

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 Domuță

Migrazione da un'architettura Monolitica a un'architettura a Microservizi
Tesi di laurea triennale.
Autore: Alessia Domuță
Copyright: © 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

1 Azienda ospitante	7
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 software	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	22
2 Progetto di <i>stage</i>	24
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
3 Svolgimento dello <i>stage</i>	37
3.1 Conoscenza del dominio di applicazione	37
3.2 Attività svolte	37
3.2.1 Analisi dei requisiti	37
3.2.2 Progettazione	37
3.2.3 Implementazione	37

3.2.4	Verifica e Validazione	37
3.3	Risultati raggiunti	38
3.3.1	Il Microservizio	38
3.3.2	Risultati quantitativi	38
3.4	Sviluppi futuri	38
4	Valutazione Retrospettiva	39
4.1	Soddisfacimento degli obiettivi di <i>stage</i>	39
4.1.1	Competenze acquisite	39
4.1.2	Bilancio formativo	39
5	Glossario dei termini	40
6	Lista degli acronimi	45

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

Elenco delle tavelle

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

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_G* (*ERP_G*), 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_G* (*CRM_G*), *Supplier Relationship Management_G* (*SRM_G*), raccolta ordini e *After Sales Service_G* (*ASS_G*), 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_G* (*BI_G*), focalizzandosi sull'analisi dei dati aziendali per ottimizzare le *performance* e supportare decisioni strategiche più informate e mirate.
- **CS (Customer Service_G):** 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.

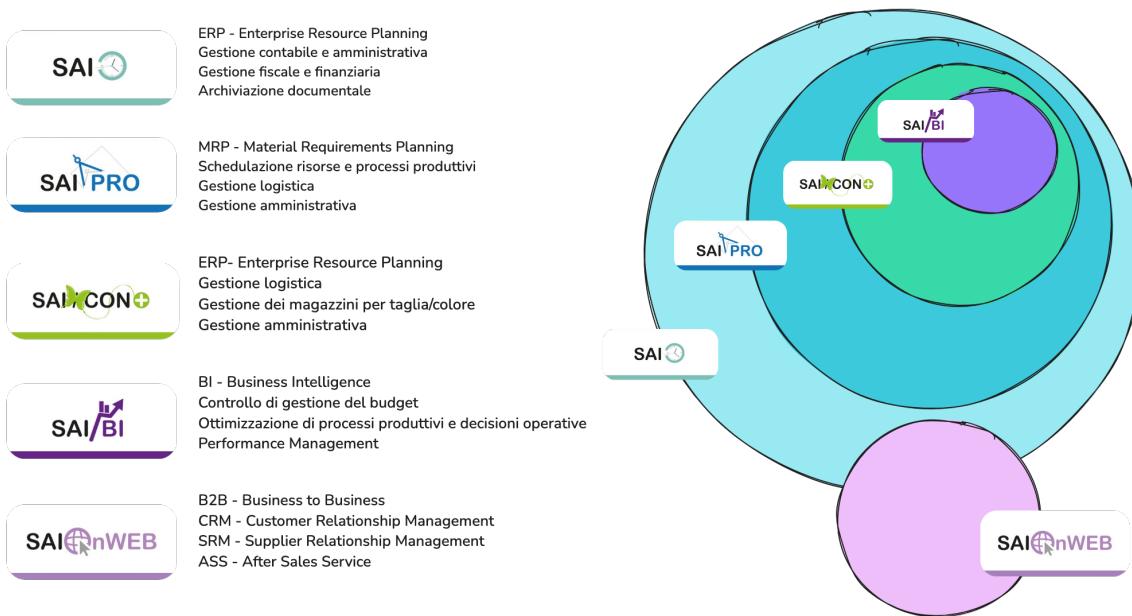


Figura 1.1: Panoramica dei prodotti e servizi offerti da SogeaSoft S.r.l.

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_G* 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_G*, 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.

