Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

UNIVERSITÀ DI PADOVA

Sviluppo sistema modulare per processi asincroni con aggiornamenti in diretta

Tesi di Laurea

Relatore

Prof. Tullio Vardanega

Laureando

Alessandro Contarini

Matricola 2101052

Anno Accademico 2025-2026

Sommario

Il presente documento descrive il lavoro svolto durante lo *stage* curricolare presso RiskAPP S.r.l., dedicato allo sviluppo di un sistema modulare per la gestione di processi asincroni con aggiornamenti in tempo reale in un'applicazione *web*.

Organizzazione del testo

- Il primo capitolo descrive il contesto aziendale in cui si è svolto lo *stage*, presentando RiskAPP S.r.l. e il settore *insurtech*$_G$ in cui opera. Vengono inoltre illustrate la clientela di riferimento e i relativi bisogni, le attività e le soluzioni offerte dall'azienda, i processi interni osservati, le tecnologie e gli strumenti utilizzati e l'attenzione dell'azienda verso l'innovazione.
- Il secondo capitolo presenta lo *stage*, descrivendo il rapporto tra lo *stage* e l'azienda, l'ambiente di lavoro in cui lo stagista è stato inserito e il problema affrontato. Sono inoltre illustrati gli obiettivi aziendali, i vincoli tecnologici e temporali, il metodo di lavoro adottato, insieme alle motivazioni personali che hanno portato alla scelta della proposta.
- Il terzo capitolo illustra il progetto realizzato, partendo dall'analisi del problema e dalla progettazione dell'architettura della soluzione. Vengono poi descritte le attività di codifica e integrazione dei componenti, con riferimento al microservizio per processi asincroni, al *backend* applicativo, al *frontend* e alle librerie di integrazione. Il capitolo presenta infine le attività di verifica e validazione, la documentazione prodotta e i risultati ottenuti.
- Il quarto capitolo propone una valutazione retrospettiva dell'esperienza di *stage*, considerando il soddisfacimento degli obiettivi aziendali e personali, la maturazione professionale raggiunta e il rapporto tra le competenze acquisite nel percorso di studi e quelle richieste nello svolgimento del progetto.

Convenzioni tipografiche

Durante la stesura del testo sono state adottate le seguenti convenzioni tipografiche:

- I termini inclusi nel glossario sono contrassegnati nell'elaborato dalla lettera "G" a pedice e collegati alla relativa voce (per esempio lead$_G$).
- I termini presenti nell'elenco degli acronimi e abbreviazioni sono contrassegnati nel testo dalla lettera "A" a pedice e collegati alla relativa voce (per esempio HTTP$_A$).
- I termini in lingua diversa dall'italiano sono scritti in corsivo.
- Le figure e le tabelle prive di indicazione della fonte sono da considerare di produzione originale.

Ringraziamenti

Vorrei ringraziare prima di tutto i miei genitori, che mi sono stati vicini in questi anni e mi hanno sempre sostenuto. Senza il loro aiuto, la loro pazienza e la fiducia che hanno avuto in me, arrivare a questo traguardo sarebbe stato molto più difficile.
Vorrei ringraziare anche le persone che mi sono state accanto durante questo percorso, nei momenti belli e in quelli più complicati. Grazie a chi mi ha ascoltato, incoraggiato e aiutato quando ne avevo bisogno.
Ringrazio il Prof. Tullio Vardanega, relatore di questa tesi, per la disponibilità e la rapidità con cui ha saputo seguirmi e supportarmi durante lo stage e la stesura della relazione.
Un ringraziamento sincero va a Marco Mazzucato e a tutto il team RiskAPP, per avermi accolto e seguito durante questa prima esperienza nel mondo del lavoro.
Padova, Luglio 2026

Alessandro Contarini

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Il contesto aziendale

1.1 RiskAPP e il settore *insurtech*

Le informazioni riportate in questo capitolo derivano principalmente dall'osservazione diretta svolta durante il periodo di *stage*, integrata dal confronto con il tutor aziendale e con alcuni membri del *team* tecnico. Il capitolo non ha quindi l'obiettivo di ricostruire in modo completo la storia o la strategia aziendale, ma di descrivere il contesto nel quale sono stato inserito, il tipo di problemi affrontati e l'ambiente tecnico-organizzativo in cui si è collocato il progetto.

Profilo dell'azienda
RiskAPP S.r.l. è un'azienda *insurtech*$_G$, cioè una realtà che applica tecnologie digitali, automazione e analisi dei dati al settore assicurativo. Fondata nel 2015 a Conselve, in provincia di Padova, conta circa quaranta professionisti IT_A con competenze legate sia allo sviluppo *software* sia al dominio assicurativo. Questa doppia competenza è un elemento rilevante, perché le soluzioni sviluppate non si limitano a informatizzare attività generiche, ma devono rappresentare correttamente processi, vincoli e terminologie proprie del mercato assicurativo.
L'azienda opera in particolare nell'ambito delle *Commercial Lines*$_G$, ossia delle coperture rivolte ad aziende, professionisti e organizzazioni. In questo contesto la gestione di una polizza non coincide con una singola operazione, ma comprende un insieme di passaggi collegati tra loro: raccolta delle informazioni, analisi del rischio, quotazione, gestione documentale, emissione, monitoraggio e, in alcuni casi, attività successive legate al portafoglio o alla relazione con intermediari e clienti.

