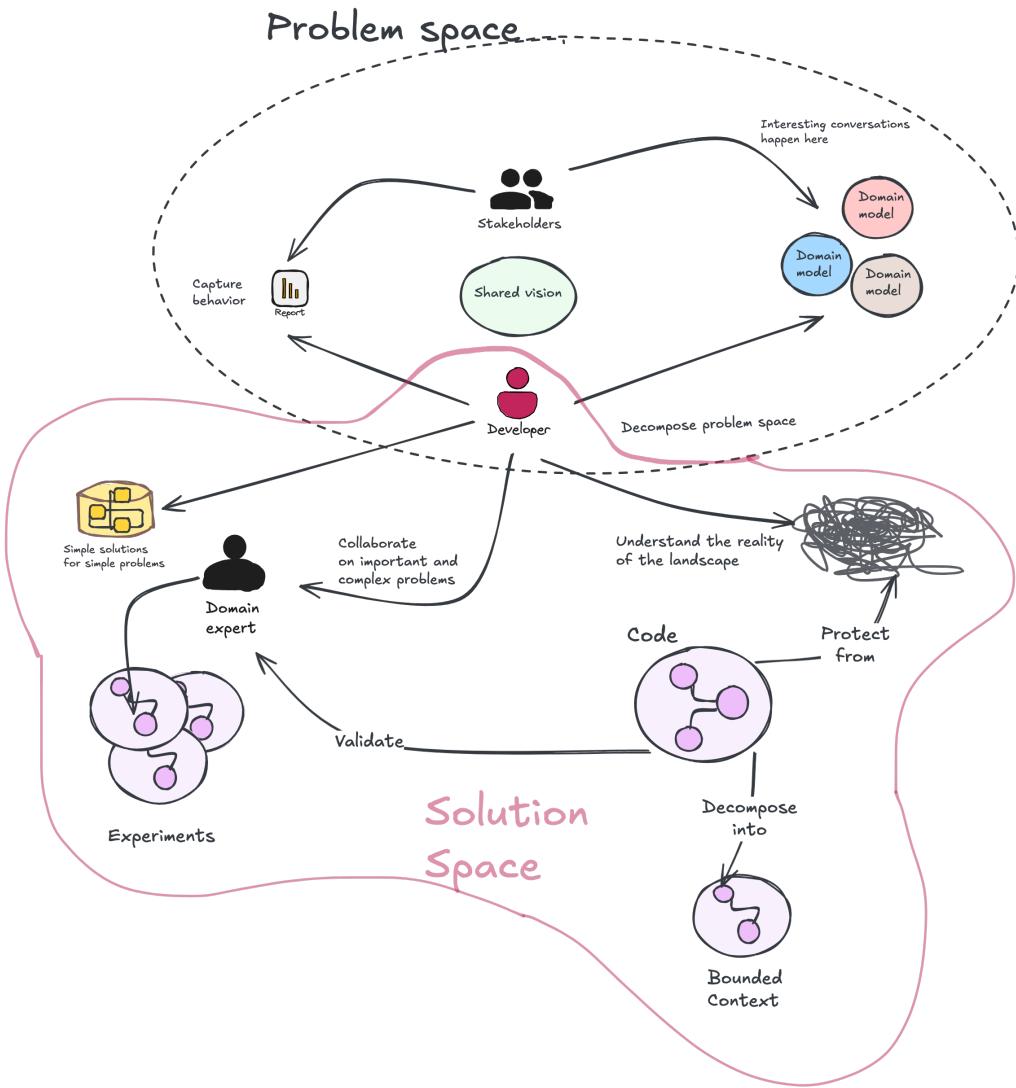


Figura 1.9: Visione d'insieme dell'utilizzo di DDD nello sviluppo *software*.

## 1.7 Tecnologie utilizzate

Per la realizzazione dei propri prodotti, SogeaSoft S.r.l. adotta un insieme di tecnologie predefinito e condiviso tra tutti i *software* sviluppati. Tuttavia, tali tecnologie si sono progressivamente evolute per garantire l'aggiornamento rispetto agli sviluppi del settore. L'obiettivo dell'azienda è implementare un'architettura *esagonale<sub>G</sub>* nello sviluppo del *software* SAI e delle sue declinazioni, ossia un modello di progettazione che consente la sostituzione o l'aggiornamento delle tecnologie senza alterare il comportamento del sistema sottostante, che rimane tecnologicamente agnostico.

Come si può osservare in Figura 1.10, durante il mio *stage*, una parte del processo di formazione (Sezione 1.5.3) è stata dedicata allo studio dell'evoluzione del sistema SAI, dal suo primo rilascio fino ad oggi, al fine di comprendere in modo più approfondito gli obiettivi del mio progetto. L'analisi delle tecnologie adottate in passato è rilevante perché una parte del progetto prevede l'aggiornamento del sistema tramite l'implementazione di soluzioni tecnologiche più moderne, garantendone la continuità operativa e l'efficienza.

I linguaggi di programmazione utilizzati sono:

- **C++**: è un linguaggio di programmazione orientato agli oggetti molto diffuso. In questo caso, è particolarmente apprezzato per la sua efficienza nell'elaborazione di operazioni ad

alte prestazioni, la gestione diretta della memoria e la capacità di sviluppare applicazioni di sistema complesse. È alla base di tutto il sistema SAI;

- **C#**: è un linguaggio orientato agli oggetti sviluppato da Microsoft, progettato per la creazione di applicazioni su piattaforme come Windows, *web* e dispositivi mobili. È utilizzato in SAIonWeb, oggetto del mio progetto di *stage*.

Questi linguaggi di programmazione sono utilizzati specificamente per lo sviluppo delle applicazioni basate sulla piattaforma SAI, mentre le tecnologie di supporto, che forniscono il fondamento per il funzionamento e la gestione del prodotto, comprendono una serie di strumenti e *framework* integrati. Tali tecnologie sono:

- **Qt**: *framework* multipiattaforma utilizzato principalmente per lo sviluppo di applicazioni con interfaccia grafica, alla base del sistema SAI.
- **KDE**: è un ambiente *desktop open-source* per sistemi operativi Linux. Offre un’interfaccia grafica altamente personalizzabile, con un focus sull’usabilità e sull’integrazione di applicazioni. Il suo *framework* di sviluppo è Qt .
- **Angular**: è un *framework open-source* per lo sviluppo di *single-page web application<sub>G</sub>* ossia siti *web* composti da una sola pagina che aggiorna dinamicamente il contenuto senza ricaricare l’intera pagina. Con il suo approccio basato su componenti, Angular promuove lo sviluppo modulare e scalabile, motivo per cui è stato scelto da SogeaSoft S.r.l. in SAIonWeb.
- **ASP.NET Core**: è un *framework open-source* per lo sviluppo di applicazioni *web* moderne e scalabili. Permette di creare applicazioni *web*, *API<sub>G</sub>* e microservizi, supportando C#.

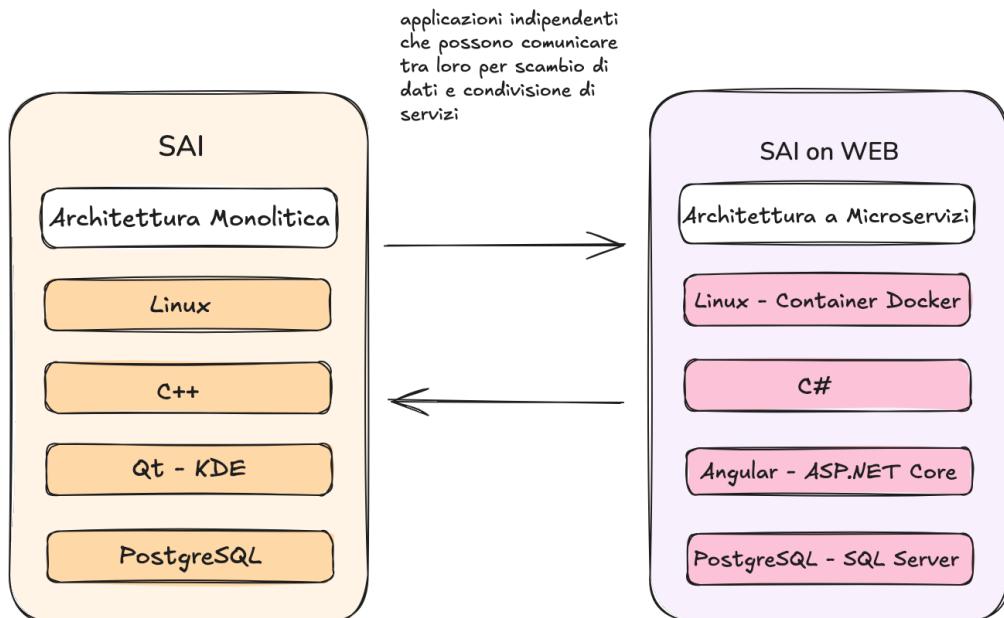


Figura 1.10: Schema delle tecnologie utilizzate da SogeaSoft S.r.l. Informazioni apprese durante il processo di formazione

L’ambiente di sviluppo in SogeaSoft S.r.l. è configurato per operare principalmente su sistemi basati su Linux o, in alternativa, su *container Docker<sub>G</sub>* eseguiti su sistemi operativi Windows.

Questo approccio consente una maggiore flessibilità e compatibilità con diverse configurazioni di sistema. Il processo di codifica si svolge localmente sulle singole *workstation*, dove gli sviluppatori utilizzano Microsoft Visual Studio, un ambiente di sviluppo integrato (IDE) altamente configurabile e supportato.

Il tracciamento delle modifiche apportate al codice secondo il controllo di versione avviene con l'uso di *Git<sub>G</sub>*, un particolare *Version Control System* (VCS).

SAI è il *framework* di base su cui si fondano tutti i prodotti derivati di SogeaSoft S.r.l. Si tratta di un sistema piuttosto datato, che utilizza tecnologie oggi difficili da manutenere, da cui nasce la necessità di migrare verso un'architettura a microservizi. La logica sottostante risulta complessa da modificare, pertanto sono state adottate diverse soluzioni. Ad esempio, il sistema impiega un'architettura basata su *web service*, che adotta il modello *Representational State Transfer<sub>G</sub>* (*REST<sub>G</sub>*) per lo scambio di dati tra i moduli. Questo modello architettonico prevede che le informazioni transitino tramite connessioni dedicate, note come *Application Programming Interfaces<sub>G</sub>* (*API<sub>G</sub>*). Tuttavia, tale modello non definisce l'intero sistema, poiché la fase di migrazione è ancora in corso e rappresenta il *focus* principale del mio progetto di *stage*, come verrà approfondito nella Sezione 2.

Nel dettaglio si utilizzano:

- **Advanced Message Queuing Protocol<sub>G</sub> (AMQP<sub>G</sub>)**: è un protocollo di messaggistica *open-source* progettato per la gestione affidabile e sicura delle code di messaggi tra sistemi distribuiti. È utilizzato da SogeaSoft S.r.l. per facilitare l'integrazione tra le applicazioni e la gestione dei flussi di dati.
- **RabbitMQ**: è un *MessageBroker<sub>G</sub>* *open-source* che implementa il protocollo AMQP per la gestione di code di messaggi tra applicazioni. Permette una comunicazione asincrona e affidabile tra sistemi distribuiti.
- **Debezium**: è una piattaforma *open-source* per il cambiamento di dati in tempo reale che consente di monitorare e registrare le modifiche apportate ai database. Essa si integra con sistemi di messaggistica come RabbitMQ per diffondere le modifiche ai dati attraverso flussi di eventi.
- **Swagger**: è uno strumento che fornisce un'interfaccia grafica interattiva per esplorare e testare le API RESTful. Permette agli sviluppatori di testare *endpoint<sub>G</sub>* API direttamente dal *browser<sub>G</sub>* senza bisogno di scrivere codice aggiuntivo.
- **DBeaver**: *client<sub>G</sub>* utilizzato per accedere, interrogare e manipolare *database* relazionali basati sul *Structured Query Language<sub>G</sub>* (*SQL<sub>G</sub>*), un linguaggio di manipolazione dei dati ampiamente diffuso.

Riguardo ai sistemi di gestione dei *database*, SogeaSoft S.r.l. supporta una varietà di *Database Management Systems<sub>G</sub>* (*DBMS<sub>G</sub>*), offrendo così una notevole flessibilità nella gestione e nell'archiviazione dei dati. Durante il mio *stage*, ho avuto l'opportunità di interagire con diversi DBSM, tra cui PostgreSQL, SQL Server e IBM DB2, che sono tra le soluzioni più comunemente utilizzate dalle aziende clienti.

## 1.8 L'innovazione in SogeaSoft S.r.l.

Le informazioni sulle innovazioni tecnologiche adottate da SogeaSoft S.r.l. mi sono state fornite principalmente tramite i racconti del mio *tutor*, che ha condiviso con me le pratiche aziendali

e i progetti più recenti. Non essendo disponibili dati documentati diretti o misurazioni quantitative, mi sono basata sulla descrizione dei processi e delle soluzioni adottate durante il mio periodo di *stage*.

Sebbene l'azienda non abbia intrapreso l'adozione di innovazioni radicali, la sua capacità di evolversi in modo continuo e sostenibile le ha consentito di mantenere una posizione competitiva nel mercato. L'approccio adottato, che prevede l'allocazione di risorse umane e finanziarie in base alle necessità percepite, ha favorito l'implementazione di miglioramenti incrementali, garantendo così una continua crescita della capacità produttiva e un aggiornamento regolare delle tecnologie impiegate.

Tuttavia, SogeaSoft S.r.l. ha considerato la migrazione verso un'architettura a microservizi solo quando il processo di manutenzione del *software* (Sezione 1.6.5) è diventato eccessivamente oneroso. Questo è avvenuto a causa della crescente difficoltà di gestire le tecnologie obsolete, che richiedevano modifiche strutturali complete del sistema, una soluzione che si è rivelata non praticabile nel lungo periodo.

Una volta che l'azienda ha riconosciuto l'importanza di integrare nuove tecnologie pur mantenendo intatto il sistema di base, si è manifestato un crescente interesse per l'innovazione. Ad esempio, durante il mio *stage*, ho avuto l'opportunità di implementare una *demonstration* utilizzando la tecnologia RabbitMQ (approfondita nella Sezione 1.7), in cui ho sviluppato un sistema di aggiornamento dei dati in tempo reale. Questa *demo* ha comportato l'utilizzo di tecnologie relativamente recenti e, nel caso di Debezium (Sezione 1.7), non ufficialmente documentate per questa combinazione specifica<sup>16</sup>.

---

<sup>16</sup>Fonente: Documentazione ufficiale di Debezium, Source Connectors.

# 2 Progetto di *stage*

## 2.1 Gestione degli *stage* in SogeaSoft S.r.l.

SogeaSoft S.r.l. riconosce il valore strategico degli *stage* curricolari, considerandoli un'opportunità sia per la formazione di potenziali nuovi dipendenti, sia per l'esplorazione di nuove tecnologie e la valutazione critica dei sistemi attualmente in uso. Tali percorsi formativi consentono non solo di trasferire conoscenze e competenze, ma anche di promuovere un'analisi approfondita delle soluzioni tecnologiche adottate dall'azienda, favorendo l'innovazione e l'ottimizzazione dei processi.

Da ciò che ho potuto comprendere durante il mio periodo a SogeaSoft S.r.l., le attività svolte nell'ambito degli *stage* possono includere:

- l'integrazione di nuove funzionalità nei sistemi esistenti, con l'obiettivo di migliorarne le prestazioni e l'efficienza;
- lo studio e lo sviluppo di strumenti autonomi a supporto dei processi di sviluppo o dei prodotti in uso, come ad esempio Swagger, una piattaforma per la documentazione e il *testing* delle API (approfondite nella Sezione 1.7);
- l'analisi formale di strumenti già utilizzati in azienda, ma impiegati prevalentemente in modo empirico, al fine di standardizzarne e ottimizzarne l'uso;
- la conduzione di attività di monitoraggio sulle *performance* di determinati sistemi, per identificarne eventuali criticità e proporre soluzioni migliorative.

L'evoluzione tecnologica in SogeaSoft S.r.l. può avvenire attraverso diverse strategie, tra cui l'ampliamento delle funzionalità della *codebase* esistente, l'analisi teorica dello stato dell'arte o la realizzazione di *software* sperimentali, quali *Proof of Concept<sub>G</sub>* (*PoC<sub>G</sub>*) o prodotti già pronti per un utilizzo immediato, come il *Minimum Viable Product<sub>G</sub>* (*MVP<sub>G</sub>*). Nel contesto del mio *stage*, le attività svolte hanno combinato questi approcci, consentendo lo sviluppo di una soluzione concreta ma non immediatamente utilizzabile (ossia un prototipo<sub>G</sub>, la cui differenza è visibile nella Figura 2.1); nonché la produzione di una documentazione tecnica approfondita.