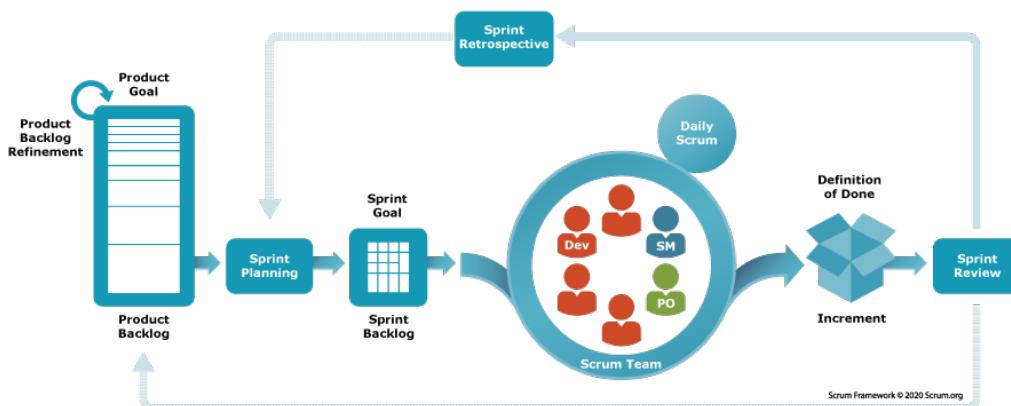


Figura 1.2: Diagramma del flusso di lavoro del modello Scrum, basato sulla filosofia *Agile*.
Fonte: Documentazione Scrum (*ultimo accesso 3/03/2025*)

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.¹

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

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

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_G (*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.²

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_G* (*VCS_G*), che traccia l'evoluzione del *software* in modo strutturato. Le versioni del codice vengono archiviate all'interno di *repository_G*, strutture dati appositamente organizzate per gestire i cambiamenti. Queste *repository* si basano su tre principali tipologie di *branch_G*, 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.³

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.

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

³ISO/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.⁴

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

⁴ISO/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 programmatore 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*.

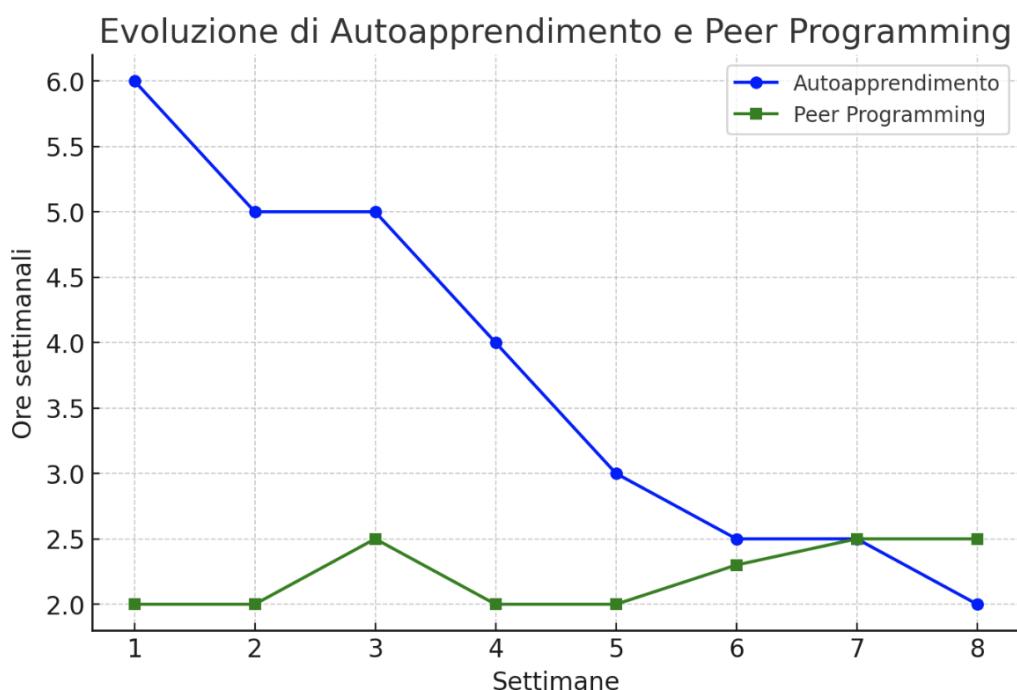


Figura 1.5: Rendicontazione ore dedicate ai processi di formazione durante il mio *stage*

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.⁵ 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*.

⁵I. Sommerville, Software Engineering, 10^a 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.⁶ 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.⁷

Nello specifico SogeaSoft S.r.l. applica il concetto di *Domain – Driven Design*_G (*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.⁸ 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*_G):** 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*_G):** 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*_G:** 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.