Problemi generali osservati
Durante lo *stage* ho osservato come RiskAPP affronti principalmente problemi legati alla complessità operativa dei processi assicurativi. Le informazioni da gestire sono spesso numerose, eterogenee e distribuite tra soggetti diversi. Una pratica può coinvolgere dati anagrafici, documenti tecnici, valutazioni commerciali, comunicazioni tra operatori e stati di avanzamento che devono rimanere coerenti nel tempo.
Da questa complessità derivano alcune esigenze ricorrenti:

- ridurre la frammentazione prodotta dall'uso di strumenti separati, come fogli di calcolo, scambi di posta elettronica e archivi documentali non collegati;
- rendere tracciabili le operazioni svolte, così da sapere chi ha modificato una pratica, quando e con quale effetto sul processo;
- mantenere visibile lo stato di avanzamento delle attività, evitando che una pratica resti sospesa o difficile da ricostruire;
- limitare attività manuali e ripetitive, soprattutto quando riguardano dati già presenti nel sistema o documenti ricorrenti;
- integrare nuove funzionalità con applicativi e procedure già esistenti presso il cliente.

Il contesto aziendale osservato è quindi caratterizzato dall'incontro tra competenze tecnologiche e conoscenza del dominio assicurativo. La tecnologia non viene introdotta come elemento autonomo, ma come mezzo per rendere più ordinati, controllabili e digitalizzati processi che, se gestiti tramite strumenti separati o scambi informali, rischiano di diventare difficili da monitorare.

1.2 Clientela di riferimento e bisogni

La clientela di RiskAPP è composta principalmente da organizzazioni che operano nel mercato assicurativo e finanziario: compagnie assicurative, banche, intermediari e *broker*. Non si tratta quindi di utenti finali generici, ma di soggetti che partecipano, con ruoli diversi, alla gestione, alla distribuzione o al controllo di prodotti assicurativi.

Tipologie di clienti
Le dimensioni e la struttura dei clienti possono variare in modo significativo. Da un lato vi sono realtà strutturate, come compagnie assicurative o gruppi bancari, caratterizzate da volumi elevati, procedure consolidate, canali distributivi articolati e integrazioni con più sistemi informativi. Dall'altro lato vi sono intermediari e *broker* di dimensioni più contenute, spesso concentrati su specifici portafogli, territori o segmenti di clientela.
Questa varietà rende necessario progettare soluzioni adattabili, perché lo stesso processo assicurativo può assumere forme diverse a seconda dell'organizzazione che lo utilizza. Una compagnia può avere bisogno di controllare prodotti, tariffe e portafogli; una banca può essere interessata alla coerenza con i propri canali commerciali; un intermediario o un *broker* può dare maggiore priorità alla rapidità di quotazione, alla gestione ordinata della documentazione e alla produzione di materiali chiari per il cliente finale.

Bisogni comuni
Nonostante le differenze organizzative, i bisogni osservati possono essere ricondotti a un'esigenza comune: rendere più controllabili processi assicurativi che coinvolgono dati, documenti, ruoli e sistemi differenti. In particolare, i clienti hanno bisogno di:

- strumenti che raccolgano le informazioni in un punto riconoscibile del processo;
- flussi operativi che riducano passaggi manuali e duplicazioni di dati;
- stati di avanzamento chiari e consultabili dagli operatori coinvolti;
- documentazione organizzata e collegata alla pratica corretta;
- possibilità di configurare prodotti, regole e passaggi in base al contesto aziendale;
- integrazione con sistemi già presenti, senza imporre una sostituzione completa dell'ambiente operativo del cliente.

Il segmento principale in cui l'azienda opera è quello delle polizze *corporate* e delle piccole e medie imprese (PMI_A), appartenenti all'ambito delle *Commercial Lines*$_G$. In questo contesto il bisogno riguarda l'intero ciclo operativo: dalla raccolta del *lead*$_G$, cioè il primo contatto commerciale potenzialmente interessato a una copertura, alla gestione del *funnel*$_G$ commerciale, passando per la quotazione, l'emissione del titolo e l'incasso.
Il prodotto *software* deve quindi mediare tra esigenze commerciali, operative, informative e regolative. Da questo punto di vista, il valore non consiste solo nell'automatizzare singole attività, ma nel mantenere collegate le informazioni lungo le diverse fasi del processo assicurativo, lasciando al tempo stesso margine di adattamento ai diversi contesti organizzativi.

1.3 Attività e soluzioni offerte

Le attività di RiskAPP si inseriscono nei bisogni appena descritti attraverso la progettazione e la realizzazione di soluzioni digitali per il settore assicurativo. L'azienda non offre strumenti informatici isolati, ma applicazioni pensate per accompagnare i flussi operativi dei clienti nelle diverse fasi del processo.

Digitalizzazione del ciclo assicurativo
Una parte centrale dell'offerta riguarda la gestione digitale del ciclo operativo assicurativo. Le funzionalità osservate supportano attività come:

- raccolta e organizzazione delle informazioni sul cliente o sul rischio;
- gestione delle opportunità commerciali e delle trattative;
- quotazione dei prodotti e confronto tra opzioni disponibili;
- produzione, raccolta e ordinamento della documentazione;
- monitoraggio delle pratiche e controllo degli stati del processo;
- predisposizione di dati utili ad analisi, *report* e decisioni operative.

La Figura 1.1 rappresenta in forma schematica le principali fasi del ciclo operativo appena descritto. Il diagramma non intende dettagliare tutte le varianti possibili del processo, ma rendere più intuitivo il passaggio da un primo contatto commerciale alla gestione e al monitoraggio della pratica assicurativa.