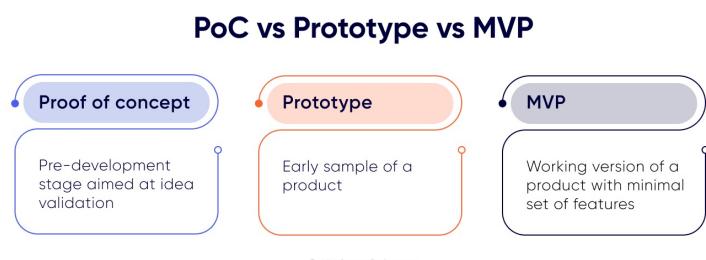


Figura 2.1: Differenza tra PoC, Prototipo, MVP. Fonte: <https://clockwise.software/blog/proof-of-concept-in-software-development/> (ultimo accesso 12/03/2025)

SogeaSoft S.r.l. attribuisce particolare valore agli *stage*, poiché rappresentano un'opportunità strategica per affrontare una delle sue principali sfide tecnologiche: la migrazione del proprio prodotto da un'architettura *monolitica<sub>G</sub>* a un'architettura a *microservizi<sub>G</sub>*, come discusso nella Sezione 1.8. Gli *stage* costituiscono una risorsa vantaggiosa sotto molteplici aspetti: da un lato, permettono di ottimizzare l'investimento in ricerca e sviluppo grazie a costi contenuti; dall'altro, consentono all'azienda di entrare in contatto con prospettive innovative, idee originali e persone non condizionate da paradigmi consolidati.

Un ulteriore fattore determinante nell'impiego di tirocinanti per lo sviluppo di soluzioni innovative è la gestione delle risorse interne. Il personale aziendale è prevalentemente impegnato nel mantenimento e nell'evoluzione dei sistemi attualmente in produzione, rendendo complesso il reindirizzamento delle competenze su progetti sperimentali. L'inserimento di studenti permette di destinare risorse dedicate a iniziative di ricerca e innovazione, permettendo al contempo un processo di trasferimento di conoscenze tra le diverse generazioni di sviluppatori.

## 2.2 Il *software* SAIonWeb

Il tema principale del mio *stage* ha riguardato lo sviluppo e l'evoluzione di SAIonWeb, un *software* basato sul *framework* SAI (Sistema Aziendale Integrato). SAIonWeb è uno strumento progettato per la gestione dei processi aziendali nell'ambito dell'*Enterprise Resource Planning* (ERP). In particolare, il *software* è stato concepito per rispondere alle esigenze specifiche del settore manifatturiero, nello specifico all'industria dell'abbigliamento, offrendo supporto integrato all'intero ciclo di vita del prodotto: dalla gestione e approvvigionamento delle materie prime, attraverso la fase di confezionamento e produzione, fino alla distribuzione e commercializzazione finale<sup>1</sup>.

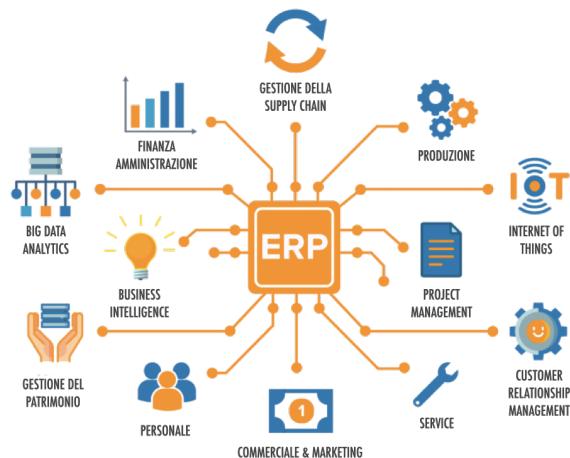


Figura 2.2: Esempio di un ERP generico. Fonte:<https://www.bo.cam-com.gov.it/it/promozione-interna/sistemi-gestionali-erp> (ultima visita 15/03/2025)

Come anticipato nella Sezione 1.2, SAIonWeb costituisce uno strumento di elevata rilevanza strategica, implementato da diverse realtà imprenditoriali operanti nel settore manifatturiero, con l'obiettivo di automatizzare e ottimizzare molteplici processi operativi aziendali. I processi specifici supportati dal sistema e le relative implicazioni funzionali sono oggetto di approfondimento nella sezione successiva.

<sup>1</sup>Fonte: <https://sogeasoft.com/p/sai> (ultima visita 12/03/2025)

## 2.2.1 Funzionalità generali di SAIONWeb

SAI ERP (e dunque anche SAIONWeb) è un sistema *software* gestionale progettato per integrare e automatizzare i processi aziendali, consentendo alle organizzazioni di gestire in modo efficiente risorse, dati e operazioni. Questi sistemi offrono un ambiente centralizzato in cui le diverse aree aziendali, dalla produzione alla logistica, dalla contabilità alla gestione delle risorse umane, possono operare in modo coordinato, migliorando la tracciabilità delle informazioni e ottimizzando la produttività.

Le sue funzionalità principali includono:

- **gestione delle materie prime:** monitoraggio degli approvvigionamenti, delle scorte e della qualità dei materiali;
- **pianificazione della produzione:** organizzazione delle fasi produttive, assegnazione delle risorse e gestione dei tempi di lavorazione;
- **tracciabilità dei prodotti:** controllo dell'avanzamento di ciascun capo, dalla fase iniziale di lavorazione fino alla distribuzione. In particolare, una schermata d'esempio è visibile in Figura 2.3;
- **gestione degli ordini e delle vendite:** registrazione degli ordini, gestione delle consegne e fatturazione automatizzata;
- **logistica e distribuzione:** coordinamento delle spedizioni, gestione dei magazzini e ottimizzazione delle scorte;
- **integrazione con il sistema contabile:** gestione di pagamenti, bilanci e rendicontazione finanziaria;
- **monitoraggio delle performance aziendali:** generazione di *report* analitici e strumenti di *business intelligence* per supportare il processo decisionale.

Attraverso queste funzionalità, SAI e SAIONWeb consentono alle aziende del settore moda di migliorare la gestione delle operazioni, ridurre i tempi di produzione e garantire una maggiore efficienza operativa.

The screenshot shows a software interface for managing production cycles. On the left, a sidebar menu lists various modules: Mappa produzione, Articoli, Componenti, Cicli, Ordini, Tabelle, Checklist, Tracciabilità, and Statistiche. A vertical navigation bar on the far left lists steps from 1 to 8: Articoli, Dati generali, Fasi del ciclo, Attività, Dettaglio attività, Postazioni assegnate, Configurazione postazioni, and Stato. The main panel is titled 'CICLI / MODIFICA CICLO' and shows a cycle for 'Valvola 42'. It displays four activities: 001 (Preparazione delle macchine e delle attrezzature), 002 (Lavorazione dei materiali), 003 (Assemblaggio), and 004 (Test funzionale preliminare). Below the activities is a table with columns: Codice, Descrizione, Tipologia, and Stato. The table contains three rows with data: 0001 (Taglio, Lavoro, Configurato), 0002 (Formatura, Lavoro, Configurato), and 0003 (Lavorazione dei materiali, Lavoro, Configurato). Buttons for 'Duplica ciclo' and 'Chiudi' are at the top right, and a search bar with filter options is at the bottom right.

Figura 2.3: Schermata del *software* SAI. Fonte: *slide* di presentazione utilizzate durante la fase di formazione

## 2.2.2 L'architettura di SAI

### Il monolite

Il *software* SAI è caratterizzato da un'architettura monolitica, termine che evoca l'idea di una struttura compatta e unitaria, in cui le varie componenti sono strettamente interconnesse e formano un insieme indivisibile. Nel contesto dell'architettura *software*, tale espressione indica un sistema in cui tutte le funzionalità sono integrate in un'unica unità, costituendo una struttura unificata in cui ciascun componente dipende dagli altri per garantire il corretto funzionamento dell'intero sistema<sup>2</sup>.

Le principali caratteristiche di un'architettura monolitica sono le seguenti:

- **integrazione delle funzionalità in un unico blocco applicativo:** tutte le funzionalità sono raccolte in un'unica entità, che viene distribuita come un blocco indivisibile. Questo implica che ogni modifica apportata a una parte del codice può avere ripercussioni su altre parti del sistema, rendendo complessi gli aggiornamenti e la manutenzione. Di conseguenza, con l'evolversi del progetto, l'accumulo di debito tecnico<sup>3</sup> può far sì che il *software* diventi eccessivamente complesso e difficilmente adattabile a nuove tecnologie<sup>3</sup>;
- **scalabilità verticale:** la scalabilità in un sistema monolitico si ottiene potenziando l'intera applicazione per affrontare un aumento del carico di lavoro<sup>3</sup>. Anche qualora solo alcune funzionalità necessitassero di risorse aggiuntive, l'intero sistema deve essere aggiornato, comportando uno spreco di risorse. Non essendo sempre possibile una scalabilità modulare, si tende a duplicare l'intera applicazione per garantire le prestazioni richieste;
- **bassa granularità<sub>G</sub>:** l'architettura monolitica presenta una scarsa separazione delle funzionalità, che risultano strettamente integrate e difficilmente isolabili<sup>4</sup>. Questo limita significativamente la flessibilità del sistema e ostacola l'integrazione di nuovi moduli o tecnologie senza compromettere la stabilità complessiva;
- **database centralizzato:** ciò implica che tutte le componenti dell'applicazione accedono e manipolano direttamente gli stessi dati. Questo comporta un **forte accoppiamento (*tightly coupled*)<sup>5</sup>** tra i moduli e rende complessi gli aggiornamenti o le modifiche al database, poiché qualsiasi cambiamento può avere ripercussioni sull'intero sistema.

L'architettura monolitica del sistema SAI rappresenta una sfida significativa in termini di manutenibilità e adattabilità alle tecnologie moderne. Tale struttura, caratterizzata da un'elevata interdipendenza tra i componenti, limita la capacità di evoluzione e di risposta ai cambiamenti del contesto operativo. Questa situazione evidenzia la necessità di intraprendere un percorso di migrazione verso un'architettura a microservizi, che consentirebbe di migliorare la modularità del sistema attraverso componenti indipendenti e specializzati.

---

<sup>2</sup>O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," International Journal of Computer Science and Information Security, vol. 17, no. 3, pp. 123-131, 2019.

<sup>3</sup>J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From Monolith to Microservices: A Classification of Refactoring Approaches," in Proceedings of the IEEE International Conference on Software Architecture (ICSA), Stuttgart, Germany, 2018.

<sup>4</sup>M. Cojocaru, A. Uta, and A.-M. Oprescu, "MicroValid: A Validation Framework for Automatically Decomposed Microservices," in Proceedings of the 11th IEEE/ACM International Conference on Cloud Computing (CloudCom), December 2019.

<sup>5</sup>Fonete: <https://martinfowler.com/articles/break-monolith-into-microservices.html> (ultima visita 14/03/2025)

## I microservizi

Al contrario, un'architettura basata su microservizi presenta caratteristiche distintive che la differenziano in modo significativo da quella monolitica, garantendo maggiore flessibilità e modularità.

Riprendendo i punti descritti nella precedente sezione, segue una descrizione delle principali caratteristiche di questa architettura:

- **funzionalità distribuite in più *codebase*:** le funzionalità del sistema sono organizzate in servizi separati, ognuno dei quali dispone di una propria *codebase* autonoma<sup>6</sup>. I microservizi comunicano tra loro tramite reti, spesso utilizzando API, mantenendo un elevato grado di indipendenza;
- **scalabilità orizzontale:** è possibile aumentare le risorse relative a una singola funzionalità semplicemente aggiungendo istanze del microservizio corrispondente<sup>7</sup>, senza dover stratificare l'intera applicazione. In un contesto di *Domain-Driven Design* (DDD), si parla di **evoluzione del dominio**, ossia della capacità di migliorare e adattare il sistema in funzione dei cambiamenti del dominio applicativo<sup>8</sup>, evitando di creare monoliti stratificati e complessi da gestire (come ad esempio SAI);
- **alta granularità:** l'architettura a microservizi è caratterizzata da un'elevata granularità<sup>9</sup>, in cui ogni servizio è dedicato a una specifica funzionalità e opera in modo autonomo. Questa separazione netta consente di mantenere una chiara distinzione tra le responsabilità dei vari servizi, agevolando la manutenibilità e l'evoluzione del sistema. Il DDD supporta questa fase attraverso la definizione di **Bounded Contexts** e l'applicazione del principio di **Separation of Concerns**<sup>9</sup>, che garantiscono una chiara demarcazione tra i diversi ambiti di competenza e funzionalità.
- **database distribuito:** i microservizi dispongono di database indipendenti<sup>10</sup>. Ogni microservizio è responsabile della gestione dei propri dati e, qualora necessiti di informazioni da un altro servizio, effettua una richiesta esplicita tramite meccanismi di comunicazione inter-servizio, spesso implementati tramite **Domain Events**<sup>9</sup>. Questo approccio riduce il forte accoppiamento (*loose coupling*)<sup>11</sup> tra le componenti, garantendo maggiore modularità e resilienza del sistema.

Queste caratteristiche rendono l'architettura a microservizi particolarmente adatta a contesti dinamici e complessi, in cui la capacità di adattamento e la modularità rivestono un ruolo fondamentale nel garantire la qualità e la manutenibilità del sistema. Nella pratica applicativa, tuttavia, tali caratteristiche ideali non sempre trovano piena realizzazione. I sistemi reali presentano spesso complessità intrinseche che rendono difficoltosa una netta separazione delle funzionalità, mentre vincoli di natura economica, temporale o tecnica possono limitare la possibilità di effettuare migrazioni complete verso questa architettura.

<sup>6</sup>O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," International Journal of Computer Science and Information Security, vol. 17, no. 3, pp. 123-131, 2019.

<sup>7</sup>Fonte: <https://dev.to/somadevtoo/horizontal-scaling-vs-vertical-scaling-in-system-design-3n09> (ultima visita: 14/03/2024)

<sup>8</sup>E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003

<sup>9</sup>M. Cojocaru, A. Uta, and A.-M. Oprescu, "MicroValid: A Validation Framework for Automatically Decomposed Microservices," in Proceedings of the 11th IEEE/ACM International Conference on Cloud Computing(CloudCom), December 2019

<sup>10</sup>S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, O'Reilly Media, 2019.

<sup>11</sup>Fonte:<https://martinfowler.com/articles/break-monolith-into-microservices.html> (ultima visita: 14/03/2025)

L'implementazione dei microservizi richiede pertanto un approccio pragmatico che consideri attentamente il contesto specifico, valutando il rapporto costi-benefici delle diverse soluzioni possibili. Questo implica accettare l'inevitabile assenza di soluzioni perfette e adottare strategie incrementali che consentano di bilanciare gli obiettivi architetturali con i vincoli operativi.

L'obiettivo del mio *stage* è quello di fornire una base formale e metodologica al lavoro empirico già avviato da SogeaSoft S.r.l. riguardo alla migrazione verso un'architettura a microservizi, con l'intento di superare i limiti strutturali del sistema attuale e di garantire una maggiore flessibilità e scalabilità in futuro. Una rappresentazione visuale di questa attività è visibile nella Figura 2.4.

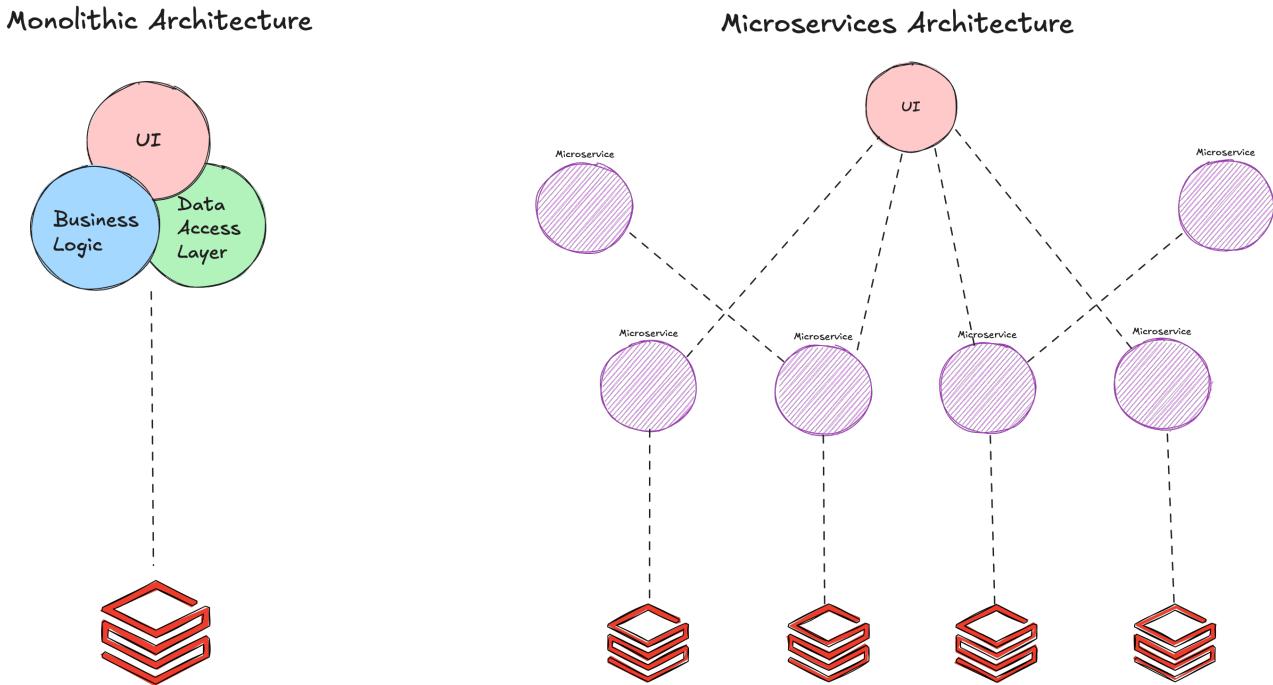


Figura 2.4: Rappresentazione grafica della suddivisione in microservizi

### 2.2.3 La migrazione

SogeaSoft S.r.l. desidera dunque migrare da un'architettura monolitica a un'architettura a microservizi. Già diverse attività sono state attuate a questo proposito e il mio compito è stato quello di porre una base teorica più solida ed eventualmente confermare o confutare le scelte implementate. Inoltre ho avuto l'occasione di partecipare ai processi decisionali riguardo a particolari casi d'uso nel mio progetto specifico di *stage*.