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

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

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

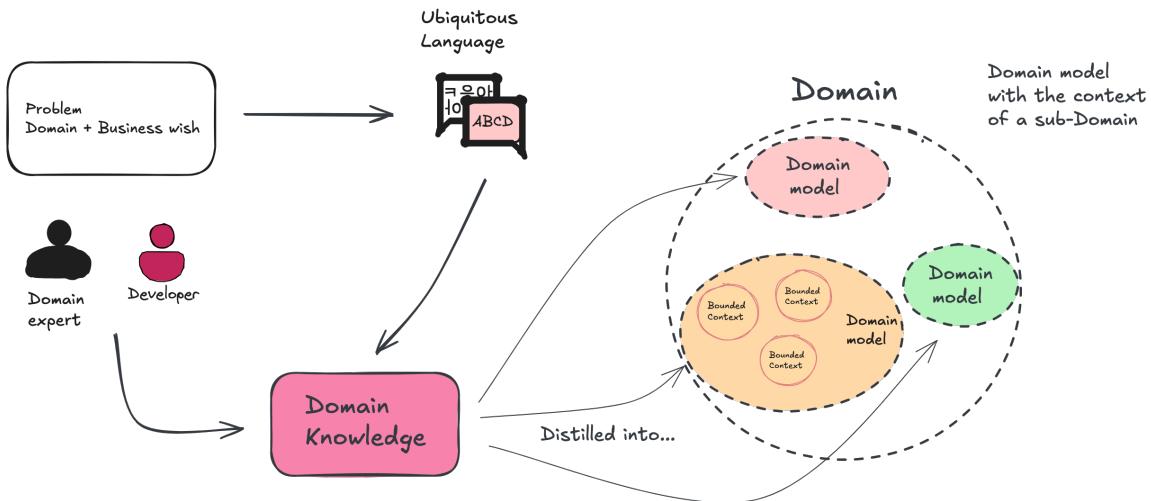


Figura 1.6: Visualizzazione del processo di Analisi dei Requisiti nel contesto DDD.

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.⁹

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 $aggregati_G$, le $entità_G$, i $value\ objects_G$ e le funzioni di dominio, tutti elementi di progettazione nel contesto DDD.
- **Definizione degli eventuali microservizi_G:** 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_G 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.¹⁰ 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.

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

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

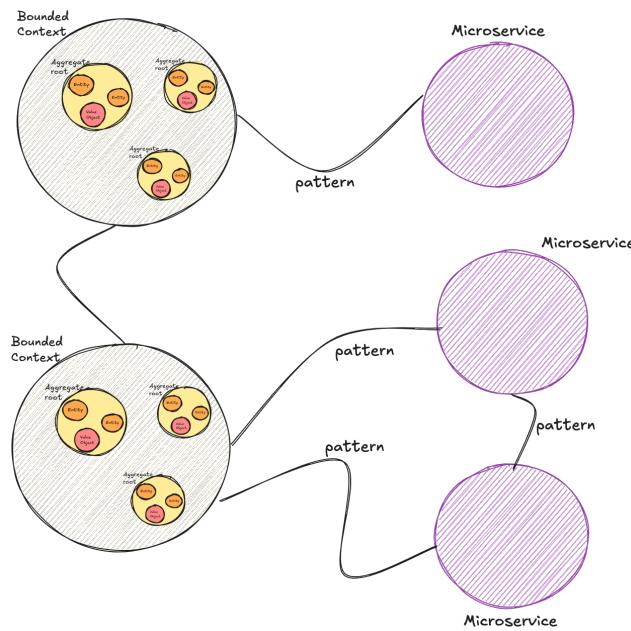


Figura 1.7: Schema dell’attività di Progettazione nel contesto DDD

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.¹¹

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_G* (*ITS_G*). 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.¹²

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*.