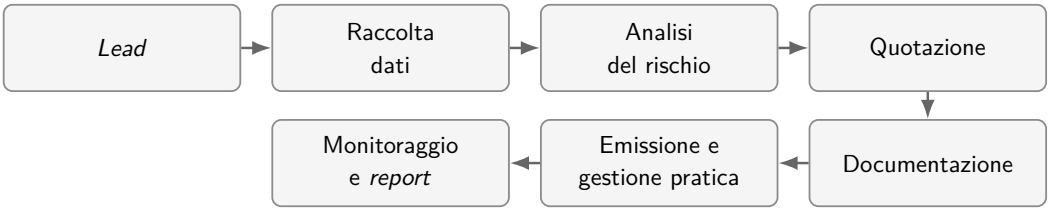


Figura 1.1: Schema semplificato del ciclo operativo assicurativo digitalizzato.

La piattaforma consente quindi di gestire in modo più unitario informazioni, documenti, comunicazioni e stati di avanzamento. Questo riduce la frammentazione che può nascere dall'uso di strumenti separati e permette agli operatori di consultare e aggiornare le informazioni con maggiore continuità.

Configurazione, integrazione e adattamento
Alle funzionalità di prodotto si affiancano attività di analisi, configurazione e integrazione. Poiché compagnie, banche, intermediari e *broker* lavorano con prodotti, canali e procedure differenti, le soluzioni devono poter essere adattate al contesto del singolo cliente senza perdere una base applicativa comune.
La configurabilità è importante perché permette di rappresentare regole operative, passaggi approvativi e differenze di prodotto senza dover riscrivere ogni volta l'intero sistema. Allo stesso tempo, l'integrazione con ambienti esistenti consente alla piattaforma di dialogare con altri applicativi, riducendo il rischio che il nuovo strumento diventi un ulteriore elemento isolato. In questo modo RiskAPP risponde ai bisogni dei clienti combinando competenza assicurativa, configurabilità del prodotto e sviluppo *software*.

Ruolo dell'automazione
Un ulteriore elemento dell'offerta riguarda l'automazione di attività ripetitive o ad alto contenuto informativo. In un processo assicurativo molti passaggi richiedono di leggere documenti, classificare informazioni, verificare dati mancanti o produrre sintesi utili agli operatori. L'automazione permette di ridurre parte di questo carico, ma deve essere progettata in modo da mantenere il controllo umano sui passaggi più significativi.
Nel contesto osservato, l'automazione non sostituisce il ruolo dell'operatore, ma lo supporta rendendo più accessibili le informazioni e più rapidi alcuni passaggi. Questo aspetto è particolarmente rilevante nei processi assicurativi, dove la correttezza del dato e la comprensibilità delle decisioni hanno un peso operativo e regolativo.

Collegamento tra bisogni e soluzioni
Il rapporto tra bisogni dei clienti e soluzioni offerte può essere ricondotto a un

principio comune: **la risposta tecnologica non è identica per tutti i clienti, ma nasce dall'adattamento di una base applicativa comune a processi differenti**. In questo modo la stessa piattaforma può sostenere esigenze commerciali, operative e informative diverse, mantenendo però un obiettivo condiviso, ossia rendere il processo assicurativo più controllabile, integrato e coerente nel tempo.
La Tabella 1.1 sintetizza il collegamento tra alcuni problemi ricorrenti osservati nei processi assicurativi, il bisogno espresso dai clienti e la risposta applicativa che RiskAPP tende a proporre. L'inserto è utile per ricondurre l'elenco dei bisogni a una logica più generale: partire da una criticità operativa e trasformarla in una funzionalità o in una caratteristica della piattaforma.

Problema osservato	**Bisogno del cliente**	**Risposta applicativa**
Dati distribuiti tra strumenti diversi.	Unificare le informazioni della pratica.	Piattaforma centralizzata.
Passaggi manuali e ripetitivi.	Ridurre duplicazioni ed errori.	Automazione dei flussi.
Stato della pratica poco visibile.	Rendere consultabile l'avanzamento.	Stati di processo tracciati.
Procedure diverse tra clienti.	Adattare il sistema al contesto.	Configurazione di prodotti e regole.
Sistemi informativi già presenti.	Evitare strumenti isolati.	Integrazione con applicativi esterni.

Tabella 1.1: Sintesi del rapporto tra problemi operativi dei clienti, bisogni applicativi e risposte offerte dalla piattaforma.

1.4 Processi interni

All'interno di RiskAPP il lavoro tecnico è organizzato in modo da sostenere l'evoluzione continua degli applicativi e, al tempo stesso, mantenere coerenza tra le diverse parti del prodotto. Nel periodo di *stage* ho potuto osservare come questa impostazione si traduca sia nella suddivisione delle responsabilità di sviluppo, sia nelle attività ricorrenti necessarie per mantenere aggiornati e allineati i sistemi.

1.4.1 Sviluppo

Il processo di sviluppo è organizzato attorno a tre principali aree di competenza:

- *backend*, dedicata alla logica applicativa, alla gestione dei dati, alle integrazioni e ai servizi;
- *frontend*, dedicata alle interfacce utente, alla presentazione delle informazioni e all'interazione con gli operatori;
- intelligenza artificiale, dedicata alle funzionalità di automazione, analisi e supporto informativo.

La Figura 1.2 sintetizza questa organizzazione mettendo in evidenza il rapporto tra coordinamento tecnico e aree specialistiche. Lo schema serve a mostrare che le responsabilità sono distribuite per competenza, ma restano collegate da un livello di coordinamento comune.