Come si può osservare nella Figura 2.5, uno degli elementi centrali della migrazione riguarda l'introduzione di una componente denominata MicroService-Middleware, implementata come un *Anti-Corruption Layer<sub>G</sub>* (*ACL<sub>G</sub>*). In ambito architettonico, un ACL è un modello progettuale che funge da interfaccia tra sistemi *legacy<sub>G</sub>* e nuove componenti, prevenendo la propagazione di modelli obsoleti o incoerenti nelle parti più moderne del sistema. Nel contesto di SogeaSoft S.r.l., il MicroService-Middleware si occupa di ricevere le richieste provenienti dall'ERP monolitico e, tramite un *Message Broker<sub>G</sub>*, instradare tali richieste verso il microservizio appropriato. Il *Message Broker*, infatti, svolge il ruolo di intermediario per lo scambio di messaggi tra componenti *software*, garantendo una comunicazione asincrona ed efficiente tra i vari microservizi.

Ogni microservizio, una volta attivato, dopo aver effettuato l'autenticazione attraverso l'applicazione nativa di SAI chiamata *Identity Server*, esso esegue la funzione richiesta in modo

autonomo e indipendente, restituendo l'esito tramite lo stesso meccanismo di messaggistica. Per rendere accessibili le funzionalità esposte dai microservizi, l'azienda ha implementato una *WebAPI Gateway*, che rappresenta un punto di accesso centralizzato alle API dei vari servizi. La *WebAPI Gateway* consente di aggregare e orchestrare le chiamate ai microservizi, creando un'interfaccia unificata verso l'utente finale.

Grazie a questa architettura, il sistema è in grado di supportare una *Single Page Application<sub>G</sub>* (SPA), un tipo di applicazione *web* che carica una singola pagina *web* e aggiorna dinamicamente il contenuto man mano che l'utente interagisce. Questo approccio garantisce un'esperienza utente più fluida e interattiva, riducendo i tempi di caricamento e ottimizzando le *performance* dell'applicazione.

L'adozione di queste soluzioni riflette la volontà dell'azienda di superare le limitazioni dell'architettura monolitica, garantendo maggiore modularità e adattabilità alle nuove tecnologie, pur mantenendo la continuità dei servizi già in essere.

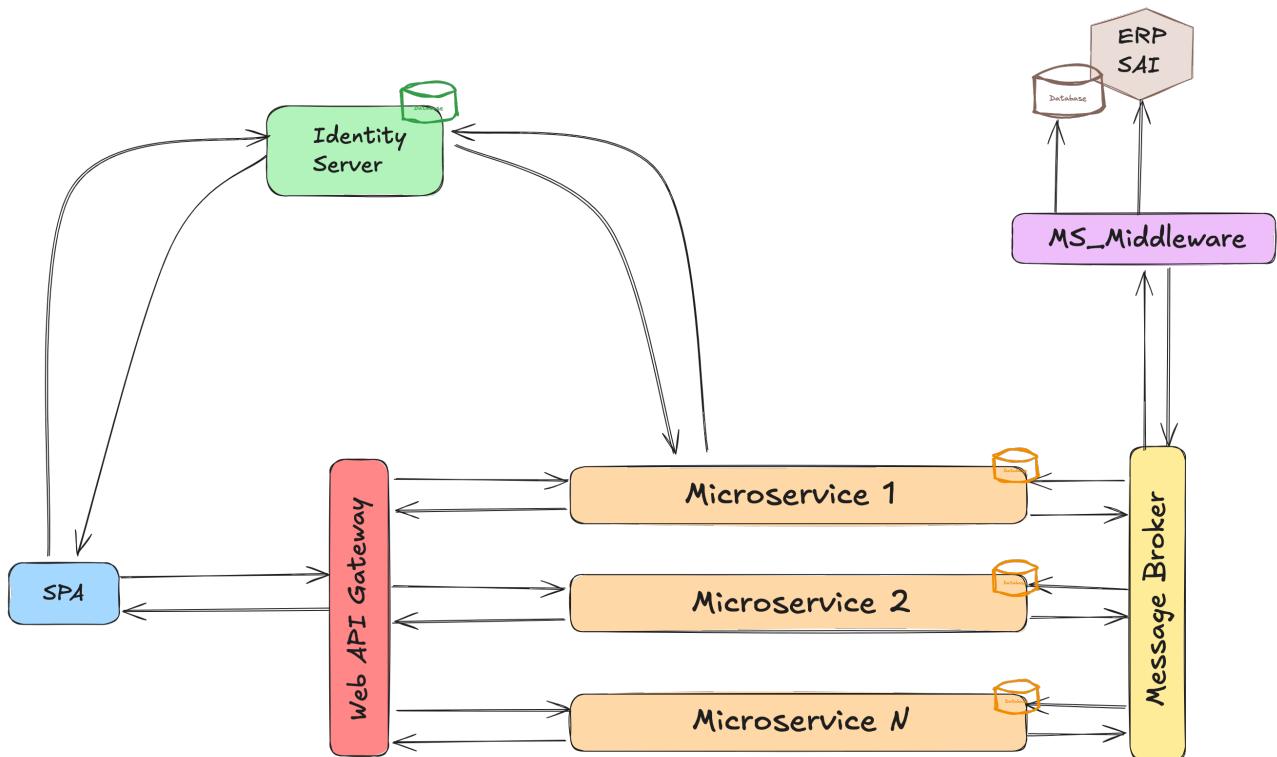


Figura 2.5: Piano di migrazione di SogeaSoft S.r.l.. Fonte: informazioni apprese durante il processo di formazione.

## 2.3 Obiettivi del progetto di *stage*

### 2.3.1 Obiettivi aziendali

Nel contesto del mio progetto di *stage*, SogeaSoft S.r.l. ha deciso di intraprendere un percorso di approfondimento riguardante la suddivisione del monolite in microservizi. Tale scelta si inserisce nell'ottica di favorire una maggiore modularità e flessibilità del sistema, nonché di facilitare futuri interventi di manutenzione e aggiornamento tecnologico.

Considerando che l'obiettivo dello *stage* è dimostrare concretamente le competenze acquisite, si è ritenuto opportuno effettuare l'estrazione di un microservizio come risultato tangibile dell'attività svolta.

A supporto dell'intero progetto, il *tutor* aziendale ha redatto un Piano di lavoro, un documento esaustivo che descrive l'azienda, gli obiettivi progettuali, i vincoli eventualmente presenti, la metodologia adottata e i risultati attesi. Tale documento costituisce un riferimento fondamentale per garantire la coerenza e la chiarezza delle attività intraprese durante lo *stage*.

<b>Obiettivi aziendali</b>	
<b>Obbligatori</b>	
<b>OB1</b>	Studio della letteratura esistente sulle architetture monolitiche, sulle architetture a microservizi e sui metodi di migrazione
<b>OB2</b>	Documentazione relativa ai requisiti
<b>OB3</b>	Documentazione dei servizi esistenti e delle relazioni tra essi
<b>OB4</b>	Individuazione di un piano indicativo
<b>Desiderabili</b>	
<b>DE1</b>	Documentazione dei rischi
<b>Facoltativi</b>	
<b>FA1</b>	Realizzazione del PoC

Tabella 2.1: Descrizione degli obiettivi aziendali per il progetto di *stage*; corrispondono agli obiettivi redatti dal *tutor* aziendale nel Piano di lavoro.

Gli obiettivi aziendali riportati nella Tabella 2.1 sono classificati secondo la seguente notazione:

- **Obiettivo obbligatorio (OB)**: rappresentano i requisiti obbligatori, vincolanti, che dovranno necessariamente essere soddisfatti;
- **Obiettivo desiderabile (DE)**: rappresentano i requisiti desiderabili, non vincolanti, ma dal riconoscibile valore aggiunto;
- **Obiettivo facoltativo (FA)**: rappresentano i requisiti facoltativi, rappresentanti di valore aggiunto ma non strettamente competitivo.

La rappresentazione tabellare utilizza la notazione **[sigla][identificativo]**, dove la sigla indica la tipologia di obiettivo secondo la classificazione sopra descritta, mentre l'identificativo è un numero progressivo che garantisce l'univocità del requisito in combinazione con la sigla.

### 2.3.2 Vincoli

Per quanto concerne i **vincoli temporali**, lo *stage* curricolare deve svolgersi nell'arco di un monte ore compreso tra 300 e 320 ore, come previsto dal corso di studi, al fine di garantire un carico di lavoro equivalente a 12 crediti formativi universitari. Nel caso specifico del mio percorso formativo, tali ore sono state distribuite su un periodo di 8 settimane, con un impegno settimanale di 40 ore, articolate dal lunedì al venerdì, dalle ore 8:30 alle ore 17:00.

Per quanto riguarda i **vincoli tecnologici**, il piano di lavoro ha indicato i seguenti strumenti e metodologie:

- linguaggio di sviluppo: C#;
- ambiente di sviluppo: Visual Studio, Visual Studio Code;
- metodologia di sviluppo: Scrum (nell'ambito del *framework Agile*) e *DevOpsG*, un insieme di pratiche e strumenti che integrano lo sviluppo *software* (Dev) e le operazioni (Ops) volte alla gestione delle infrastrutture tecnologiche.

Tuttavia, nel corso dello *stage* ho avuto l'opportunità di interagire anche con numerose altre tecnologie, che ho approfondito nella Sezione 1.7.

## 2.4 Metodo di lavoro

### 2.4.1 Pianificazione

Sulla base degli obiettivi di progetto delineati nella Sezione 2.3.1 (Tabella 2.1) e della pianificazione oraria proposta nel Piano di lavoro (Tabella 2.2), insieme al *tutor* aziendale, abbiamo stabilito le principali tappe intermedie (*milestone*) per monitorare il progresso del progetto. La collocazione temporale di queste *milestone* è stata determinata tenendo conto sia delle attività che dipendono strettamente l'una dall'altra, in particolare per quanto riguarda gli aspetti formativi, sia della necessità di produrre risultati tangibili, quali documentazione e codice in modo regolare. Ciò ha permesso al *tutor* di monitorare il progresso del progetto e individuare tempestivamente eventuali rallentamenti.

La definizione dettagliata delle singole attività è avvenuta nel corso delle sessioni di *Sprint planning* con il *Product Owner*. Durante questi incontri, mi venivano assegnate le attività specifiche, venivano chiariti i risultati attesi e venivano fissati appuntamenti per valutare il grado di avanzamento, fornire supporto in caso di difficoltà o testare i risultati raggiunti.

Durata in ore	Descrizione attività
16	Conoscenza ambiente di lavoro e contesto
32	Studio della letteratura esistente sulle architetture monolitiche, sulle architetture a microservizi e sui metodi di migrazione
24	Analisi dell'applicazione esistente per identificare i servizi e le funzionalità da suddividere in microservizi
16	Stesura della documentazione relativa ai requisiti
24	Studio dell'architettura a microservizi già esistente e analisi dei possibili miglioramenti per facilitarne la migrazione
32	Identificazione dei servizi monolitici da decomporre e delle relazioni tra essi.

Continua nella pagina successiva

Durata in ore	Descrizione attività
<b>24</b>	Documentazione dei servizi esistenti e delle relazioni tra essi e del piano
<b>16</b>	Identificazione dei potenziali rischi e delle sfide associate alla migrazione
<b>24</b>	Inizio dello sviluppo di un Proof of Concept
<b>16</b>	Documentazione dei rischi
<b>8</b>	Individuazione di un piano indicativo di migrazione
<b>24</b>	Proseguimento del Proof of Concept
<b>16</b>	Valutazione teorica dell'architettura
<b>32</b>	Revisione della documentazione e del Proof of Concept
<b>304 ore totali</b>	

Tabella 2.2: Ripartizione delle ore in base alle attività

### 2.4.2 Modello di sviluppo

SogeaSoft S.r.l. adotta un modello di sviluppo *Agile* basato su Scrum, come approfondito nella Sezione 1.4. Tuttavia, durante il mio stage, non ho partecipato a tutte le ceremonie previste dal framework Scrum. In particolare, sebbene il *Daily Scrum meeting* si svolgesse quotidianamente con il mio *tutor* (che si alternava al *Product Owner*, in base alla disponibilità e alle necessità di ciascuno), il mio coinvolgimento nelle altre ceremonie è stato limitato. Infatti, ho partecipato esclusivamente allo *Sprint planning* e alla *Sprint review*, che non avevano una durata fissa, ma variavano a seconda dei *task* assegnati e della disponibilità del *Product Owner*. Nel complesso, il mio lavoro è stato principalmente solitario, con sviluppo individuale e il supporto del *tutor* limitato a momenti specifici. Uno schema utile per visualizzare il processo che ha caratterizzato le attività di sviluppo durante il mio *stage* è visibile in Figura 2.6.

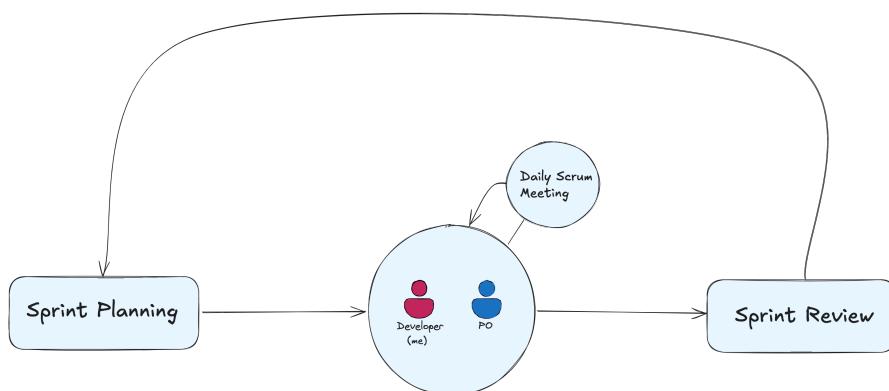


Figura 2.6: Rappresentazione grafica dell'attività di sviluppo effettiva che ha caratterizzato il mio *stage* secondo *Agile*, nel contesto Scrum.

### 2.4.3 Strumenti

Gli strumenti utilizzati durante il mio *stage*, oltre alle tecnologie approfondite nella Sezione 1.7, sono stati diversificati e funzionali a diversi aspetti del lavoro. In primo luogo, ho impiegato la documentazione che redigevo progressivamente, utilizzando la *Wiki* di Azure come base di lavoro. Inoltre, per facilitare la gestione delle attività quotidiane e il processo di redazione della tesi, ho creato dei documenti personali in cui annotavo le cose da fare e le note importanti.

Ogni sessione di *Sprint planning* è stata accompagnata dalla definizione dettagliata dei *task* da parte del *Product Owner*, al fine di evitare ambiguità nell'interpretazione delle attività. Questi *task* sono stati organizzati e conservati in una cartella Drive, dove ho anche archiviato libri, articoli e altre fonti utili trovate durante il progetto.

Poiché le pianificazioni erano gestite in modo intuitivo, non è stato necessario un ricorso frequente al calendario, in quanto il *Product Owner* mi comunicava direttamente le date degli incontri successivi. Un ulteriore strumento di monitoraggio utile è stato il *report* settimanale, che inviavo al mio *tutor* interno. Questi *report* mi hanno permesso di tracciare lo stato di avanzamento del progetto, identificare eventuali rallentamenti o problematiche, e di valutare se fosse opportuno rivedere le priorità tra le attività in corso.