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

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

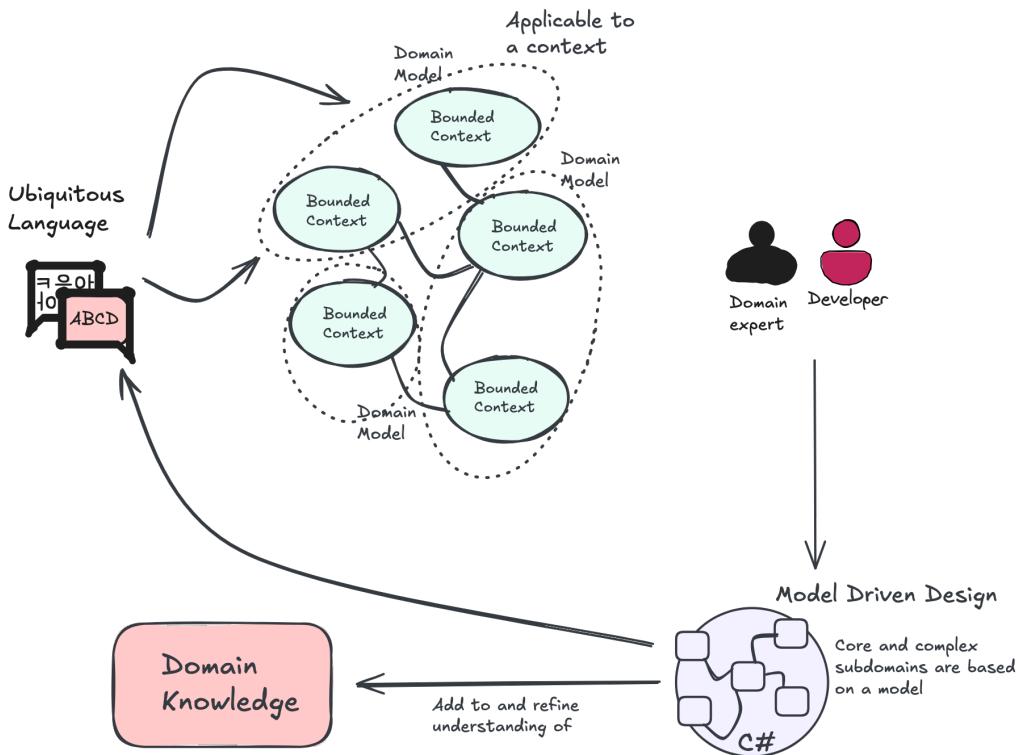


Figura 1.8: Come DDD permea il ciclo di vita di un *software*.

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_G*" (nuova funzionalità) o "*bug_G-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_G* (*PR_G*). 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.¹³ 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;

¹³ISO/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*_G. 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.¹⁴

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.¹⁵

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:

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

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

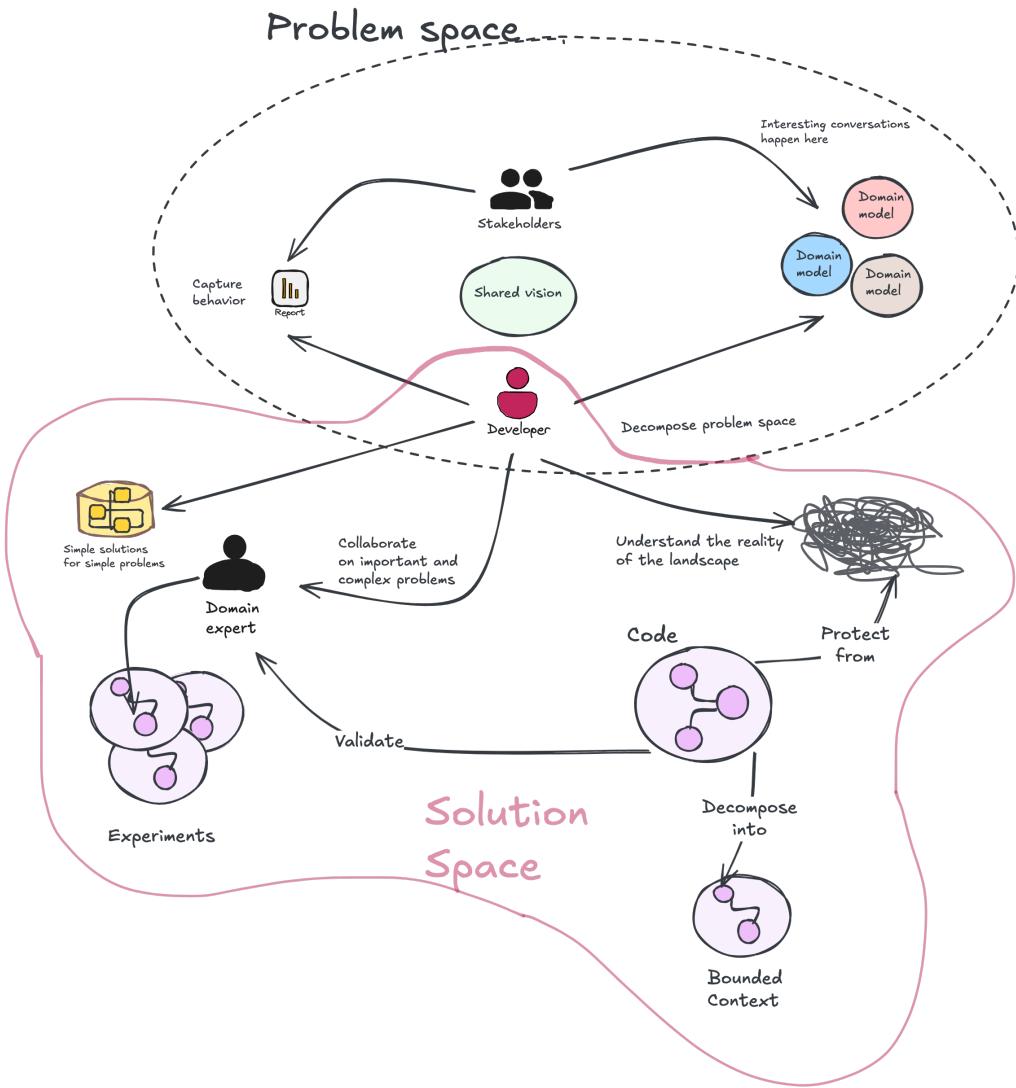


Figura 1.9: Visione d'insieme dell'utilizzo di DDD nello sviluppo *software*.