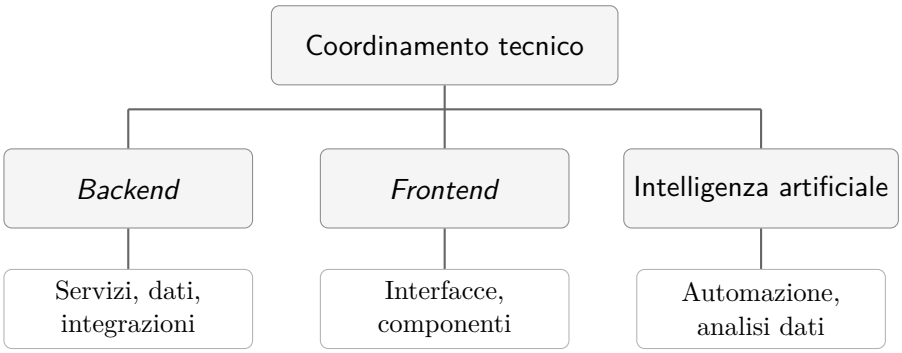


Figura 1.2: Schema delle principali aree tecniche osservate e del coordinamento comune tra responsabilità specialistiche.

Questa suddivisione permette di separare responsabilità tecniche diverse, mantenendo al tempo stesso un coordinamento comune sul prodotto. Per ciascuna area è presente un responsabile tecnico, incaricato di coordinare il *team* dal punto di vista tecnologico. Il suo ruolo consiste nel dare coerenza alle scelte implementative, supportare gli sviluppatori nelle decisioni più rilevanti e verificare che le soluzioni adottate siano compatibili con l'ecosistema aziendale.
Questa organizzazione è utile soprattutto quando una funzionalità coinvolge più componenti. Una modifica visibile all'utente, ad esempio, può richiedere interventi sull'interfaccia, sul *backend*, sulle integrazioni e sulle strutture dati. La presenza di aree specializzate e di responsabili tecnici aiuta a mantenere allineate le diverse parti dell'applicazione.

1.4.2 Manutenzione

La manutenzione degli applicativi viene svolta periodicamente, sia sulla base delle segnalazioni ricevute dagli utenti o dal *team*, sia in seguito all'individuazione di dipendenze con criticità note. Questo tipo di attività consente di intervenire su problemi emersi nell'uso quotidiano, ma anche di prevenire rischi legati a librerie non aggiornate o a componenti esterne che richiedono attenzione.
Un ulteriore aspetto della manutenzione riguarda l'allineamento del codice che utilizza librerie interne sviluppate dall'azienda. Quando queste librerie evolvono, gli applicativi che le impiegano devono essere aggiornati in modo coerente, così da mantenere uno stesso *stack* tecnologico. Questo riduce il rischio di errori dovuti a disallineamenti tra ambienti, versioni o componenti condivise.
La manutenzione, quindi, non riguarda soltanto la correzione di errori già visibili, ma comprende anche attività di prevenzione, aggiornamento e uniformazione tecnica. Queste sono necessarie per mantenere nel tempo applicativi che devono continuare a integrarsi con prodotti, librerie e ambienti diversi.

1.4.3 Organizzazione

L'organizzazione interna combina specializzazione tecnica e coordinamento trasversale. Ogni settimana è prevista una riunione di allineamento dell'intero *team*, utile per condividere lo stato generale delle attività e mantenere una visione comune sulle priorità. A questa si affiancano riunioni dedicate alle singole aree specialistiche, nelle quali vengono affrontati aspetti più specifici legati a *backend*, *frontend* e intelligenza artificiale.
Oltre agli incontri programmati, il confronto avviene anche all'occorrenza, quando una decisione tecnica, una criticità o una dipendenza tra componenti richiedono un coordinamento più immediato. Nel contesto osservato, questo equilibrio è importante perché le soluzioni sviluppate devono evolvere insieme ai bisogni dei clienti senza perdere coerenza tecnica tra le diverse parti del sistema.

1.5 Tecnologie e strumenti

Le tecnologie osservate durante lo *stage* coprono sia la comunicazione interna sia lo sviluppo applicativo. Questa sezione non intende descrivere in modo esaustivo ogni strumento, ma collocare le tecnologie principali all'interno del modo di lavorare dell'azienda.

1.5.1 Comunicazione e coordinamento

La comunicazione interna avviene principalmente tramite **Slack** e **Microsoft Teams**, utilizzati soprattutto per mantenere il contatto con i membri del *team* che, nella giornata, lavorano da remoto. Questi strumenti supportano il confronto quotidiano, gli aggiornamenti rapidi e le riunioni operative quando non tutti sono presenti nella stessa sede.
Per il coordinamento tecnico viene utilizzato **GitHub**, in particolare per la gestione del codice sorgente, il tracciamento delle modifiche e la revisione delle implementazioni tramite *pull request*$_G$, cioè proposte di modifica sottoposte a revisione prima dell'approvazione. Durante lo *stage*, questi strumenti hanno facilitato il confronto con il tutor aziendale e con gli altri membri del *team* tecnico.

1.5.2 *Backend*

Per lo sviluppo *backend* il *framework* di riferimento è **Django**, affiancato da **Django *REST*$_A$ Framework** per la realizzazione di API_A $REST_{AG}$, cioè interfacce che permettono a componenti *software* diversi di comunicare tramite richieste $HTTP_A$.
Il linguaggio principale utilizzato in questo ambito è **Python**, coerentemente con l'ecosistema Django e con le esigenze di sviluppo applicativo dell'azienda.
Il lato *backend* si appoggia inoltre a diverse tecnologie di supporto:

- **PostgreSQL**, utilizzato come *database* relazionale di riferimento;
- **Redis**, impiegato per *cache*$_G$, cioè memorizzazione temporanea di dati ad accesso rapido;

- **RabbitMQ**, utilizzato per la messaggistica tra servizi;
- **Celery**, adottato per l'esecuzione di attività in *background*$_G$, ossia non direttamente legate alla risposta immediata dell'interfaccia utente.