### 2.4.4 Revisioni di progresso

Durante l'intero periodo di *stage*, le interazioni con il *tutor* aziendale sono state caratterizzate da una costante frequenza e prossimità. La vicinanza fisica delle postazioni lavorative ha notevolmente facilitato le opportunità di confronto professionale, contribuendo a mitigare la naturale difficoltà nell'avanzare richieste di supporto. Ciononostante, uno dei primi *feedback* che ho ricevuto ha riguardato la necessità di incrementare la frequenza delle richieste di assistenza, evitando un'eccessiva ostinazione nella ricerca autonoma di soluzioni, e di migliorare la comunicazione complessiva, sia in merito ai progressi conseguiti che alle criticità riscontrate.

In aggiunta agli incontri giornalieri individuali (*Daily meeting*), che si svolgevano successivamente alle riunioni del *tutor* relative ai suoi progetti, ho sistematicamente beneficiato dell'opportunità di comunicare l'avanzamento delle attività o le problematiche incontrate, senza la necessità di attendere la successiva sessione programmata.

Le valutazioni formali dei progressi, condotte con l'altro *Product Owner*, avvenivano nell'ambito degli *Sprint Review*, durante i quali questo verificava l'efficacia funzionale del lavoro svolto. In due circostanze distinte ho presentato il prototipo sviluppato all'intero *team* di sviluppo, considerata l'immediata applicabilità dello stesso. Tali presentazioni hanno costituito di fatto l'illustrazione del prodotto finale. La seconda occasione ha riguardato lo sviluppo di una dimostrazione funzionale (PoC) relativa a una funzionalità di particolare interesse: la sincronizzazione dei dati in tempo reale.

## 2.5 Motivazioni della scelta

La selezione dell'opportunità di *stage* è stata guidata da una riflessione evolutiva sulla natura e sugli obiettivi formativi di tale esperienza. Inizialmente, consideravo lo *stage* prevalentemente come un'occasione di applicazione pratica delle conoscenze teoriche acquisite durante il percorso accademico. Tuttavia, la partecipazione all'evento STAGE-IT, tenutosi a Padova nell'aprile 2024, ha determinato un ampliamento di tale prospettiva, conducendomi a pensare l'esperienza di *stage* anche come un'opportunità di sviluppo professionale e personale nonché di confronto con sfide innovative.

L’architettura a microservizi, approfondita teoricamente durante il corso di Ingegneria del *Software*, aveva suscitato in me un particolare interesse, che non aveva trovato adeguate opportunità di esplorazione pratica nell’ambito dei progetti didattici. L’individuazione di una proposta di *stage* focalizzata su tale paradigma architettonico ha rappresentato pertanto un’opportunità particolarmente rilevante per il mio percorso formativo. La reciproca soddisfazione nella selezione è stata ulteriormente confermata dall’interesse manifestato dall’azienda per la mia candidatura, considerata la limitata attrattività del progetto tra gli altri studenti.

Sottolineo che la decisione è stata principalmente determinata dall’interesse per la tematica progettuale, piuttosto che da una preesistente conoscenza o affinità con l’azienda ospitante. La natura manifatturiera di SogeaSoft S.r.l. non ha costituito un elemento determinante nella selezione, poiché un’attività di tipo artigianale sarebbe risultata maggiormente in linea con i miei ideali, in contrapposizione all’ottimizzazione della produzione industriale su larga scala. Inoltre, è emerso come l’interesse principale dell’azienda sia di natura economica, senza una significativa considerazione per la sostenibilità o l’impatto ambientale. Sebbene sia vero che il miglioramento dell’efficienza possa talvolta contribuire alla riduzione degli sprechi, qualora ciò sia avvenuto si tratta di un esito accidentale piuttosto che intenzionale.

### 2.5.1 Obiettivi e aspettative personali

La scelta dello *stage* è stata dunque guidata da motivazioni personali, in particolare dalla curiosità verso l’argomento specifico dei microservizi.

L’esperienza precedente con il progetto di Ingegneria del *Software* mi ha permesso di identificare alcuni aspetti personali da migliorare e argomenti tecnici da approfondire. Tra questi, il mio interesse per i microservizi, che ho descritto in dettaglio nella sezione 2.5, e per le API come metodi di comunicazione tra microservizi. Durante il progetto universitario, non ho avuto l’opportunità di esplorare adeguatamente questi temi, sia per una non ottimale organizzazione del gruppo di lavoro, sia per una distribuzione inefficiente delle attività. Mi sono ritrovata infatti a dedicare tempo ad attività poco significative (*shallow work*) a discapito di altre più formative, tra cui l’approfondimento di queste tecnologie che producono reale valore, che rappresentano un esempio di attività di *deep work*<sup>12</sup>.

Questa esperienza mi ha fatto comprendere come la scarsa fiducia nelle mie capacità e la difficoltà nel comunicare apertamente (sia le difficoltà che i progressi) mi abbiano portato ad una situazione in cui le decisioni venivano prese da altri per me. Questi aspetti sono diventati obiettivi di miglioramento personale, per i quali lo *stage* ha rappresentato un’importante opportunità di crescita.

Sebbene ritenessi di aver già fatto progressi nelle mie capacità comunicative, i *feedback* ricevuti dal *tutor* aziendale mi hanno fatto comprendere che ero solo all’inizio di questo percorso di sviluppo personale.

Analogamente, sempre l’esperienza del corso di Ingegneria del *Software* mi ha insegnato l’importanza di accettare l’assenza di soluzioni perfette nello sviluppo *software* e di comprendere la naturalezza della curva di apprendimento. Durante lo *stage*, ho voluto lavorare sulla capacità di non pretendere risultati perfetti fin da subito, accettando che il processo di apprendimento implica necessariamente la possibilità di non conoscere tutto e di commettere errori come parte integrante della crescita professionale.

---

<sup>12</sup>C. Newport, Deep Work: Rules for Focused Success in a Distracted World, Grand Central Publishing, 2016

Obiettivi personali	
<b>P1</b>	Sviluppare maggiore fiducia nelle mie capacità di <i>problem solving</i> e ampliare il repertorio di approcci risolutivi
<b>P2</b>	Approfondire la conoscenza teorica e pratica delle architetture a microservizi, con particolare attenzione ai meccanismi di comunicazione tra servizi
<b>P3</b>	Acquisire la capacità di valutare criticamente le soluzioni tecniche, considerando i compromessi necessari in contesti <b>reali</b> piuttosto che perseguire soluzioni teoricamente perfette
<b>P4</b>	Consolidare la comprensione dei principi di progettazione delle API tra microservizi e delle relative <i>best practices</i> implementative
<b>P5</b>	Migliorare nelle competenze di comunicazione professionale, con particolare riferimento alla presentazione tecnica dei risultati durante gli <i>Sprint review</i>
<b>P6</b>	Migliorare le capacità di comunicazione interpersonale in ambito lavorativo, sia nella richiesta di supporto che nella condivisione dei progressi conseguiti
<b>P7</b>	Sviluppare una maggiore consapevolezza del processo di apprendimento professionale, accettando la naturale curva di apprendimento e la progressiva acquisizione di competenze
<b>P8</b>	Ottimizzare la gestione del tempo lavorativo attraverso un approccio più strutturato e consapevole, privilegiando periodi prolungati di <i>deep work</i> , ossia concentrazione profonda

Tabella 2.3: Tabella degli obiettivi personali nel contesto dello svolgimento dello *stage*

# 3 Svolgimento dello *stage*

## 3.1 Conoscenza del dominio di applicazione

Le attività iniziali del progetto hanno previsto un’immersione approfondita nel dominio applicativo, costituito da un ecosistema manifatturiero dedicato alla produzione di beni per aziende terze. Il sistema in esame si configura come una soluzione trasversale e implementabile in molteplici realtà produttive che adottano metodologie operative simili. L’architettura attuale è basata sulla piattaforma SAI, mentre l’obiettivo dello *stage* è stato quello di condurre un’analisi completa finalizzata alla rimodellazione del dominio.

Le caratteristiche principali del sistema attuale sono:

- **tracciabilità delle risorse umane:** il sistema mantiene un registro completo del personale operativo, con identificazione univoca di ciascun lavoratore all’interno dell’ecosistema informativo;
- **monitoraggio del processo produttivo:** ogni fase della catena di lavorazione è dettagliatamente documentata, con registrazione sistematica dello stato avanzamento lavori per ciascuna operazione in corso;
- **gestione dell’inventario:** implementazione di meccanismi di controllo delle materie prime, con funzionalità di rilevamento delle soglie critiche e conseguente attivazione di processi di approvvigionamento, alcuni dei quali completamente automatizzati;
- **amministrazione integrata:** il sistema incorpora moduli per la gestione contabile, l’elaborazione documentale e la pianificazione logistica del trasporto.

Il progetto di *stage* si è focalizzato principalmente sull’analisi approfondita del dominio applicativo, con l’obiettivo di aggiornare e ridefinire il modello esistente. Questo processo ha previsto l’identificazione e la delimitazione dei *bounded contexts* secondo i principi del *Domain-Driven Design*, cercando di mantenere una rappresentazione chiara e coerente delle diverse aree funzionali del sistema. La fase successiva ha riguardato la formalizzazione dei requisiti funzionali e non funzionali, definendo così i confini tecnici e prestazionali necessari per lo sviluppo del microservizio. Come anticipato nella Sezione 1.6.2, le situazioni descritte possono costituire un modello di dominio completo.

L’obiettivo principale del progetto di *stage* era individuare un modello di dominio specifico e svilupparne un prototipo. Dopo un’attenta valutazione, è stato scelto il **monitoraggio del processo produttivo** come ambito di intervento. Durante il lavoro, ho potuto constatare quanto sia complesso separare completamente i modelli di dominio, poiché alcuni si intersecano inevitabilmente. Questo mi ha portato alla consapevolezza che non esiste una soluzione perfetta in senso assoluto, bensì un compromesso tra le diverse esigenze e realtà operative.

## 3.2 Attività svolte

### 3.2.1 Analisi dei requisiti

Tra i vari modelli di dominio individuati, è stato dunque selezionato quello relativo al **monitoraggio del processo produttivo**.

L'analisi dei requisiti è stata condotta in collaborazione con il *Product Owner*. L'obiettivo primario è stato comprendere approfonditamente le esigenze e le aspettative degli utenti finali, con particolare attenzione al contesto preesistente nel sistema monolitico.

Per pervenire a una definizione precisa dei requisiti, si è adottata la metodologia delle *User Story*, uno strumento concettuale che consente di descrivere sinteticamente e in forma narrativa i bisogni specifici degli utenti. Attraverso la redazione di un documento articolato che raccoglieva tali *User Story*, è stato possibile procedere all'identificazione dei requisiti funzionali iniziali. Queste sono state sviluppate in modo incrementale, parallelamente all'approfondimento della conoscenza del dominio specifico.

Un elemento metodologico di particolare rilevanza è stato l'utilizzo di prototipi dimostrativi. Tali strumenti si sono rivelati estremamente efficaci nel facilitare la visualizzazione della soluzione tanto per il cliente quanto per il *team* di sviluppo, contribuendo a dissipare eventuali ambiguità concettuali.

Nello specifico le attività di analisi hanno previsto un'accurata ispezione da remoto dei sistemi informativi presso l'impianto industriale destinatario del servizio. Tale indagine mi ha consentito di acquisire una comprensione approfondita della giornata lavorativa tipo, fornendo elementi informativi che hanno supportato lo sviluppo nella formulazione di nuove *User Story* e nell'individuazione di questioni critiche da approfondire mediante confronto con i rappresentanti e i consulenti. Io ho interagito solo con il *Product Owner*, infatti è solo quest'ultimo che ha contatto diretto con l'azienda cliente.

Partendo dall'esame approfondito della situazione rappresentata nella Figura 3.1. Tale attività ha permesso di delineare con precisione le necessità operative e gli obiettivi funzionali, costituendo la base per lo sviluppo del prototipo.

L'immagine descrive una situazione reale semplificata con cui l'azienda si interfaccia: la gestione della produzione di un certo **Articolo** finito, la cui produzione prevede un certo **CicloDiLavoro** composto da diverse **Fasi** che sono predefinite. Consideriamo ogni componente corrispondere a un *bounded context*. Nello specifico:

- La produzione di un **Articolo** richiede un **OrdineProduzione**, che rappresenta la quantità da produrre e la quantità prodotta.
- **OrdineProduzioneFasi** tiene traccia dello stato di avanzamento della produzione, monitorando quante unità sono state prodotte in relazione a ciascuna **Fase** e in base alla disponibilità di materiali. Questo strumento è fondamentale per garantire che la produzione si svolga correttamente, considerando anche eventuali imprevisti come la scarsità di risorse. Infatti, se la quantità richiesta non può essere raggiunta a causa di una carenza di materiali, il conteggio si ferma automaticamente quando il numero di unità programmato è stato raggiunto oppure manualmente quando le risorse sono state esaurite. Questo processo tiene conto di diversi fattori, come la quantità da produrre, quella effettivamente prodotta e lo stato del processo (aperto, in corso, chiuso), fornendo un controllo dettagliato del progresso produttivo;
- Il riquadro nell'immagine include i *bounded context* presi in analisi, in particolare il processo di **Rilevamento**, che prevede l'inserimento dei dati da parte dei lavoratori. Ogni

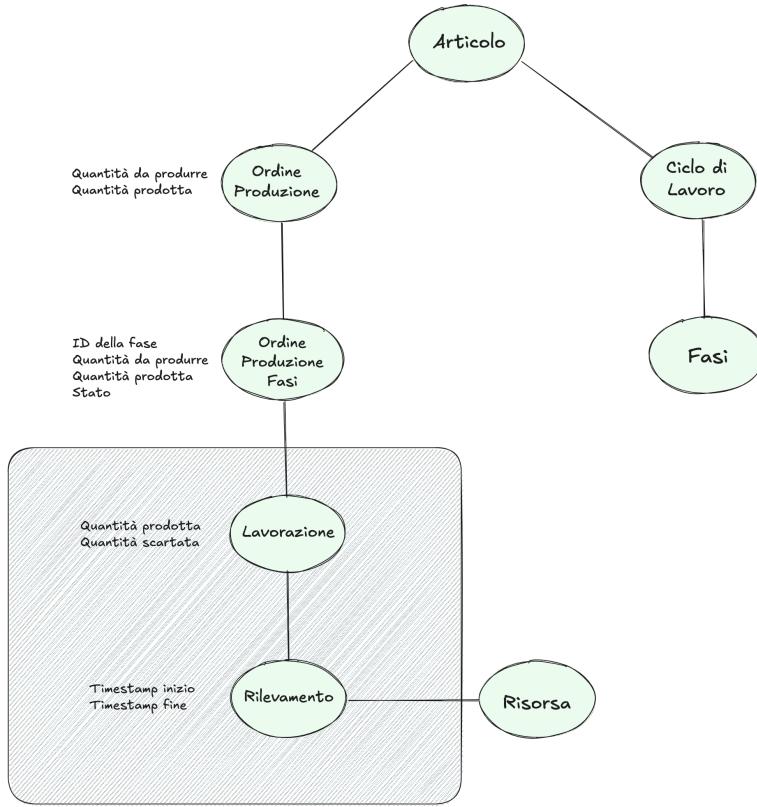


Figura 3.1: Schema visuale del modello di dominio preso in esame durante l’attività di analisi dei requisiti

lavoratore deve segnare, attraverso un *badge*, l’inizio e la fine del proprio turno di lavoro, insieme al numero di pezzi prodotti in base alla fase di **Lavorazione** in corso. Una volta completata una fase di **Rilevamento**, questa azione porta all’avanzamento della **Lavorazione**, la quale tiene traccia sia della quantità prodotta che della quantità scartata, monitorando quindi la qualità del processo;

- Una **Risorsa** è rappresentata da tutti gli elementi necessari per il completamento di una **Fase** produttiva: include l’operaio che esegue il lavoro, la macchina utilizzata per ciascuna fase di produzione, la fase stessa, il numero di unità prodotte, nonché l’ora di inizio e di fine di ciascun intervento.

Il problema principale è far comunicare in tempo reale **Rilevamento** e **OrdineProduzioneFasi**, senza che quest’ultimo debba essere portato all’interno del microservizio che individuiamo come **MS\_Rilevamento**.

La soluzione ideale infatti sarebbe quella di includere **OrdineProduzione** e **OrdineProduzioneFasi** nell’unico microservizio **MS\_Rilevamento**. Questo perché secondo i principi del *Domain-Driven Design* la divisione in *bounded contexts* dovrebbe attuare una scomposizione in microservizi che siano poco accoppiati tra di loro. In questo caso invece c’è un alto grado di accoppiamento tra i *bounded context* appena citati e **Lavorazione**, quindi scomporli in servizi differenti non è in linea con i principi DDD<sup>1</sup>.

Questa soluzione non è applicabile al momento perché il monolite è strettamente collegato in vari punti con **OrdineProduzione** e con **OrdineProduzioneFasi**. La priorità è dunque estrarre **Rilevamento** e **Lavorazione** in un unico servizio, e in un secondo momento completare la divisione ideale.