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_G* 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* multiplataforma 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_G* (SPA_G) 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_G e microservizi, supportando C#.

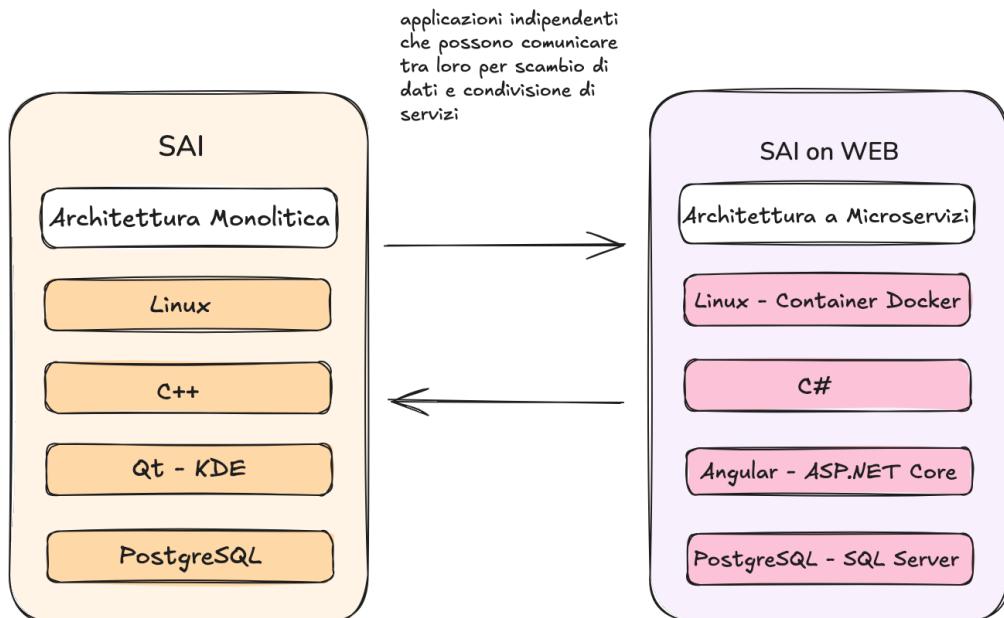


Figura 1.10: Schema delle tecnologie utilizzate da SogeaSoft S.r.l. Informazioni apprese durante il processo di formazione

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_G* 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_G*, 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_G* (*REST_G*) per lo scambio di dati tra i moduli. Questo modello architettonico prevede che le informazioni transitino tramite connessioni dedicate, note come *Application Programming Interfaces_G* (*API_G*). 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_G (AMQP_G)**: 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_G* *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_G* API direttamente dal *browser_G* senza bisogno di scrivere codice aggiuntivo.
- **DBeaver**: *client_G* utilizzato per accedere, interrogare e manipolare *database* relazionali basati sul *Structured Query Language_G* (*SQL_G*), 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_G* (*DBMS_G*), 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

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'impiego di tecnologie relativamente recenti e, nel caso di Debezium (Sezione 1.7), non ufficialmente documentate per questa combinazione specifica¹⁶.

¹⁶Fonente: 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_G* (*PoC_G*) o prodotti già pronti per un utilizzo immediato, come il *Minimum Viable Product_G* (*MVP_G*). 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_G, la cui differenza è visibile nella Figura 2.1); nonché la produzione di una documentazione tecnica approfondita.

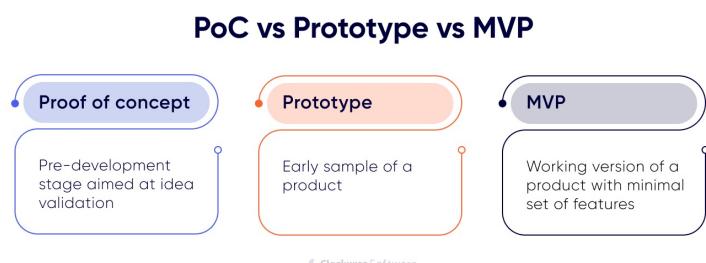


Figura 2.1: Differenza tra PoC, Prototipo, MVP. Fonte: <https://clockwise.software/blog/proof-of-concept-in-software-development/> (ultimo accesso 12/03/2025)

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_G* a un'architettura a *microservizi_G*, 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 finale¹.