1.5.3 *Frontend*

Per lo sviluppo *frontend* il riferimento principale è **React**, affiancato da **TypeScript**. React permette di costruire interfacce modulari basate su componenti riutilizzabili, mentre TypeScript introduce una tipizzazione statica che contribuisce a rendere il codice più controllabile e manutenibile.
Nel contesto aziendale, il *frontend* ha il compito di rendere accessibili le funzionalità della piattaforma, presentare le informazioni in modo chiaro e comunicare all'utente lo stato dei processi applicativi. Questo aspetto è importante perché molti processi assicurativi sono complessi anche dal punto di vista informativo: l'interfaccia deve aiutare l'operatore a capire quale pratica sta gestendo, quali dati sono disponibili, quali passaggi sono stati completati e quali azioni sono ancora possibili.

1.5.4 Infrastruttura applicativa

Dal punto di vista infrastrutturale, **Docker** viene utilizzato per containerizzare i servizi e rendere più riproducibili gli ambienti di sviluppo ed esecuzione. **Nginx** svolge invece il ruolo di *web server* e *reverse proxy*$_G$, cioè riceve le richieste esterne e le instrada verso i servizi applicativi corretti.
Questi strumenti contribuiscono a mantenere più ordinata la gestione degli ambienti, soprattutto quando un'applicazione è composta da più servizi. La riproducibilità dell'ambiente è un aspetto rilevante per ridurre errori dovuti a configurazioni locali diverse e per rendere più semplice il passaggio tra codifica, verifica e rilascio.

1.6 Propensione verso l'innovazione

La propensione verso l'innovazione di RiskAPP emerge soprattutto dal modo in cui l'azienda affronta attività assicurative tradizionalmente frammentate o manuali. L'obiettivo non è introdurre tecnologia come elemento isolato, ma trasformare procedure operative complesse in flussi digitali più controllabili, nei quali le informazioni possano essere raccolte, elaborate e rese disponibili in modo coerente durante le diverse fasi del processo.

Centralità del dato
Un elemento centrale di questa impostazione è l'attenzione al dato assicurativo. Documenti, comunicazioni e informazioni operative vengono progressivamente ricondotti a strutture utilizzabili dal sistema, così da supportare attività come analisi, quotazione, monitoraggio e confronto tra pratiche. In questo senso, l'innovazione non riguarda soltanto l'automazione di singole operazioni, ma anche la possibilità di rendere osservabile un processo che, se gestito solo tramite scambi informali o strumenti separati, risulterebbe più difficile da controllare.

La centralità del dato ha anche un effetto sul lavoro tecnico. Una funzionalità non deve soltanto funzionare dal punto di vista dell'interfaccia, ma deve produrre, aggiornare e conservare informazioni coerenti con il resto del sistema. Questo richiede attenzione alla modellazione dei dati, alla comunicazione tra componenti e alla tracciabilità degli stati.

Automazione e controllo umano
Nel contesto osservato, l'uso di automazioni e di strumenti basati su intelligenza artificiale si inserisce in una logica di supporto alle decisioni. Il sistema può ridurre tempi operativi, evidenziare informazioni rilevanti e facilitare il passaggio tra le varie fasi, ma il dominio assicurativo richiede comunque attenzione alla verifica, alla responsabilità dell'operatore e alla comprensibilità degli esiti prodotti.
L'innovazione assume perciò un carattere pragmatico: migliora l'efficienza del lavoro senza eliminare la necessità di controllo umano sui passaggi più significativi. Questo equilibrio è importante perché consente di introdurre automazione in processi che restano sensibili dal punto di vista operativo, economico e informativo.

Impatto sullo *stage*
Questa impostazione ha influenzato anche il modo in cui ho interpretato l'ambiente di *stage*. Le attività tecniche non erano scollegate dal contesto aziendale, ma inserite in un prodotto che deve evolvere insieme alle esigenze del mercato assicurativo e dei suoi operatori. Comprendere questo contesto è stato quindi utile per leggere le scelte progettuali non solo come decisioni implementative, ma anche come parte di una strategia più ampia di digitalizzazione dei processi assicurativi.
Il capitolo successivo riprende questo quadro dal punto di vista del progetto di *stage*, descrivendo come le esigenze di integrazione, tracciabilità e aggiornamento dei processi abbiano orientato la proposta tecnica sviluppata.

Capitolo 2

Lo *stage*

2.1 Rapporto tra lo *stage* e l'azienda

Sono entrato in contatto con RiskAPP S.r.l. tramite l'evento Stage-IT, organizzato da Confindustria in collaborazione con l'Università degli Studi di Padova. L'incontro con l'azienda è avvenuto quindi all'interno di un contesto pensato per mettere in relazione studenti e organizzazioni interessate ad attivare percorsi formativi con contenuto tecnico concreto.

Inserimento nella strategia aziendale
Nel caso di RiskAPP, lo *stage* si inserisce in una modalità di innovazione graduale, vicina ai problemi applicativi osservati nel lavoro quotidiano. L'azienda non ha presentato il progetto come un'attività separata dal prodotto, ma come un'occasione per approfondire un'esigenza già emersa nel proprio contesto: migliorare la gestione di operazioni applicative lunghe e rendere più tempestiva la comunicazione degli aggiornamenti verso l'utente. In questa prospettiva, gli *stage* rappresentano per l'azienda anche uno strumento per esplorare soluzioni tecniche circoscritte, valutarne l'utilità rispetto al prodotto e ottenere un primo risultato concreto senza interrompere le attività ordinarie del *team*.