<sup>1</sup>E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003

Un altro problema è rappresentato dalla necessità di `MS_Rilevamento` di leggere dei dati da due fonti. Per monitorare lo stato di una `Lavorazione` in relazione a `OrdineProduzione`, è necessario confrontare i dati provenienti sia dal microservizio, sia dal monolite. Questo perché altrimenti non è possibile accedere ai dati essenziali dell'ordine della fase da lavorare, senza portare `OrdineProduzione` all'interno di `MS_Rilevamento`. Come farlo sarà oggetto della sezione successiva.

Nell'ambito del mio *stage* didattico, la complessità del dominio applicativo e i vincoli temporali intrinseci al progetto hanno determinato la necessità di selezionare i requisiti individuati in base alle priorità.

Sebbene l'implementazione si sia concentrata su un sottoinsieme specifico delle entità individuate, ho compreso comunque tutti i requisiti che ho individuato nelle attività di analisi. Questa scelta metodologica ha consentito di mantenere una visione olistica del progetto, garantendo al contempo una realizzazione maggiormente sostenibile e allineata con le risorse disponibili.

I requisiti individuati li ho definiti seguendo le linee guida aziendali, assicurando così una perfetta coerenza con gli standard e le aspettative di SogeaSoft S.r.l. La Tabella 3.1 rappresenta una sintesi concisa di tali requisiti, offrendo una panoramica strutturata degli obiettivi progettuali complessivi.

<b>Tipo</b>	<b>Obbligatori</b>	<b>Desiderabili</b>	<b>Opzionali</b>
Dominio	23	0	0
Funzioni	89	12	8
Prestazioni	0	0	7
Sincronizzazione	5	2	1
Progettazione	12	0	0
159 requisiti totali			

Tabella 3.1: Tabella dei requisiti individuati durante l'attività di analisi dei requisiti

### 3.2.2 Progettazione

Durante il mio *stage*, l'analisi dei requisiti e le attività di progettazione si sono sviluppate in modo parallelo. Con l'aumentare della mia conoscenza del dominio applicativo e l'individuazione dei requisiti, la progettazione, guidata dal *Product Owner*, è avvenuta in modo naturale, senza una netta separazione tra le due attività.

Poiché la progettazione consiste nel definire come realizzare concretamente quanto emerso dall'analisi dei requisiti, questo capitolo si concentrerà sulla descrizione dei *pattern* identificati e delle soluzioni architetturali adottate per soddisfare i requisiti individuati.

La soluzione individuata è rappresentata nella Figura 3.2 e rappresenta il modo in cui `MS_Rilevamento` potrà comunicare con SAI attraverso un `MS_Middleware`. Questo rappresenta molto bene il piano di migrazione visibile nella Figura 2.5.

Date le dimensioni molto grandi del monolite in esame, l'azienda ha scelto di optare per una migrazione graduale. La soluzione più ragionevole è risultata dunque quella di applicare il *pattern Anti-Corruption Layer* (ACL) prevedendo la lunga convivenza del monolite e del nuovo sistema a microservizi. In questo caso questo *pattern* trova rappresentazione in `MS_Middleware`.

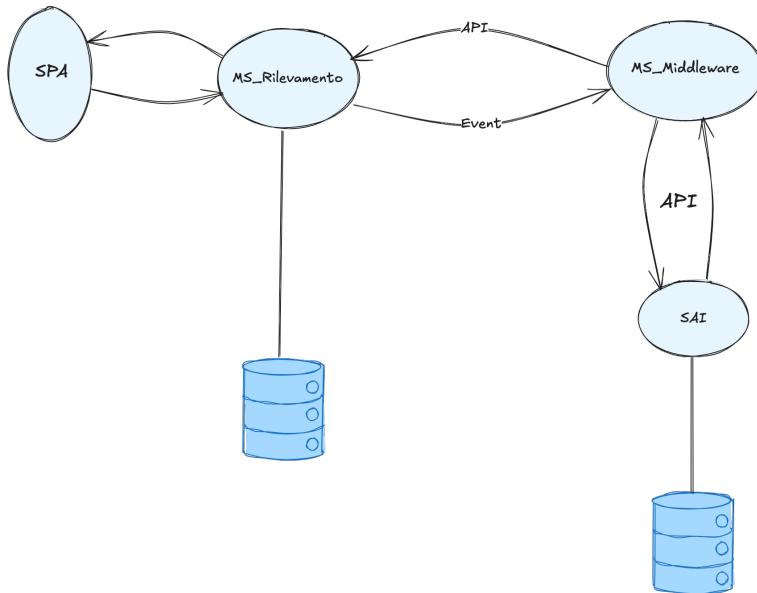


Figura 3.2: Attività di progettazione per quanto concerne l'estrazione di `MS_Rilevamento`

Un altro *pattern* che ho proposto è il *Change Data Capture*, una tecnica utilizzata per identificare e tracciare le modifiche apportate ai dati in un *database*. Nel nostro caso è utile nei contesti di sincronizzazione tra sistemi eterogenei, ossia di `MS_Rilevamento` e di `SAI`, il monolite.

Per l'implementazione di questo microservizio, insieme al *Product Owner* siamo giunti alla scelta del modello di **architettura esagonale**, noto anche come *pattern* di porte e adattatori che mira a creare architetture liberamente accoppiate in cui i componenti delle applicazioni possano essere testati in modo indipendente, senza dipendenze da archivi di dati o interfacce utente. Viene utilizzato per isolare la logica aziendale (logica di dominio) dal codice dell'infrastruttura correlato. Le **porte** sono punti di ingresso indipendenti dalla tecnologia in un componente dell'applicazione. Gli **adattatori** invece, implementano le interfacce definite dalle porte, consentono una comunicazione fluida tra il nucleo applicativo e i sistemi esterni, facilitando future estensioni o sostituzioni di componenti senza impattare sulla logica di dominio centrale.

Nello specifico abbiamo implementato tutti gli elementi legati al dominio, incluse la dichiarazione degli oggetti e le configurazioni fondamentali in una parte chiamata **Core**. In questo modello, il *Core* include:

- **modelli di dominio**, che descrivono gli oggetti fondamentali e il loro comportamento, definendo le entità, i *value objects* e gli aggregati;
- **servizi di dominio**, che implementano logiche più complesse, mantenendo il dominio autonomo rispetto agli adattatori esterni;
- **eventi di dominio**, che permettono di notificare cambiamenti interni senza creare un forte accoppiamento tra componenti;
- **application services**, che rappresentano le operazioni principali eseguibili dall'applicazione, orchestrando l'interazione tra il dominio e le interfacce esterne.

L'altra componente è la **Infrastructure**, che contiene tutti gli elementi necessari per connettere il dominio con il mondo esterno. Questa parte dell'architettura comprende le porte e gli adattatori, nonché i protocolli di comunicazione che permettono l'integrazione tra le varie componenti del sistema, consentendo al *Core* di rimanere isolato e indipendente. Nello specifico, la componente *Infrastructure* comprende:

- **adattatori di persistenza**: gestiscono l’interazione con il *database*, fornendo meccanismi per il salvataggio e il recupero dei dati senza esporre dettagli implementativi al dominio. Questo può includere *repository* che implementano interfacce definite nel *Core*, consentendo di cambiare la tecnologia di persistenza senza modificare la logica applicativa;
- **adattatori di comunicazione** (API REST, messaggistica): consentono al microservizio di esporre e ricevere dati attraverso protocolli standard come REST (HTTP) o eventi asincroni (message brokers o RabbitMQ). In particolare, le operazioni PUT, POST, GET e DELETE implementano le azioni previste dai casi d’uso del dominio;
- **Web API**: rappresenta il punto di accesso del microservizio, consentendo la comunicazione con altri servizi, come il *Middleware* o il monolite esistente. Questa API traduce le richieste esterne in comandi eseguibili dal *Core* e restituisce le risposte formattate in un protocollo standard ( $JSON_G$ , approfondito nella sezione successiva);

Questa suddivisione consente di ottenere un sistema modulare, scalabile e manutenibile, in cui il dominio rimane indipendente dai dettagli tecnologici. Inoltre, la *Infrastructure* può essere sostituita o aggiornata senza impattare direttamente il *Core*, rendendo il sistema più flessibile nel tempo. Una buona rappresentazione dei risultati della progettazione è osservabile nella Figura 3.3.

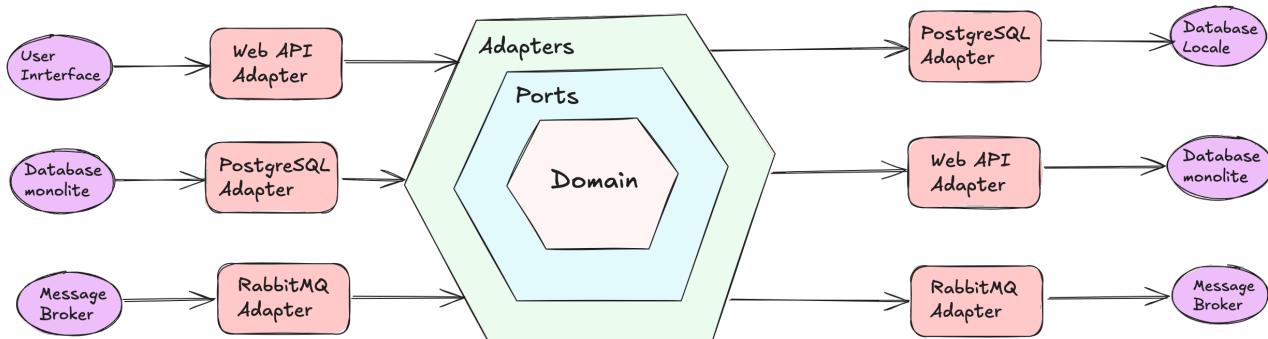


Figura 3.3: Rappresentazione grafica dell’architettura esagonale e i rispettivi adattatori e porte.

Come anticipato nella precedente sezione, un problema da risolvere è quello di effettuare due letture sul *bounded context Lavorazione*. La soluzione trovata per fare ciò è effettuare una lettura su *MS\_Rilevamento* tramite un *adapter* PostgreSQL sul *database* locale, e una su SAI tramite una connessione PostgreSQL al *database* del monolite stesso, strategia sempre visibile nella Figura 3.3.

Per la scrittura sui *database* invece, studiando i *pattern* di scomposizione per questa specifica situazione ho individuato il *pattern Database Wrapping Service*<sup>2</sup>. Nello specifico si implementa un servizio intermedio (*wrapping service*) che si occupa di esporre le informazioni di SAI tramite delle *Web API* senza esporre direttamente il *database* sottostante, il tutto sempre riconducibile alla Figura 3.3.

In questo modo, il *team* che gestisce il monolite è responsabile dell’aggiornamento delle API in caso di modifiche alla struttura del *database*, e il *team* di *MS\_Rilevamento* non deve preoccuparsi dei cambiamenti interni a SAI. Questo approccio riduce l’accoppiamento tra i sistemi, rendendo più facile l’evoluzione della struttura dei dati.

<sup>2</sup>S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, O'Reilly Media, 2019

Infine, insieme al *Product Owner*, abbiamo pensato a come sincronizzare questi dati. Dallo studio della letteratura sui *pattern* è emerso come migliore per il nostro problema il ***Change Data Capture (CDC)*** implementato con un ***Batch Delta Copier***<sup>2</sup>.

Il *Change Data Capture* è un *pattern* che monitora e cattura le modifiche ai dati in un sistema di origine (nel nostro caso SAI) per sincronizzarle con sistemi di destinazione, garantendo così la coerenza delle informazioni attraverso piattaforme diverse. Il *Batch Delta Copier* è un'implementazione specifica di CDC che funziona identificando periodicamente i dati nuovi o modificati dalla sorgente (il *delta*) e copiandoli in blocco verso la destinazione durante finestre temporali programmate. Nel nostro caso è un sistema basato sui *transaction log*: quando una nuova transazione arriva nel *database* del microservizio, viene registrata in un file di *log* senza impattare il monolite. È quindi possibile rilevare tali modifiche e trasferirle dal *log* a SAI. Una rappresentazione grafica di questo processo si può osservare nella Figura.

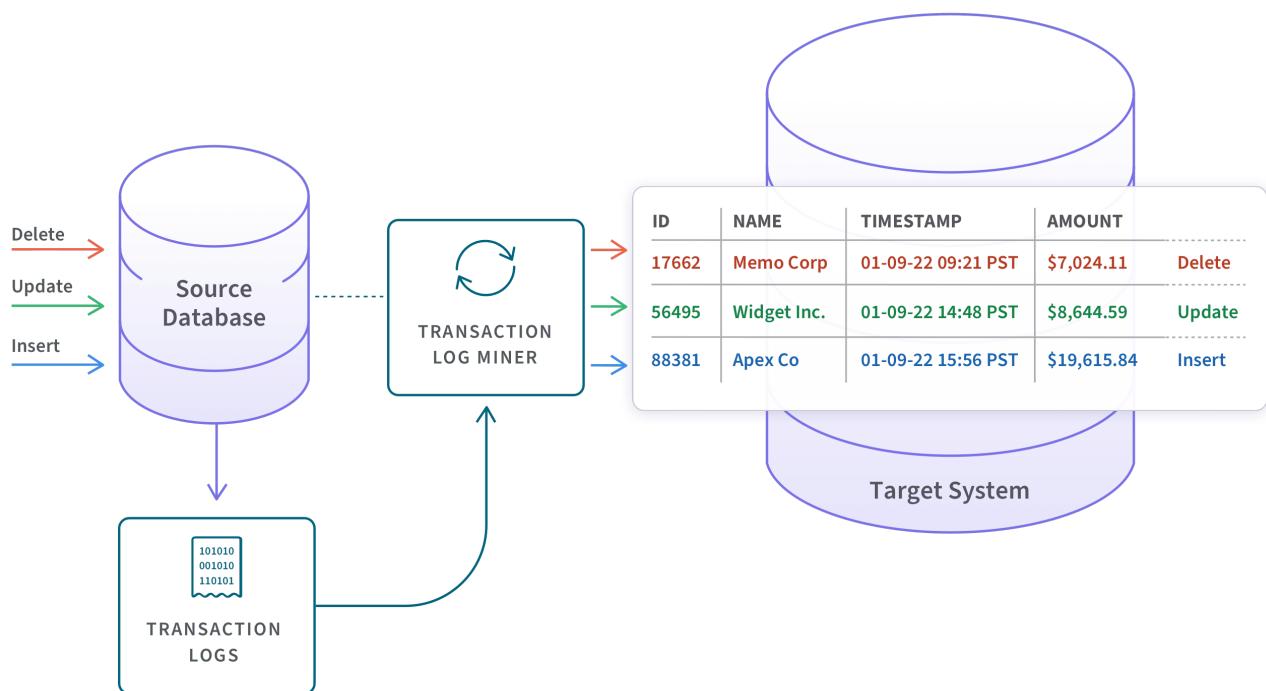


Figura 3.4: Rappresentazione del *pattern Change Data Capture (CDC)* nel contesto dell'attività di progettazione. Fonte: <https://wwwqlik.com/us/change-data-capture/cdc-change-data-capture> (*ultima visita: 29/03/2025*)

Le attività di progettazione, oltre ad avermi consentito di individuare i *pattern* da utilizzare, hanno permesso a me e al *Product Owner* di definire una serie di metriche interessanti per questo progetto. L'obiettivo primario è stato valutare e monitorare delle prestazioni dell'applicazione prima e dopo lo sviluppo del microservizio.

Inizialmente, abbiamo proceduto alla costruzione di un *Product Backlog* dettagliato, approfondito nella Sezione 1.4. Non mi sono limitata a catalogare i requisiti, ma li ho anche associati a stime temporali indicative, consentendomi una pianificazione più accurata, tutto sempre con la supervisione e consiglio del *Product Owner*. Tale strumento mi ha fornito una proiezione affidabile dei tempi necessari per l'implementazione di ciascun componente del progetto. In particolare, la previsione temporale che ho svolto rispetto ai requisiti individuati mi ha permesso di stimare un completamento del 73% dei requisiti totali contenuti nel *Product Backlog*. La Figura 3.5 presenta una rappresentazione visiva del grado di copertura previsto.

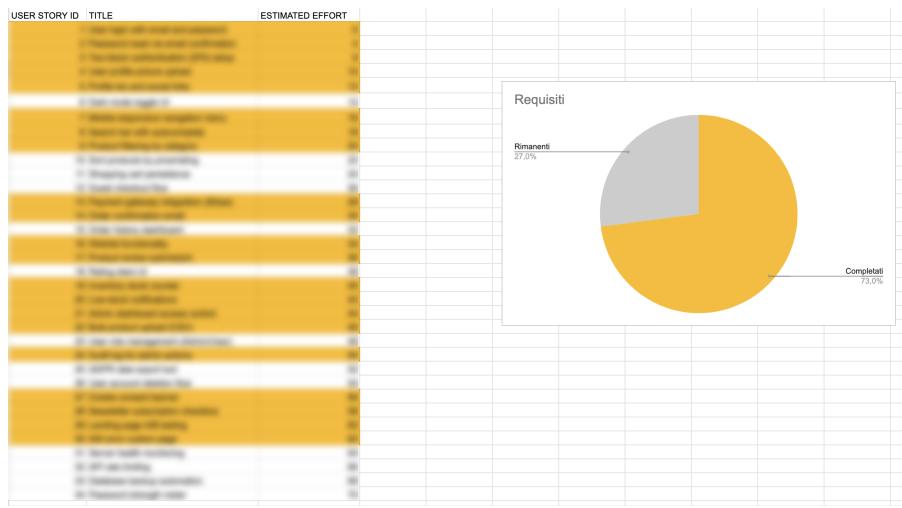


Figura 3.5: *Product Backlog* utilizzato durante il processo di progettazione, accompagnato da una misura della quantità di requisiti che le soluzioni trovate soddisfare.