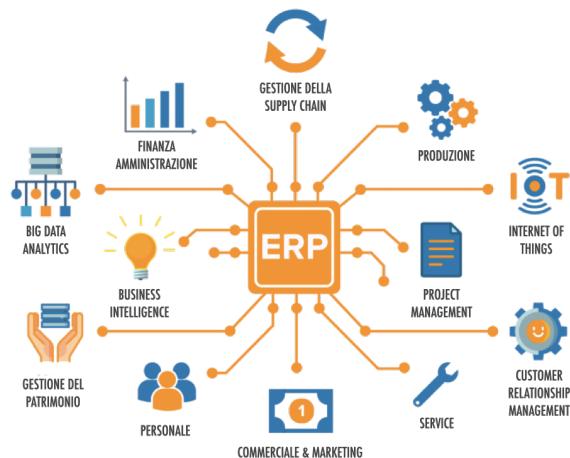


Figura 2.2: Esempio di un ERP generico. Fonte:<https://www.bo.camcom.gov.it/it/promozione-interna/sistemi-gestionali-erp> (*ultima visita 15/03/2025*)

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.

¹Fonte: <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.

Codice	Descrizione	Tipologia	Stato
0001	Taglio	Lavoro	Configurato
0002	Formatura	Lavoro	Configurato
0003	Lavorazione dei materiali	Lavoro	Configurato

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 sistema².

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 tecnico³ può far sì che il *software* diventi eccessivamente complesso e difficilmente adattabile a nuove tecnologie³;
- **scalabilità verticale:** la scalabilità in un sistema monolitico si ottiene potenziando l'intera applicazione per affrontare un aumento del carico di lavoro³. 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_G:** l'architettura monolitica presenta una scarsa separazione delle funzionalità, che risultano strettamente integrate e difficilmente isolabili⁴. 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*)⁵** 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.

²O. 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.

³J. Fritzs, 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.

⁴M. 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.

⁵Fonete: <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* autonoma⁶. 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 corrispondente⁷, 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 applicativo⁸, evitando di creare monoliti stratificati e complessi da gestire (come ad esempio SAI);
- **alta granularità:** l'architettura a microservizi è caratterizzata da un'elevata granularità⁹, 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**⁹, che garantiscono una chiara demarcazione tra i diversi ambiti di competenza e funzionalità.
- **database distribuito:** i microservizi dispongono di database indipendenti¹⁰. 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**⁹. Questo approccio riduce il forte accoppiamento (*loose coupling*)¹¹ 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.

⁶O. 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.

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

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

⁹M. 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

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

¹¹Fonte:<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.

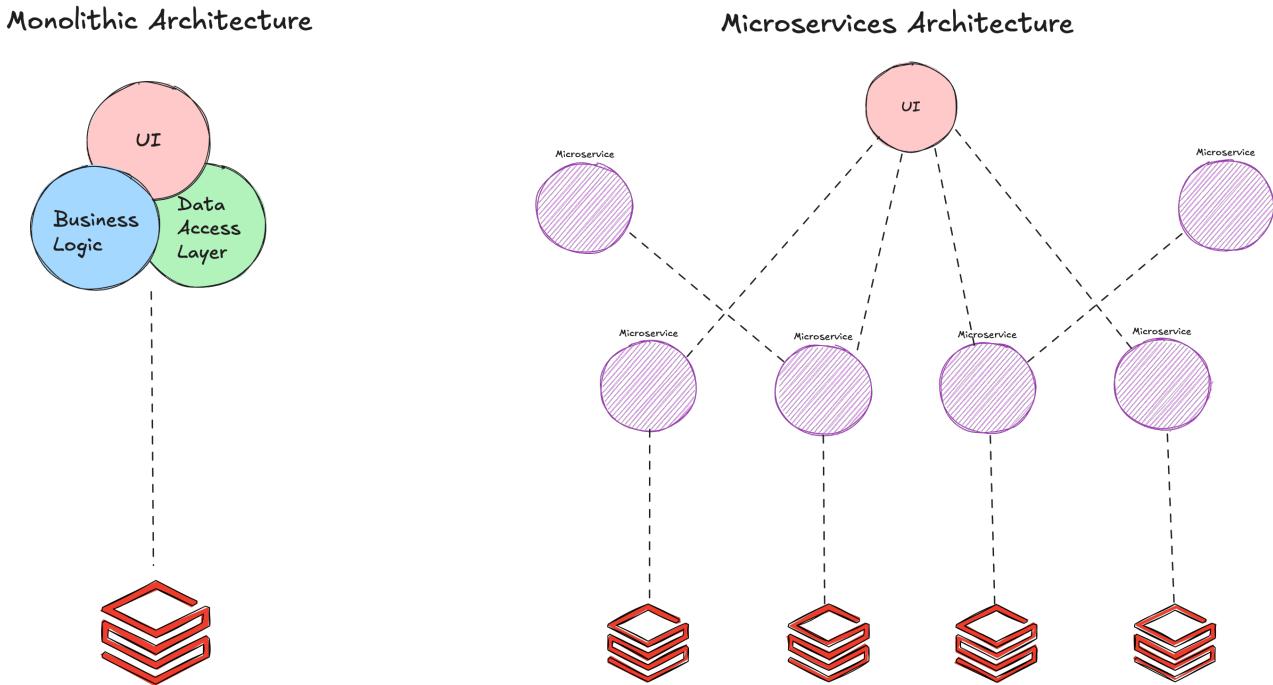


Figura 2.4: Rappresentazione grafica della suddivisione in microservizi

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_G* (*ACL_G*). In ambito architettonico, un ACL è un modello progettuale che funge da interfaccia tra sistemi *legacy_G* 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_G*, 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* (*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.