Contributo richiesto e sviluppi successivi
Questa impostazione è coerente con la propensione verso l'innovazione descritta nel capitolo precedente. RiskAPP tende infatti a introdurre nuove soluzioni quando esse possono rendere più controllabili i processi assicurativi e più fluida l'interazione tra sistemi, dati e utenti. Il contributo richiesto riguardava quindi l'analisi di un problema applicativo concreto e la realizzazione di una soluzione che potesse essere valutata non solo come risultato dello *stage*, ma anche come possibile base per sviluppi successivi. A partire da questo lavoro, l'azienda potrà considerare eventuali integrazioni nei propri applicativi, estendere l'approccio ad altri flussi operativi e disporre di indicazioni utili per orientare future evoluzioni del prodotto.
La Figura 2.1 sintetizza il modo in cui la proposta di *stage* si colloca nella strategia aziendale. Il diagramma è utile perché rende esplicito il legame tra un'esigenza applicativa già presente, il lavoro circoscritto richiesto durante lo *stage* e i possibili

sviluppi successivi: lo *stage*, quindi, non viene considerato come un'attività isolata, ma come una verifica concreta di una soluzione tecnica riutilizzabile.

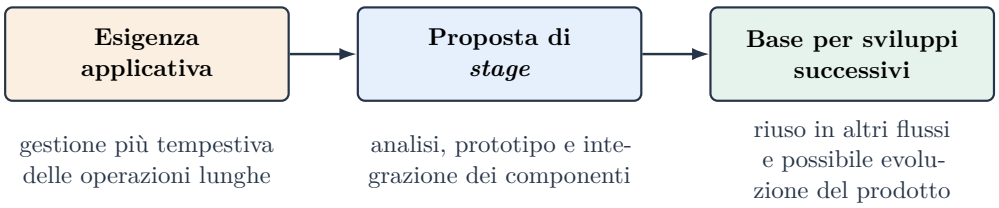


Figura 2.1: Collocazione dello *stage* tra esigenza applicativa, proposta progettuale e possibili sviluppi futuri.

2.2 Ambiente di lavoro

L'inserimento operativo nello *stage* è avvenuto in modo progressivo, a partire dalla presentazione dell'ambiente aziendale, degli strumenti utilizzati dal *team* tecnico e delle modalità di coordinamento adottate nelle attività quotidiane. Questa fase iniziale è stata importante per comprendere non solo il progetto assegnato, ma anche il modo in cui il lavoro individuale si sarebbe collegato alle attività già in corso all'interno dell'azienda.

Postazione e dotazione
Durante le giornate svolte in sede mi è stata assegnata una **postazione in *open space***, condivisa con gli altri membri del *team* tecnico. La vicinanza fisica con gli sviluppatori ha facilitato il confronto diretto, soprattutto nelle fasi in cui era necessario chiarire requisiti, scelte implementative o comportamenti attesi del sistema. La postazione era dotata di un ***monitor* secondario**, utile per lavorare con più finestre aperte e per condividere rapidamente una schermata durante momenti di confronto tecnico.
Come dotazione principale mi è stato fornito un **portatile MacBook Air**, utilizzato per l'intera durata del tirocinio. Il dispositivo è rimasto a mia disposizione anche nei giorni di lavoro da remoto e nei giorni festivi, consentendomi di mantenere continuità nell'ambiente di sviluppo e negli strumenti configurati per il progetto.

Accessi e coordinamento
Per poter partecipare alle attività del *team*, mi sono stati forniti gli accessi agli strumenti aziendali necessari. In particolare, sono stato inserito nei canali di comunicazione su **Slack** e **Microsoft Teams**, utilizzati per aggiornamenti rapidi, richieste di supporto e riunioni operative. Per la parte di sviluppo ho inoltre ricevuto l'accesso alle ***repository* private dell'azienda** su GitHub, così da poter consultare il codice necessario, lavorare sulle componenti assegnate e coordinare le modifiche con il resto del *team*.
L'insieme di questi strumenti ha reso l'inserimento abbastanza naturale: il lavoro poteva procedere in autonomia quando le attività erano chiare, ma restava sempre

possibile confrontarsi con il tutor aziendale o con gli altri sviluppatori nei momenti in cui emergevano dubbi tecnici o decisioni da condividere.

2.3 Problema affrontato

Il problema proposto nasce da un'esigenza già presente nelle applicazioni aziendali prima dell'inizio dello *stage*: gestire operazioni che non possono sempre essere completate nel tempo di una singola richiesta. La generazione di *report* di rischio assicurativo, a partire dall'elaborazione di una grande quantità di dati, ha rappresentato la fattispecie concreta su cui affrontare questo problema più generale, perché richiede elaborazioni non immediate e rende necessario informare l'utente sullo stato di avanzamento dell'attività avviata.
La gestione adottata originariamente dall'azienda per questi processi si basava sul *polling*$_G$, cioè sull'invio periodico di richieste dal *client* al *backend* per verificare se l'operazione fosse conclusa o se fossero disponibili aggiornamenti. Questa soluzione è semplice da comprendere, ma presenta limiti evidenti quando il numero di operazioni cresce:

- produce **traffico** anche quando non ci sono novità;
- **aumenta il carico** sui servizi coinvolti;
- può comunicare gli aggiornamenti con **ritardo** rispetto al momento in cui essi diventano disponibili.