Un aspetto particolarmente significativo dell’analisi è stata la misurazione delle prestazioni del sistema esistente. Abbiamo effettuato una valutazione delle velocità di estrazione dati nel sistema monolitico preesistente, predisponendo un *benchmark* essenziale per le successive comparazioni. Questo approccio ha gettato le basi per una valutazione oggettiva dei miglioramenti prestazionali introdotti dal nuovo servizio.

L’analisi si è poi estesa a considerazioni economiche più ampie, introducendo il concetto di *Return On Investment<sub>G</sub>* (*ROI<sub>G</sub>*). Tale prospettiva ha rivelato la complessità intrinseca della trasformazione architettonale: l’estrazione di un microservizio comporta investimenti iniziali significativi in termini di risorse, formazione e sviluppo. La validità dell’intervento dipende quindi dalla capacità di generare un vantaggio tangibile per l’organizzazione.

Abbiamo quindi proceduto a una misurazione comprensiva delle metriche di *performance*, analizzando la velocità dei processi interni nel sistema corrente. Questo approccio metodologico ha predisposto un solido *framework* per le successive attività di validazione, permettendo un confronto tra le prestazioni del sistema monolitico e la nuova soluzione implementata.

### 3.2.3 Implementazione

La fase di progettazione è stata senza dubbio la più complessa, soprattutto a causa della mancanza di soluzioni definitive e della relativa novità del concetto di migrazione verso i microservizi<sup>3</sup>. Nonostante le difficoltà iniziali, una volta definita la strategia da adottare, l’implementazione è proseguita in modo naturale, seguendo il percorso tracciato.

La fase di codifica è stata improntata su un approccio *learn by doing*: pur avendo una base di riferimento, ho dovuto affinare le mie competenze sul campo, confrontandomi regolarmente con il *tutor* aziendale e il *Product Owner* per ricevere *feedback* e indicazioni utili.

Nello specifico, ho potuto implementare una struttura di progetto molto precisa, già anticipata nella Sezione 3.2.2 e rappresentata nella Figura 3.6.

<sup>3</sup>D. Shadija, M. Rezai, and R. Hill, "Towards an Understanding of Microservices," in Proceedings of the 2017 IEEE International Conference on Computer and Information Technology (CIT), Helsinki, Finland, 2017, pp. 1072-1078



Figura 3.6: Struttura del codice utilizzata per l'estrazione del microservizio.

Analizzando più approfonditamente la struttura del microservizio `Sogea.RilevamentoProduzione`, posso espandere diversi concetti architettonici, descritti come segue.

### Domain-Driven Design (DDD)

Tutto ruota attorno al progetto `Sogea.Rilevamento.Produzione.Domain`, che è il cuore dell'applicazione. Qui ci sono entità, aggregati e *value objects*, regole di *business* e tutta la logica che descrive cosa significa *Rilevamento Produzione* per l'azienda. È isolato dal *database*, dalle API e da dettagli tecnici, proprio come DDD suggerisce. L'immagine 3.7 racconta un esempio di entità (`Rilevamento`) all'interno di un aggregato (`LavorazioneAggregate`). Sono implementati nello specifico i metodi `IniziaRilevamento` e `RegistraRilevamento`.

> Sogea.RilevamentoProduzione.ApplicationService	●	25	
> Sogea.RilevamentoProduzione.Command	●	26	
> Sogea.RilevamentoProduzione.Configuration	●	27	
✓ Sogea.RilevamentoProduzione.Domain	●	28	
> bin	●	29	
> CausaleAggregate	●	30	
> FaseTipo	●	31	
✓ LavorazioneAggregate	●	32	
> Events	●	33	
> Exceptions	●	34	
> Services	●	35	
☛ Durata.cs	M	36	
☛ Enums.cs	M	37	
☛ ILavorazioneRepository.cs	M	38	
☛ Lavorazione.cs	M	39	
☛ Quantita.cs	M	40	
☛ Rilevamento.cs	M	41	
< obj		42	
		43	
		44	
		45	

```

public static Rilevamento IniziaRilevamento(Risorsa risorsa, DateTime inizio)
=> new Rilevamento
{
  Stato = RilevamentoStato.Aperto,
  Durata = Durata.DurataIniziale(inizio),
  CausaleSospensioneId = null,
  RisorsaId = risorsa.Id,
  Macchina = risorsa.Tipo == RisorsaAggregate.Enums.RisorsaTipo.Macchina
};

0 references | Qodo Gen: Options | Test this method
public static Rilevamento RegistraRilevamento(Risorsa risorsa, DateTime fine, short minuti)
=> new Rilevamento
{
  Stato = RilevamentoStato.Chiuso,
  Durata = Durata.DurataCompletaFromMinuti(fine, minuti),
  CausaleSospensioneId = null,
  RisorsaId = risorsa.Id,
  Macchina = risorsa.Tipo == RisorsaAggregate.Enums.RisorsaTipo.Macchina
};
  
```

Figura 3.7: Definizione dei metodi dell'entità `Rilevamento` all'interno dell'aggregato `LavorazioneAggregate`.

### Command Query Responsibility Segregation (CQRS)

Il *Command Query Responsibility Segregation* è un *pattern* che divide il sistema in due parti: i *Command*, che gestiscono le modifiche ai dati, e le *Query*, che si occupano delle letture. Questa separazione è un'applicazione pratica caratteristica del CQRS, approccio particolarmente utile nel *Domain-Driven Design* per affrontare scenari complessi garantendo maggiore scalabilità. In questo contesto, gli *ApplicationService* svolgono un ruolo cruciale di coordinamento, poiché

racchiudono la logica delle *user story* mantenendo il dominio pulito da contaminazioni esterne. Questa struttura permette di gestire in modo più efficace e organizzato le diverse responsabilità all'interno del sistema.

Gli *handler*<sub>G</sub> sono il cuore del CQRS in questo sistema. Nello specifico gli *handler* dei comandi ricevono richieste di modifica, le validano, applicano la logica di dominio e ne aggiornano lo stato. Gli *handler* delle *query*<sub>G</sub> invece, ricevono richieste di informazioni e restituiscono i dati necessari senza modificare lo stato. Un esempio rappresentativo di un hanler è raccontato dalla Figura 3.8, in cui si può osservare un esempio dell'handler **ResourceHandler**, il quale implementa i comandi **Get**, **Create**, **Delete**, secondo il protocollo REST basato su HTTP. Questo definisce un insieme di vincoli e proprietà per la creazione di servizi *web*, già citato nella Sezione 1.7. Nell'implementazione mostrata, il **ResourceHandler** fornisce un'interfaccia conforme ai principi REST, esponendo le operazioni fondamentali per la manipolazione delle risorse attraverso i metodi standard HTTP. In questo caso, rispettivamente all'ordine di apparizione nell'immagine, *read*, *create*, *delete*.

```

public class ResourceHandler : IHandler
{
    [ActionHandler(ActionHandlerMeanings.Read, ActionHandlerNaming.DerivedFromMeaning)]
    public async Task<Resource> GetResourceAsync(Guid id)
    {
        return await this.QueryProcessor.Process(new ResourceQuery(id));
    }

    [ActionHandler(ActionHandlerMeanings.Create, "create-person")]
    public async Task<Resource> CreateResource([Payload] CreateResourceCommand payload)
    {
        Guid id = await this.DomainBus.Send(payload);

        return await this.QueryProcessor.Process(new ResourceQuery(id));
    }

    [ActionHandler(ActionHandlerMeanings.Create, "create-machine")]
    public async Task<Resource> CreateResourceMachine([Payload] CreateResourceMachineCommand payload)
    {
        Guid id = await this.DomainBus.Send(payload);

        return await this.QueryProcessor.Process(new ResourceQuery(id));
    }

    [ActionHandler(ActionHandlerMeanings.Delete, ActionHandlerNaming.DerivedFromMeaning)]
    public async Task DeleteResource(Guid id, [Payload] byte[] timestamp)
    {
        await this.DomainBus.Send(new DeleteResourceCommand(id, timestamp));
    }
}

```

Figura 3.8: Il **ResourceHandler** implementa le operazioni per la creazione, lettura e cancellazione dei dati seguendo i principi REST, fungendo da interfaccia tra le richieste HTTP e la logica applicativa del sistema di **RilevamentoProduzione**.

Sempre nella Figura 3.8, si osserva la parola chiave **payload**. In questo caso **payload** svolge un ruolo fondamentale come contenitore di dati per le operazioni di creazione delle risorse. Questo è un attributo che indica che il parametro contiene i dati necessari per l'operazione di creazione. Quando una richiesta di creazione arriva al *handler*, il **payload** contiene tutte le informazioni necessarie fornite dal *client* (come proprietà, valori e configurazioni della risorsa da creare). Questo viene poi inoltrato al **DomainBus** attraverso il metodo **Send()**, che lo trasmette al livello di dominio appropriato per l'elaborazione.

Questo **payload** corrisponde a un *Data Transfer Object*, ossia un oggetto usato per trasferire dati tra sistemi o livelli di un'applicazione, senza contenere logica di *business*. Serve a ottimizzare la comunicazione e ridurre il numero di chiamate tra componenti. È un **payload** perché rappresenta il contenuto effettivo trasmesso in una richiesta o risposta.

## Data Transfer Object (DTO)

I *Data Transfer Objects*<sub>G</sub> (DTOs<sub>G</sub>) sono rilevanti in questo contesto perché permette di disaccoppiare il modello di dominio delle interfacce esterne e fornisce anche una struttura chiara per i dati in ingresso e in uscita, senza esporre dettagli implementativi interni.

Come anticipato nel paragrafo precedente, quando un *client* invia una richiesta, utilizza un DTO come *payload*<sub>G</sub>, ossia la parte di un messaggio o di una richiesta che contiene i dati effettivi da elaborare. Questo DTO viene poi elaborato dall'*handler* appropriato che esegue la logica necessaria. I DTO permettono di controllare esattamente quali dati vengono trasferiti, facilitano la validazione e migliorano la sicurezza nascondendo i dettagli implementativi del dominio. Nella Figura 3.9 si può osservare un esempio di *payload*, un DTO che contiene il messaggio con le informazioni che verranno effettivamente mandate al *client* che fa la richiesta. Nel caso specifico, si tratta di un **rilevamento** rispetto a una **lavorazione**.

```
> Sogea.RilevamentoProduzione.ApplicationService      5  namespace Sogea.RilevamentoProduzione.DTOs.Jobs
> Sogea.RilevamentoProduzione.Command                 6  {
> Sogea.RilevamentoProduzione.Configuration          7  0 references | Qodo Gen: Options | Test this class
> Sogea.RilevamentoProduzione.Domain                8  public class JobDetection : BaseDto
> Sogea.RilevamentoProduzione.DTOs                  9  {
> bin                                               10  0 references
> < Jobs                                            11  public Guid Id { get; set; }
> Duration.cs                                       12  0 references
> Enums.cs                                         13  public Guid ResourceId { get; private set; }
> Job.cs                                            14  0 references
> JobDetection.cs                                  15  public bool Machine { get; private set; }
> Quantity.cs                                      16  0 references
> TimesheetListItem.cs                           17  public Duration Duration { get; private set; }
> obj                                               18  0 references
> ProductionOrderPhase                         19  public JobDetectionStatus Status { get; private set; }
> Reasons                                           20  0 references
> Resources                                         21  public Guid? CausaleSospensioneId { get; private set; }
> Sogea.RilevamentoProduzione.DTOs.csproj
> Sogea.RilevamentoProduzione.Events
```

Figura 3.9: Rappresentazione di un *Data Transfer Object*

In sintesi, in questo microservizio, CQRS con *handler* e DTO offre una separazione chiara delle responsabilità, migliora la manutenibilità del codice e permette di ottimizzare separatamente le operazioni di lettura e scrittura.

## Web API

I microservizi, per loro natura intrinseca, comportano significative attività di integrazione per garantire una comunicazione efficace tra i diversi componenti. Un esempio concreto di questa necessità è rappresentato dalla soluzione implementata per il problema, presentato nelle sezioni immediatamente precedenti, della duplice lettura, che si verificava sia sul sistema monolitico sia sul microservizio MS\_Rilevamento da parte di OrdineProduzioneFase. Per risolvere questa criticità, è stata sviluppata un'**API** dedicata che funge da intermediario nella comunicazione, ottimizzando così il flusso di dati tra le diverse parti dell'architettura.

Il componente fondamentale che implementa l'operazione di lettura all'interno dell'API è il metodo **FindByFaseAsync** della classe **OrdineProduzioneRepository**. Questo metodo esemplifica l'implementazione di un'operazione di recupero dati secondo il paradigma REST, approfondito nella Sezione 1.7.

L'operazione è raccontata nella Figura 3.10, e si articola attraverso i seguenti passaggi.

```

public async Task<OrdineProduzione> FindByFaseAsync(int ordineProduzioneFaseId)
{
    string url = this.Configuration.Server.Replace("{productionOrderId}", ordineProduzioneFaseId.ToString());
    RestClient client = new RestClient(url);
    RestRequest request = new RestRequest(Method.GET);
    request.AddHeader("DITTA", this.Configuration.Ditta);
    IRestResponse response = await client.ExecuteAsync(request);

    if (response.IsSuccessStatusCode && response.Content != null)
    {
        RootDto rootDto = JsonConvert.DeserializeObject<RootDto>(response.Content);
        string id = rootDto.ProductionOrder.Id;
        string codice = rootDto.ProductionOrder.Codice;

        OrdineProduzioneStato stato = (OrdineProduzioneStato)rootDto.ProductionOrder.Stato;
        List<OrdineProduzioneFase> fasi = new List<OrdineProduzioneFase>();

        foreach (var phaseDto in rootDto.ProductionOrder.ProductionOrderPhase)
        {
            Fase fase = new Fase(phaseDto.Codice, phaseDto.FaseTipoId, phaseDto.AccoppiataConMacchina);
            OrdineProduzioneFase ordineProduzioneFase = new OrdineProduzioneFase(
                int.Parse(phaseDto.Codice),
                fase,
                (OrdineProduzioneFaseStato)phaseDto.OrdineProduzioneFaseStato,
                (OrdineProduzioneFaseTipo)phaseDto.OrdineProduzioneFaseTipo,
                phaseDto.QuantitaDaProdurre,
                phaseDto.QuantitaProdotta,
                await this.ConvertToInt(phaseDto.MacchinaId),
                phaseDto.RepartoId
            );

            fasi.Add(ordineProduzioneFase);
        }

        OrdineProduzione ordineProduzione = new OrdineProduzione(id, codice, stato, fasi);
        return ordineProduzione;
    }
    return null;
}

```

Figura 3.10: Implementazione dell’API per la comunicazione tra `OrdineProduzioneFase` e il database monolitico

Inizialmente, viene costruito dinamicamente l’ $endpoint_G$  della richiesta. L’URL<sub>G</sub> di destinazione viene generato sostituendo il segnaposto `productionOrderId` con l’identificativo effettivo della fase di produzione (`ordineProduzioneFaseId.ToString()`). Questa sostituzione avviene tramite il metodo `Replace` dell’oggetto `Configuration.Server`.

Viene dunque istanziato un *client* REST (`new RestClient(url)`) e configurata una richiesta HTTP di tipo **GET** (`new RestRequest(Method.GET)`). La scelta del metodo GET è conforme alle convenzioni REST, dove GET rappresenta un’operazione di lettura che non modifica lo stato delle risorse sul *server*.

Per contestualizzare la richiesta all’ambiente aziendale specifico, viene aggiunto un *header* personalizzato denominato "DITTA" tramite il metodo `AddHeader`, utilizzando il valore memorizzato nella configurazione dell’applicazione.

L’esecuzione della richiesta avviene in modalità asincrona (`await client.ExecuteAsync(request)`), una scelta che ottimizza l’utilizzo delle risorse di sistema evitando blocchi durante l’attesa della risposta dalla rete.

Nella fase di elaborazione, il codice verifica il successo della risposta e la presenza di contenuti. In caso positivo, i dati  $JSON_G$  ricevuti vengono deserializzati in un oggetto `RootDto` tramite `JsonConvert.DeserializeObject<RootDto>`. Da questo DTO radice, vengono estratte le informazioni necessarie come identificativi, codici e stati dell’ordine di produzione. Questo è un perfetto esempio di come vengono strutturati gli aggregati nel contesto DDD.