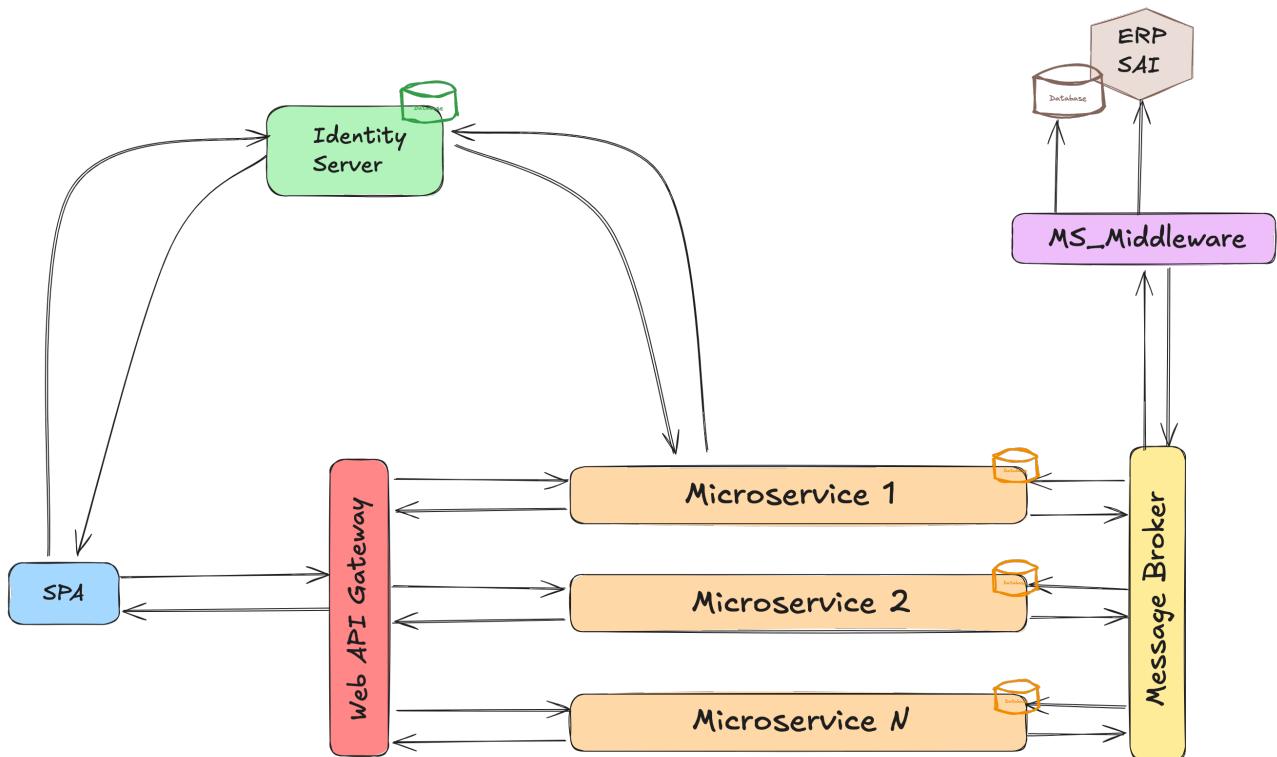


Figura 2.5: Piano di migrazione di SogeaSoft S.r.l.. Fonte: informazioni apprese durante il processo di formazione.

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*.

Obiettivi aziendali	
Obbligatori	
OB1	Studio della letteratura esistente sulle architetture monolitiche, sulle architetture a microservizi e sui metodi di migrazione
OB2	Documentazione relativa ai requisiti
OB3	Documentazione dei servizi esistenti e delle relazioni tra essi
OB4	Individuazione di un piano indicativo
Desiderabili	
DE1	Documentazione dei rischi
Facoltativi	
FA1	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
24	Documentazione dei servizi esistenti e delle relazioni tra essi e del piano
16	Identificazione dei potenziali rischi e delle sfide associate alla migrazione
24	Inizio dello sviluppo di un Proof of Concept
16	Documentazione dei rischi
8	Individuazione di un piano indicativo di migrazione
24	Proseguimento del Proof of Concept
16	Valutazione teorica dell'architettura
32	Revisione della documentazione e del Proof of Concept
304 ore totali	

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.