La struttura iniziale, rappresentata in Figura 2.2, era quindi basata su un'interazione ripetuta tra *frontend* e *backend*: dopo l'avvio dell'operazione, il *client* continuava a interrogare il servizio applicativo per conoscere lo stato dell'elaborazione. Il diagramma è utile perché rende visibile il punto critico di questa impostazione: l'aggiornamento non nasce da una notifica del sistema, ma da richieste ripetute del *client*, anche quando l'elaborazione non ha ancora prodotto novità. Questo schema era funzionale in scenari semplici, ma rendeva l'aggiornamento dell'interfaccia dipendente dalla frequenza delle richieste periodiche.

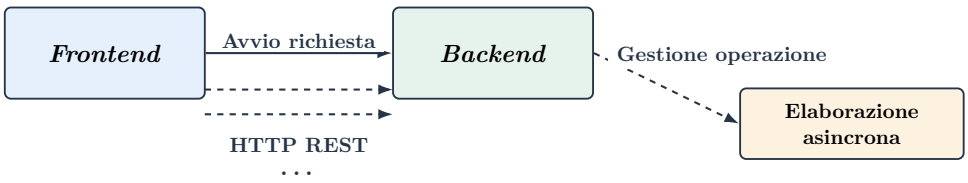


Figura 2.2: Schema della gestione iniziale degli aggiornamenti tramite richieste periodiche dal *frontend* al *backend*.

Da qui nasceva la necessità di valutare una modalità di aggiornamento più tempestiva verso il *frontend*, capace di comunicare i cambiamenti senza dipendere da interrogazioni periodiche del *client*.
La Figura 2.3 sintetizza l'idea architetturale proposta per superare questo limite. In particolare, lo schema distingue due forme di comunicazione:

- la comunicazione tradizionale tramite richieste $HTTP_A$ $REST_A$, usata dal *frontend* per dialogare con il *backend* e dal *backend* per interagire con il microservizio dedicato alla gestione degli aggiornamenti;
- la comunicazione in tempo reale tramite WebSocket$_G$, un canale persistente e bidirezionale che consente al sistema di inviare notifiche verso il *frontend* senza attendere una nuova interrogazione periodica da parte del *client*.

La figura è utile in questo capitolo non per descrivere tutti i dettagli implementativi, ma per chiarire il motivo della proposta: separare la gestione degli aggiornamenti asincroni e ridurre la dipendenza dal *polling*$_G$.

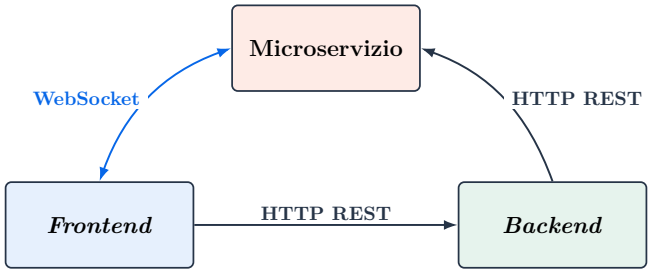


Figura 2.3: Schema dei componenti coinvolti nella proposta di *stage* e dei canali di comunicazione previsti.

La proposta si collega quindi a una strategia più ampia rispetto alla durata del singolo *stage*: rendere l'architettura applicativa più adatta a gestire operazioni asincrone, migliorare l'esperienza dell'utente attraverso aggiornamenti più immediati e fornire all'azienda componenti che possano essere riutilizzati o estesi in contesti futuri.

2.4 Obiettivi e vincoli dello *stage*

2.4.1 Obiettivi aziendali

RiskAPP ha definito gli obiettivi che intendeva raggiungere attraverso lo *stage*, classificandoli come obbligatori (OB), desiderabili (D) e facoltativi (F). La Tabella 2.1 riporta questa suddivisione, permettendo di distinguere il nucleo minimo della proposta dalle estensioni da valutare solo dopo il consolidamento delle parti principali.

Codice	Descrizione
OB1	Sviluppo di un microservizio *backend* basato su Django, Django $REST_A$ Framework e Django Channels.
OB2	Sviluppo di una libreria Python per l'integrazione del microservizio con *backend* Django.
OB3	Sviluppo di una libreria TypeScript per l'integrazione con *frontend* React.
OB4	Definizione e implementazione della comunicazione tra *backend* e microservizio.

Codice	Descrizione
OB5	Definizione e implementazione della comunicazione tra microservizio e *frontend* tramite WebSocket$_G$.
OB6	Implementazione di un sistema $CRUD_A$ per la gestione delle principali entità applicative e dei *report* PDF_A collegati al *frontend*.
OB7	Documentazione dell'architettura del sistema, delle modalità di comunicazione tra i componenti e delle istruzioni di integrazione del microservizio.
OB8	Produzione della documentazione tecnica delle librerie *client* e del microservizio sviluppato, distinguendo le indicazioni per l'utilizzo da quelle per la manutenzione futura.
OB9	Sviluppo dei *test* di unità del microservizio realizzato.
OB10	Sviluppo dei *test* di integrazione.
D1	Realizzazione di una piccola *dashboard* per il monitoraggio di eventi e *task* asincroni.
F1	Progettazione e implementazione di *test* e strumenti di *monitoring* lato *frontend*.

Tabella 2.1: Obiettivi aziendali dello *stage*, indicati con codice, descrizione e livello di priorità.