Nel DDD, gli aggregati rappresentano raggruppamenti di entità e oggetti di valore che vengono trattati come un’unità coesa per garantire l’integrità dei dati. Qui vediamo come i dati provenienti da un sistema esterno (in formato JSON) vengono prima trasformati in un DTO neutro (`RootDto`), che agisce come struttura intermedia, e poi progressivamente ricostruiti negli oggetti di dominio veri e propri (`OrdineProduzione`, `OrdineProduzioneFase`, `Fase`). Questa operazione è visibile nella Figura 3.11.

```

namespace Sogea.RilevamentoProduzione.WebAPI.DTOs
{
    public class ProductionOrderDto
    {
        public string ArticoloId { get; set; }
        public string CommessaId { get; set; }
        public string Id { get; set; }
        public int Stato { get; set; }
        public List<ProductionOrderPhaseDto> ProductionOrderPhase { get; set; }
    }
}

```

Figura 3.11: Definizione della classe `ProductionOrderDto` contenente le proprietà principali e le relazioni con le fasi degli ordini di produzione nel sistema, nonché la relazione in quanto aggregato.

Tornando alla Figura 3.10, particolarmente interessante è la sezione di codice che itera attraverso le fasi dell'ordine di produzione (`foreach (var phaseDto in rootDto.ProductionOrder.ProductionOrderPhase)`). Per ciascuna fase, viene creato un nuovo oggetto `Fase` con i dettagli estratti dal DTO, seguito dalla creazione di un oggetto `OrdineProduzioneFase` completo di tutti i dati necessari.

Infine, tutti gli oggetti `OrdineProduzioneFase` vengono aggregati in una lista e utilizzati per costruire l'oggetto `OrdineProduzione` completo, che viene restituito come risultato dell'operazione. In caso di insuccesso della richiesta o in assenza di dati validi, il metodo restituisce `null`.

Questa implementazione dimostra un'efficace integrazione tra sistemi distribuiti, ottimizzando le comunicazioni e riducendo la duplicazione di richieste di dati attraverso un'interfaccia REST ben strutturata.

## Change Data Capture

Per affrontare invece la problematica della sincronizzazione dei dati, ho potuto esplorare e implementare un'architettura specializzata basata sulla tecnologia **Debezium**, una piattaforma di **Change Data Capture (CDC)**, approfondita nella Sezione 1.7.

L'architettura di sincronizzazione adottata ha integrato Debezium con **RabbitMQ**, quest'ultimo utilizzato come *message broker* per la gestione efficiente del flusso di informazioni. Questa combinazione tecnologica ha consentito la realizzazione di un sistema di code (inteso come liste) di messaggi altamente reattivo, in grado di catturare e propagare le modifiche ai dati nel momento stesso in cui queste venivano apportate nel sistema di origine.

Il principio operativo di questa soluzione si basa sul meccanismo di CDC, attraverso il quale Debezium monitora continuamente il *log* delle transazioni del *database*, identifica le operazioni di inserimento, aggiornamento o eliminazione dei dati e trasforma queste informazioni in eventi

che vengono successivamente instradati attraverso RabbitMQ verso i sistemi destinatari, come si può osservare in Figura 3.12.

Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/	product-updates	classic	D Args	running	16	0	16	0.00/s	0.00/s	0.00/s

Figura 3.12: Coda implementata con Debezium e gestita da RabbitMQ.

L’implementazione dunque si è svolta senza particolari difficoltà, seguendo le attività di progettazione precedentemente definite. Rispetto alla previsione iniziale illustrata nella Figura 3.5, avevo stimato di implementare circa il 73% dei requisiti individuati. Tuttavia, al termine dello sviluppo, ho potuto constatare di aver raggiunto un 76%, superando leggermente le aspettative iniziali.

In particolare, l’esplorazione di **Debezium** era inizialmente prevista solo a livello teorico, ma sono riuscita anche a soddisfare i primi requisiti per la sua implementazione, ponendo le basi per l’adozione completa di questo *framework* nell’applicazione del *pattern Change Data Capture* (CDC).

### 3.2.4 Verifica e validazione

Il processo di verifica del *software* è stato articolato in più fasi, iniziando dal controllo delle singole unità di codice per poi estendersi all’intero sistema, come illustrato nella Sezione 1.6.4. In SogeaSoft S.r.l., è prassi consolidata che qualsiasi modifica al codice venga sottoposta a revisione prima di essere integrata nel *branch* stabile di sviluppo. Nello specifico, le modifiche devono essere pubblicate tramite una *Pull Request* su Microsoft Azure e sottoposte alla verifica di un altro sviluppatore.

Quando viene aperta una *Pull Request*, viene automaticamente avviata una *pipeline* di *test* sulla piattaforma Microsoft Azure. In questo ambiente, il codice viene compilato e sottoposto a *test* automatici per verificarne la correttezza.

Per tutti i *task* rilevanti, ho provveduto a scrivere una serie di ***test* unitari** volti a garantire il corretto funzionamento delle logiche definite durante le attività di progettazione. In SogeaSoft S.r.l. non sono previsti requisiti minimi di copertura per i *test* unitari; tuttavia, ho scelto autonomamente di raggiungere il 100% di copertura per tutte le funzionalità da me sviluppate. Questo approccio mi ha permesso di scrivere codice più solido e aderente alle specifiche progettuali.

Oltre ai *test* unitari, ho eseguito anche ***test* di integrazione**, concentrandomi in particolare sulla verifica del corretto funzionamento delle chiamate REST, su cui si basa gran parte del

sistema. Questi *test* avevano l'obiettivo di garantire che le funzionalità sviluppate fossero correttamente esposte nell'ambiente *server* e rispondessero in modo adeguato alle richieste.

Per la verifica e la documentazione delle API, ho utilizzato **Swagger**, descritto più nel dettaglio nella Sezione 1.7. Questo *framework* mi ha permesso di monitorare in tempo reale il comportamento delle nuove implementazioni, fornendo un'interfaccia intuitiva per testare i servizi esposti. Il funzionamento di Swagger è illustrato nella Figura 3.13.

Per quanto riguarda i **test di sistema**, le sessioni di *Sprint Review* hanno rappresentato un'importante occasione di verifica, durante la quale le funzionalità sviluppate venivano sottoposte a un processo di valutazione strutturato. Questo momento coinvolgeva attivamente il *Product Owner*, garantendo un confronto diretto sulla conformità delle implementazioni rispetto ai requisiti iniziali.

Durante queste revisioni, oltre al *Product Owner*, erano presenti tutti i membri del *team* di sviluppo che interagivano con il microservizio in questione. Il *Product Owner*, essendo in contatto diretto con gli *stakeholder* e gli esperti di dominio, ha svolto il ruolo di intermediario, assicurando che le soluzioni implementate rispondessero alle esigenze aziendali.

L'azienda, avendo già un quadro chiaro delle richieste degli *stakeholder*, è stata in grado di valutare con precisione il livello di soddisfacimento dei requisiti. Questo approccio ha permesso di ottenere un riscontro immediato e di identificare eventuali aree di miglioramento prima del rilascio definitivo, favorendo un ciclo di sviluppo iterativo e allineato alle necessità del *business*.

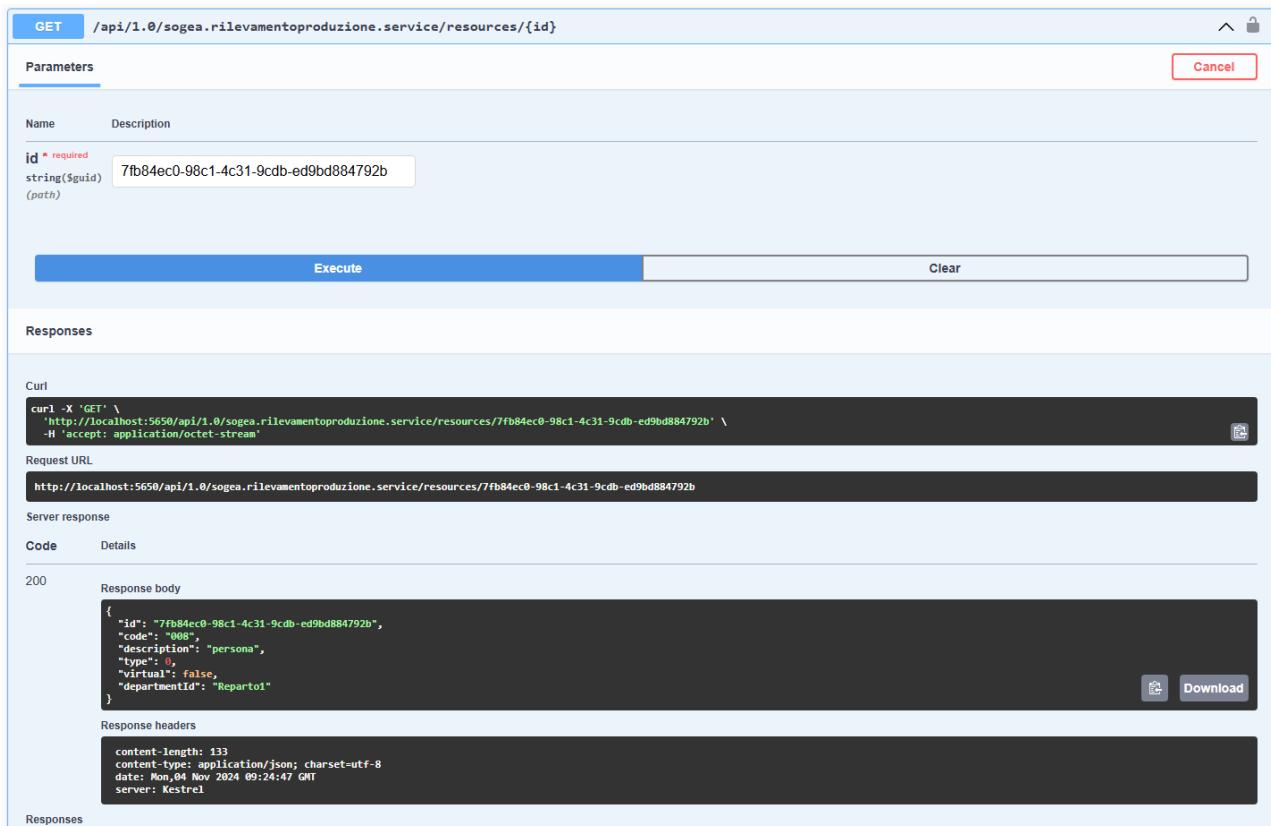


Figura 3.13: Schermata d'esempio di Swagger per la lettura di un dato.

Come anticipato nella Sezione 1.6.4, il processo di validazione fornisce evidenza oggettiva sulle capacità del *software* di soddisfare le aspettative e i bisogni del committente.

Data la natura accademica del mio *stage*, non sono state condotte validazioni formali che avrebbero previsto il coinvolgimento diretto degli *stakeholder* esterni o di un *domain expert*.

Nel mio caso, la **validazione** del lavoro svolto è stata effettuata da diverse figure, a seconda della rilevanza degli avanzamenti apportati: il *Product Owner*, il *Team Leader* (che ha ricoperto anche il ruolo di mio *tutor* aziendale) e, in alcuni casi, gli altri sviluppatori. Sebbene un'attività simile si sia già svolta durante i test di sistema, la validazione avrebbe richiesto un processo più strutturato e formale.

L'obiettivo della validazione è verificare che il *software*, in specifiche condizioni operative, sia in grado di soddisfare gli scopi per cui è stato progettato. Per questo motivo, è essenziale il coinvolgimento diretto degli *stakeholder*. Di conseguenza, si può affermare che la vera validazione sarà effettuata in autonomia da SogeaSoft S.r.l., mentre il mio contributo si è limitato a una fase preliminare di verifica interna.

Idealmente, la validazione di una funzionalità dovrebbe avvenire una sola volta, senza necessità di ripetere il processo. Per raggiungere questo obiettivo, il *team* deve presentare prove oggettive, supportate dalla documentazione, che dimostrino la conformità della funzionalità agli scenari di esecuzione e il pieno soddisfacimento dei requisiti.

Per garantire un processo futuro di validazione efficace, ho dedicato particolare attenzione alla manutenzione e allaggiornamento della documentazione tecnica su Wiki Azure. In questo modo, il codice da me sviluppato è accompagnato da tutte le informazioni progettuali necessarie, facilitando la comprensione e la verifica delle soluzioni implementate.

## 3.3 Risultati raggiunti

### 3.3.1 Il microservizio

L'estrazione del microservizio dal sistema monolitico preesistente ha raggiunto l'obiettivo prefissato, riuscendo a preservare le funzionalità originarie pur trasferendole in un'architettura indipendente. L'esito positivo di questa migrazione è evidenziato dalla piena operatività del componente estratto, ora funzionante come entità autonoma al di fuori del contesto monolitico in cui era originariamente integrato. Al momento è possibile utilizzarlo solo tramite **Swagger**, ma l'azienda prevede di implementare anche i collegamenti con l'interfaccia più intuitiva attualmente in uso.

Il processo di estrazione ha richiesto l'implementazione di soluzioni tecniche che, sotto il profilo dell'eleganza architettonica, non sono perfette. Tuttavia, questi compromessi sono stati consapevolmente accettati in considerazione dell'obiettivo primario del progetto, che consisteva nell'estrazione funzionale del microservizio piuttosto che nella realizzazione di un'architettura ideale dal punto di vista teorico.

Al termine del periodo di *stage*, il microservizio estratto ha raggiunto il livello di autonomia desiderato, pur mantenendo specifici canali di comunicazione con il sistema monolitico originario per garantire la continuità dei processi aziendali. Questa configurazione ha consentito al servizio di assumere la responsabilità esclusiva della gestione dei **rilevamenti** e delle **lavorazioni** associate alle **fasi** produttive, interfacciandosi con il sistema di gestione degli **ordini di produzione** e delle relative fasi (*OrdineProduzioneFase*), come è possibile osservare nella Figura 3.14.

L'estrazione del microservizio ha prodotto un significativo miglioramento nell'efficienza operativa dell'organizzazione, risolvendo problematiche strutturali del sistema precedente, tratte nella Sezione 3.2.1. Inoltre, rispetto alle misurazioni su velocità ed efficienza citate nella Sezione 3.2.2, abbiamo osservato un miglioramento non indifferente soprattutto rispetto all'aggiornamento dei dati in tempo reale.

Infatti, nella configurazione precedente di SAI, l'aggiornamento dei dati avveniva attraverso procedure *batch* notturne, ossia di estrazione di un grande gruppo (*batch*) di dati, eseguite

<b>Jobs</b>	▼
<b>ProductionOrderPhases</b>	▼
<b>Reasons</b>	▼
<b>Resources</b>	▼
<b>Test</b>	▼
<b>Authorization</b>	▼

Figura 3.14: Schermata iniziale di Swagger per la gestione del microservizio **MS\_Rilevamento**

ogni notte, imponendo un ritardo sistematico di 24 ore nella disponibilità delle informazioni aggiornate. Questa latenza costituiva un ostacolo significativo per gli operai, perché questa operazione era spesso soggetta a errori e un operaio doveva attendere diverse ore la risoluzione del problema.

L'implementazione della tecnologia Debezium ha rivoluzionato questo paradigma, introducendo un meccanismo di aggiornamento in tempo reale che elimina questa latenza. Ciò ha permesso ai lavoratori di accedere immediatamente ai dati aggiornati, senza dover attendere l'esecuzione delle procedure *batch*. Tuttavia, questo cambiamento ha comportato un aumento del costo computazionale. Rispetto alle previsioni iniziali sul *Return on Investment*, il beneficio derivante dall'incremento della velocità e dalla riduzione della necessità di intervento umano per la risoluzione di problemi ha compensato i costi aggiuntivi, portando a un bilancio complessivo neutro, senza un guadagno significativo.

Infatti questa transizione verso un modello *event-driven<sub>G</sub>*, ossia una rilevazione e gestione di eventi immediata, comporta un inevitabile compromesso in termini di consumo di risorse computazionali. La maggiore reattività del sistema si traduce in un più intenso utilizzo dell'infrastruttura, richiedendo un bilanciamento attento tra benefici operativi e costi infrastrutturali. Considerando questo *trade-off*, l'implementazione della sincronizzazione in tempo reale è stata strategicamente limitata ai soli dati per i quali la tempestività dell'aggiornamento rappresenta un requisito critico per i processi aziendali.

### 3.3.2 Risultati quantitativi

Durante il periodo di *stage*, ho conseguito con successo **tre** dei quattro obiettivi obbligatori inizialmente concordati, unitamente all'obiettivo facoltativo. L'obiettivo classificato come desiderabile non è stato affrontato, in seguito a una rivalutazione delle priorità progettuali. Questo ha privilegiato lo sviluppo del Proof of Concept (PoC), ossia l'obiettivo facoltativo 1 (FA1), rispetto alla documentazione esaustiva dell'ecosistema dei servizi preesistenti e delle loro interdipendenze (OB3).