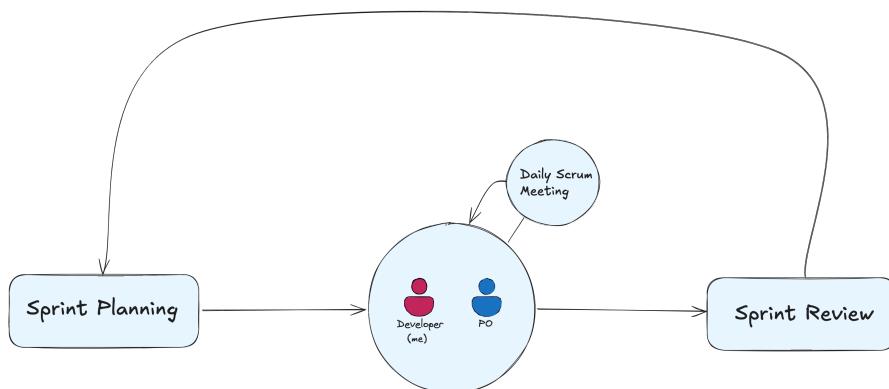


Figura 2.6: Rappresentazione grafica dell'attività di sviluppo effettiva che ha caratterizzato il mio *stage* secondo *Agile*, nel contesto Scrum.

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*¹².

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.

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

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

Descriverò, seguendo il processo a cui sono stata esposta per lo svolgimento di questo progetto, la prima fase: acquisire familiarità con il dominio di applicazione. Dunque riprenderò la Sezione 2.2.2 descrivendo in breve il "punto di partenza". Introdurrò il singolo modello di Dominio preso in considerazione, meglio descritto nella sezione successiva.

3.2 Attività svolte

3.2.1 Analisi dei requisiti

Riprendendo la sezione 1.6.1, descriverò nel dettaglio: L'individuazione nel dettaglio del modello di dominio, l'individuazione di *bounded contexts*, i casi d'uso presi in esame, il linguaggio comune utilizzato per una comprensione univoca di tutte le parti coinvolte.

Requisiti

Descriverò nel dettaglio i requisiti singoli individuati per i singoli casi d'uso.

3.2.2 Progettazione

Riprendendo la Sezione 1.6.2, descriverò nel dettaglio: la modellazione del dominio, la definizione dei microservizi individuati, e i *pattern* individuati per la comunicazione tra essi, nonché per la sincronizzazione dei dati tra monolite e il microservizio preso in esame.

3.2.3 Implementazione

Descriverò l'attività di implementazione, riprendendo la sezione relativa nel primo capitolo. Racconterò nel dettaglio i miei *task*, le difficoltà incontrate e le soluzioni ottenute, sia riguardo all'estrazione del microservizio che riguardo la gestione del *database*. Descriverò i *pattern* scelti, come li ho implementati, ed eventuali risultati rilevanti.

3.2.4 Verifica e Validazione

Racconterò il processo di verifica dei risultati raggiunti nella precedente sezione: dai *test* unitari alla validazione con il *team* di sviluppo.

3.3 Risultati raggiunti

3.3.1 Il Microservizio

Descriverò le funzionalità del servizio estratto e l'efficacia dell'aggiornamento dei dati con il monolite. Scriverò una visione qualitativa degli obiettivi raggiunti, il loro allineamento al modello di dominio individuato.

3.3.2 Risultati quantitativi

Descriverò i risultati quantitativi raggiunti: sia riguardo alla documentazione effettivamente scritta, al codice sviluppato, eventuali *report* e diagrammi, sia riguardo al miglioramento della *performance* del sistema in generale, il *tradeoff* tra miglioramenti e rallentamenti, in base agli obiettivi del progetto (e quindi del dominio).

3.4 Sviluppi futuri

Racconterò di quanto il mio progetto di stage sia rilevante per SogeaSoft S.r.l. rispetto al loro obiettivo di migrazione completa del loro sistema verso un'architettura a microservizi.

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 l'acquisizione, 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

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

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

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.

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.

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.

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.

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.

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.

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.

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*.

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

CRM Customer Relationship Management

CRP Capacity Requirements Planning

CS Customer Service

D

DBSM Database Management System

DDD Domain-Driven Design

E

ERP Enterprise Resource Planning

F

FCS Finite Capacity Scheduling

I

ITS Issue Tracking System

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

S

SAI Sistema Aziendale Integrato

SQL Structured Query Language

SRM Supplier Relationship Management

V

VCS Version Control System