Questo cambiamento delle priorità delle attività si è rivelata funzionale al rispetto del vincolo temporale delle 304 ore allocate per l'esperienza formativa. La gestione del tempo ha inoltre consentito l'estensione delle attività oltre il perimetro inizialmente definito, permettendo la realizzazione di un prototipo funzionale che rappresenta l'evoluzione naturale del Proof of Concept sviluppato.

I risultati tangibili prodotti durante il periodo di *stage* comprendono:

- un elaborato di analisi della letteratura riguardo le metodologie di migrazione verso architetture a microservizi;
- un documento tecnico sui *pattern* di migrazione, con particolare enfasi sugli approcci ritenuti più pertinenti per il contesto specifico di SogeaSoft S.r.l.;

- un’analisi formale dei requisiti che sintetizza le esigenze funzionali e non funzionali identificate;
- 5826 linee di codice sorgente, sottoposte a processi di verifica e successivamente integrate nel *branch* di sviluppo del progetto.

## 3.4 Sviluppi futuri

Il progetto di *stage* da me intrapreso riveste una particolare rilevanza per SogeaSoft S.r.l. in quanto si inserisce in un contesto di crescente esigenza di innovazione e miglioramento del sistema *software* SAIonWeb. In seguito all’acquisizione da parte di Bluenext S.r.l., come descritto nella Sezione 1.1, l’applicativo è destinato a gestire un numero sempre maggiore di clienti. Questa evoluzione comporta la necessità di aggiornamenti più frequenti e tempestivi, nonché l’adattamento del *software* alle specifiche esigenze aziendali e al relativo dominio di riferimento.

Un aspetto cruciale risiede nella capacità di garantire ai clienti l’accesso ai dati in tempo reale, superando l’attuale limite che impone tempi di attesa di 24 ore per ottenere i dati del giorno precedente. Per raggiungere tale obiettivo, risulta imprescindibile scomporre il resto dell’architettura monolitica esistente e procedere verso una migrazione a un’architettura completamente basata su microservizi.

Immagino che la prima cosa che caratterizzerà anche i progetti di *stage* futuri sarà l’approfondimento della tecnologia Debezium, proprio a questo proposito. Ma non solo, anche l’esplorazione di nuovi *pattern* risulta fondamentale. Dato che potenzialmente per ogni *bounded context* può corrispondere un microservizio, come farli comunicare, come poterli estrarre, quali compromessi attuare sono sfide che SogeaSoft S.r.l. dovrà affrontare.

In questo contesto dunque, il progetto di *stage* ha costituito un primo passo significativo verso l’obiettivo più ampio della migrazione completa. Pur rappresentando solo una parte del progetto di lungo termine, il lavoro svolto ha permesso di delineare e testare approcci metodologici e tecnici che potranno essere riutilizzati e ampliati nelle fasi successive. La mia attività ha quindi contribuito a gettare le basi per un percorso di trasformazione che, nel medio-lungo periodo, porterà a una maggiore flessibilità e scalabilità del sistema SAIonWeb.

# 4 Valutazione retrospettiva

## 4.1 Soddisfacimento degli obiettivi di *stage*

### Obiettivi aziendali

Descriverò, rispetto alle aspettative iniziali, il grado di soddisfacimento degli obiettivi di *stage* dell'azienda, sia in generale che riguardo ai singoli obiettivi.

### Obiettivi e aspettative personali

Scriverò una valutazione personale generale rispetto all'esperienza, facendo un bilancio rispetto alla Sezione 2.5 riguardo ai singoli obiettivi.

#### 4.1.1 Competenze acquisite

Descriverò le competenze e abilità acquisite, valutando la mia evoluzione in modo oggettivo, sia qualitativo che quantitativo.

#### 4.1.2 Bilancio formativo

Descriverò l'eventuale distanza tra la preparazione accademica e le competenze richieste a inizio dello *stage*.

# 5 Glossario dei termini

## A

---

**Advanced Message Queuing Protocol (AMQP)**: protocollo di messaggistica aperto che consente la comunicazione tra applicazioni distribuite.

**After Sales Service (ASS)**: è l'insieme delle attività di assistenza e supporto fornite al cliente dopo l'acquisto di un prodotto o servizio, con l'obiettivo di garantirne il corretto utilizzo e la soddisfazione.

**Aggregato**: è un insieme coerente di oggetti del dominio, con un'entità principale che garantisce la consistenza e l'integrità dei dati all'interno dei suoi confini.

**Anti-Corruption Layer (ACL)**: è un *pattern* architetturale che crea una barriera tra il dominio di un'applicazione e sistemi esterni, prevenendo che logiche o modelli di altri sistemi influenzino la struttura interna del sistema.

**Application Programming Interface (API)**: insieme di regole e protocolli che consente a diverse applicazioni di comunicare tra loro. Le API definiscono i metodi e le strutture dati necessari per interagire con i servizi e le funzionalità di un sistema.

**Architettura Esagonale**: è un modello di progettazione *software* che separa il *core* dell'applicazione dalle interfacce esterne, come *database*, servizi o interfacce utente, tramite porte e adattatori.

## B

---

**Bounded Context**: è un'unità autonoma all'interno di un sistema *software*, in cui un modello del dominio ha un significato ben definito e non ambiguo, separato dagli altri contesti per evitare incongruenze.

**Branch**: è una diramazione indipendente del codice sorgente che consente di sviluppare nuove funzionalità o apportare modifiche senza influenzare la versione principale del progetto.

**Browser**: *software* per la caccia, la presentazione e la navigazione di risorse sul *Web*.

**Bug**: difetto o errore nel *software* che causa un comportamento imprevisto o indesiderato, compromettendo il corretto funzionamento del programma.

**Business Intelligence (BI)**: è un insieme di tecnologie, processi e pratiche che consentono di raccogliere, analizzare e trasformare i dati aziendali in informazioni utili per supportare il processo decisionale.

## C

---

**Change Data Capture (CDC)**: è una tecnica che rileva e traccia le modifiche apportate ai dati in un *database*, permettendo di aggiornare in tempo reale i sistemi che ne fanno uso.

**Client**: applicazione o dispositivo che richiede servizi o risorse a un *server* in una rete.

**Command Query Responsibility Segregation (CQRS)**: è un modello architettonico che separa le operazioni di modifica dello stato (comandi) da quelle di lettura dei dati (query).

**Customer Relationship Management (CRM)**: è un approccio strategico volto a ottimizzare le interazioni e le relazioni con i clienti al fine di migliorare la soddisfazione e la fidelizzazione.

**Customer Service (CS)**: è il dipartimento o insieme di attività aziendali finalizzate a supportare i clienti prima, durante e dopo l'acquisto di un prodotto o servizio.

## D

---

**Data Transfer Object (DTO)**: è un oggetto utilizzato per trasportare dati tra processi o livelli di un'applicazione, riducendo il numero di chiamate e semplificando il trasferimento delle informazioni.

**Database Management System (DBSM)**: sistema che include sia il *software* che le strutture necessarie per gestire e organizzare i dati. Si occupa della creazione, manutenzione e manipolazione dei database, gestendo l'accesso, la sicurezza e l'integrità dei dati.

**Debito tecnico**: accumulo di problemi o inefficienze nel codice causati da scelte progettuali rapide o soluzioni provvisorie, che richiedono interventi correttivi nel lungo termine.

**DevOps**: è una pratica che unisce lo sviluppo *software* (Dev) e le operazioni IT (Ops) per migliorare la collaborazione, l'automazione e l'efficienza nel ciclo di vita delle applicazioni, dalla progettazione alla distribuzione e manutenzione.

**Docker**: piattaforma di containerizzazione che consente di creare, distribuire ed eseguire applicazioni in ambienti isolati chiamati *container*, garantendo coerenza e portabilità tra diversi sistemi.

**Domain-Driven Design (DDD)**: è un approccio allo sviluppo software che pone al centro la complessità del dominio applicativo, modellandolo attraverso concetti e strutture direttamente ispirate al contesto reale.

**Domain expert**: è una persona con una conoscenza approfondita di un settore specifico, il cui ruolo è fornire informazioni e guida nello sviluppo di un sistema che rispecchi accuratamente le esigenze del dominio applicativo.

## E

---

**Endpoint (API):** è un punto di accesso in un'API che consente di interagire con una risorsa o una funzionalità specifica.

**Enterprise Resource Planning (ERP):** è un insieme di strumenti informatici integrati che supportano la gestione e l'automazione dei processi di un'organizzazione.

**Entità:** è un oggetto del dominio identificato in modo univoco da un attributo distintivo, indipendentemente dai suoi valori o stato.

**Event-driven architecture:** è un'architettura in cui le azioni vengono eseguite in risposta a eventi, ossia cambiamenti di stato o segnali che attivano processi specifici, permettendo un'elaborazione asincrona e reattiva.

## F

---

**Feature:** è una caratteristica o funzionalità specifica di un prodotto o sistema, progettata per soddisfare un'esigenza o migliorare l'esperienza dell'utente.

**Framework:** è un insieme di strumenti, librerie e regole che forniscono una struttura di base per lo sviluppo di applicazioni software, riducendo il lavoro di codifica.

## G

---

**Git:** sistema di controllo versione distribuito che permette di gestire e tracciare le modifiche al codice, facilitando la collaborazione tra sviluppatori e la gestione delle versioni di un progetto. item **Granularità:** in un contesto di sviluppo *software* indica il livello di dettaglio o suddivisione con cui una funzionalità, un modulo o un componente è progettato o implementato.

## H

---

**Handler:** è un componente *software* che gestisce richieste o eventi, traducendoli in operazioni specifiche all'interno di un sistema.

**HTTP:** è un protocollo di comunicazione utilizzato per trasferire dati su rete, principalmente per il caricamento di pagine *web*, tra *client* (come *browser*) e *server*.

## I

---

**Issue Tracking System (ITS):** è un *software* utilizzato per gestire, monitorare e risolvere i problemi o le richieste di miglioramento durante lo sviluppo di un progetto, consentendo una gestione efficiente delle attività e delle priorità.

## J

---

**JavaScript Object Notation (JSON):** è un formato di scambio dati leggero e leggibile, utilizzato per rappresentare strutture dati, spesso impiegato nelle comunicazioni tra *client* e *server*.

## L

---

**Legacy:** nello sviluppo *software*, il termine ***legacy*** si riferisce a un sistema, codice o tecnologia obsoleta che è ancora in uso, spesso difficile da mantenere o integrare con soluzioni moderne, ma che è fondamentale per il funzionamento dell'organizzazione.

## M

---

**Message Broker:** intermediario che gestisce la ricezione, l'instradamento e la distribuzione di messaggi tra applicazioni o servizi, facilitando la comunicazione tra sistemi.

**Microservizio:** è un'architettura *software* che suddivide un'applicazione in piccoli servizi autonomi e indipendenti, ognuno dei quali gestisce una specifica funzionalità. Ogni microservizio può essere sviluppato, distribuito e scalato in modo separato, facilitando l'evoluzione e la manutenzione del sistema nel suo complesso.

**Minimum Viable Product (MVP):** è una versione base di un prodotto che include solo le funzionalità essenziali, utile per validare l'idea sul mercato con il minimo sforzo di sviluppo.

**Monolite:** in un contesto di sviluppo *software* è un'applicazione costruita come un unico blocco indivisibile, in cui tutte le funzionalità e componenti sono strettamente interconnessi e distribuiti come un'unica unità.

## N

---

**Namespace:** è uno spazio di denominazione che raggruppa identificatori univoci, come variabili o funzioni, per evitare conflitti di nome all'interno di un programma.

## P

---

**Pattern:** Un *pattern* è una soluzione ripetibile e collaudata a un problema ricorrente, che può essere applicata in vari contesti per risolvere specifiche difficoltà nel processo di progettazione o sviluppo. I *pattern* architettonici si riferiscono a soluzioni generali per la struttura di un sistema *software*, come il *pattern* a microservizi, che definisce come organizzare i componenti di un sistema. I *pattern* di *design* riguardano soluzioni a problemi di progettazione a livello di

componenti. I *pattern* di comportamento si concentrano su come gli oggetti interagiscono tra loro.

**Payload:** è la parte di un messaggio o di una richiesta che contiene i dati effettivi da elaborare, escludendo intestazioni o informazioni di controllo.

**Pipeline:** in Azure è un processo automatizzato che consente di compilare, testare e distribuire il codice attraverso diverse fasi, facilitando l'integrazione continua e la consegna continua per le applicazioni.

**Plan-Do-Check-Act (PDCA):** metodo iterativo di gestione del lavoro utilizzato per il miglioramento continuo di processi o prodotti. Anche noto come ciclo di Deming.

**Proof of Concept (PoC):** è una dimostrazione pratica volta a verificare la fattibilità o l'efficacia di un'idea o soluzione tecnica. Viene utilizzata per validare un concetto prima di investire risorse nello sviluppo completo.

**Prototipo:** è una versione preliminare di un prodotto o sistema, realizzata per testare e valutare funzionalità, *design* o prestazioni prima della produzione definitiva.

**Pull Request (PR):** è una proposta di modifica al codice in un sistema di controllo versione, che consente di revisionare e integrare le modifiche da un ramo a un altro, tipicamente dalla versione di sviluppo alla versione principale.

## Q

---

**Query:** è una richiesta formulata a un *database* per ottenere, inserire, modificare o eliminare dati in base a criteri specifici.

## R

---

**Repository:** è un archivio centralizzato in cui vengono conservati, gestiti e versionati file e codici sorgente di un progetto, spesso utilizzando un sistema di controllo delle versioni.

**Representational State Transfer (REST):** modello di architettura *software* creato per guidare la progettazione e lo sviluppo dell'architettura di applicazioni di rete. Valorizza l'uso di interfacce standard (API) e il *deployment* distribuito, indipendente e scalabile.

**Return On Investment (ROI):** è un indicatore finanziario che misura la redditività di un investimento, calcolato come il rapporto tra il guadagno ottenuto e il costo sostenuto.

## S

---

**Single-Page Application (SPA):** applicazione *web* che risponde alle richieste dell'utente sovrascrivendo il contenuto della pagina corrente, anziché seguire il comportamento standard di caricare una nuova pagina.

**Stakeholder:** tutte le persone, gruppi o organizzazioni che hanno un interesse diretto o indiretto nei risultati di un progetto o di un'azienda. Possono includere clienti, fornitori, dipendenti, azionisti, enti regolatori e la comunità in generale.

**Structured Query Language (SQL):** linguaggio standard utilizzato per gestire e manipolare dati in un *database* relazionale.

**Supplier Relationship Management (SRM):** è un approccio strategico e gestionale volto a ottimizzare e gestire le relazioni con i fornitori di un'organizzazione.

## U

---

**Ubiquitous Language:** è un linguaggio comune, condiviso tra sviluppatori ed esperti del dominio, che garantisce coerenza nella comunicazione e nella modellazione del sistema *software*.

**Uniform Resource Locator (URL):** è un indirizzo che specifica la posizione di una risorsa su una rete, consentendo di accedervi tramite un protocollo come HTTP.

**User story:** descrizione breve e informale di una funzionalità dal punto di vista dell'utente finale, utilizzata nello sviluppo *Agile* per definire i requisiti in modo chiaro e orientato al valore.

## V

---

**Value Object:** è un oggetto del dominio che rappresenta un concetto privo di identità univoca, definito solo dai suoi attributi e utilizzato per esprimere valori o proprietà immutabili.

**Version Control System (VCS):** è uno strumento che permette di tenere traccia delle modifiche apportate a un progetto nel tempo, consentendo di gestire diverse versioni del codice.

# 6 Lista degli acronimi

## A

**ACL** Anti-Corruption Layer

**AMQP** Advanced Message Queuing Protocol

**API** Application Programming Interface

**ASS** After Sales Service

## B

**BI** Business Intelligence

## C

**CDC** Change Data Capture

**CQRS** Command Query Responsibility Segregation

**CRM** Customer Relationship Management

**CRP** Capacity Requirements Planning

**CS** Customer Service

## D

**DBSM** Database Management System

**DDD** Domain-Driven Design

**DTO** Data Transfer Object

## E

**ERP** Enterprise Resource Planning

## F

**FCS** Finite Capacity Scheduling

## H

**HTTP** Hypertext Transfer Protocol

## I

**ITS** Issue Tracking System

## J

**JSON** JavaScript Object Notation

## M

**MPS** Master Production Scheduling

**MRP** Material Requirements Planning

**MVP** Minimim Viable Product

## P

**PDCA** Plan-Do-Check-Act

**PMI** Piccole Medie Imprese

**PoC** Proof of Concept

**PR** Pull Request

## R

**REST** Representational State Transfer

**ROI** Return On Investment

## S

**SAI** Sistema Aziendale Integrato

**SQL** Structured Query Language

**SRM** Supplier Relationship Management

## U

**URL** Uniform Resource Locator

## V

**VCS** Version Control System