Gli obiettivi obbligatori coprivano lo sviluppo del sistema nel suo complesso, includendo le sue parti fondamentali: il microservizio, le librerie di integrazione, le funzionalità applicative e i canali di comunicazione tra *backend*, microservizio e *frontend*. La documentazione e i *test* erano inclusi nella proposta perché la soluzione non avrebbe avuto valore soltanto come prototipo isolato, ma anche come componente comprensibile e mantenibile da parte dell'azienda. Gli obiettivi desiderabili e facoltativi, invece, riguardavano funzionalità di osservazione e monitoraggio più periferiche rispetto alla finalità principale dello *stage*; per questo motivo erano subordinati alla disponibilità di tempo dopo il completamento del nucleo obbligatorio.

2.4.2 Vincoli tecnologici

Lo *stack* tecnologico di riferimento era definito dall'azienda e doveva rimanere coerente con l'ambiente descritto nel capitolo precedente. Per la parte *backend* la proposta prevedeva l'utilizzo di Django, Django $REST_A$ Framework e Django Channels, mentre per la parte *frontend* il riferimento era costituito da React e TypeScript. Queste tecnologie rappresentavano il perimetro principale entro cui sviluppare la soluzione, così da facilitarne l'integrazione con gli applicativi e le modalità di lavoro già presenti in RiskAPP.
All'interno di questo perimetro mi è stato lasciato un **margine di autonomia** significativo. Ho potuto valutare strumenti, librerie e soluzioni aggiuntive, confrontandole con i referenti aziendali e verificando che fossero compatibili con lo *stack* richiesto. Un esempio di questo equilibrio è stato l'avvio della libreria *frontend* a partire da un *template* aziendale, poi adattato alle esigenze specifiche del progetto.
Il vincolo principale, quindi, non era limitarsi a una sequenza rigida di tecnologie

predeterminate, ma **proporre scelte comprensibili, integrabili e manutenibili nel contesto di RiskAPP**.

2.4.3 Vincoli temporali

La durata complessiva dello *stage* prevista dall'azienda era di 308 ore. La pianificazione temporale, riportata nella Tabella 2.2, suddivide il lavoro in tre fasi principali: una fase iniziale di studio e analisi del contesto, una fase centrale di progettazione e implementazione, e una fase conclusiva dedicata alle librerie di integrazione, alla verifica e alla documentazione. La tabella è utile perché mostra non solo le attività previste, ma anche il peso assegnato a ciascuna fase all'interno della proposta.

Fase	**Durata in ore**		**Descrizione dell'attività**
Studio e analisi	88	64	Studio iniziale delle tecnologie: Django, Django $REST_A$ Framework, Django Channels, WebSocket$_G$, React, TypeScript e *template* delle librerie *frontend* aziendali.
		24	Comprensione del contesto applicativo in cui il sistema deve inserirsi: studio del problema di partenza e definizione delle prime ipotesi architetturali ai fini della soluzione.
Progettazione e implementazione	150	30	Progettazione delle funzionalità applicative e dell'integrazione tra componenti, tramite API_A $REST_A$ $_G$, WebSocket$_G$ e librerie *client*.
		120	Implementazione delle funzionalità applicative previste e integrazione tra componenti.
Librerie, verifica e documentazione	70	40	Realizzazione delle librerie *client* per *frontend* e *backend*.
		30	Attività di *test* e verifica e stesura della documentazione richiesta per utilizzatori e sviluppatori.
Totale			**308 ore**

Tabella 2.2: Pianificazione temporale dello *stage*, suddivisa per fasi, attività e durata prevista.

2.4.4 Metodo di lavoro

Lo *stage* è stato affrontato come un piccolo progetto autonomo, con obiettivi definiti, una pianificazione iniziale e momenti di confronto distribuiti lungo l'intero periodo di lavoro. Questa impostazione mi ha permesso di gestire una parte significativa delle attività in autonomia, mantenendo però un collegamento costante con il tutor aziendale e con i referenti tecnici.
Il metodo seguito non si è basato su una separazione rigida tra analisi, implementazione e verifica, ma su un avanzamento progressivo: le scelte progettuali venivano approfondite, discusse e poi validate man mano che il sistema prendeva forma. In

questo modo è stato possibile correggere eventuali incertezze prima che diventassero problemi più difficili da gestire nelle fasi successive.

2.4.4.1 Interazioni e revisioni

Il confronto con l'azienda è avvenuto attraverso diverse modalità, scelte in base al tipo di attività in corso. Alcune interazioni erano programmate in modo fisso, soprattutto nei momenti in cui era necessario impostare una nuova fase di lavoro, verificare l'avanzamento rispetto agli obiettivi oppure chiarire il risultato atteso prima di procedere con la codifica.
A queste si sono affiancate revisioni programmate, dedicate al controllo delle scelte progettuali e delle implementazioni realizzate. Tali momenti sono stati utili per validare le decisioni principali, individuare eventuali modifiche da apportare e mantenere la soluzione coerente con le esigenze aziendali. Oltre agli incontri previsti, il confronto poteva avvenire anche al bisogno, quando emergevano dubbi tecnici, necessità di chiarimento o decisioni da condividere rapidamente con il tutor o con altri membri del *team*.

2.4.4.2 Pianificazione

La pianificazione del lavoro ha seguito la suddivisione generale già descritta nei vincoli temporali, ma è stata applicata in modo operativo alle attività quotidiane. In particolare, il lavoro è stato organizzato in tre momenti principali:

- **Studio e analisi**: la prima parte dello *stage* è stata dedicata allo studio delle tecnologie e alla comprensione del contesto applicativo in cui la soluzione avrebbe dovuto inserirsi. Questa fase ha permesso di definire il problema con maggiore precisione, chiarire i vincoli di integrazione con i sistemi esistenti e individuare le prime ipotesi architetturali.
- **Progettazione e implementazione**: la parte centrale è stata dedicata non solo alla definizione dell'architettura della soluzione, ma anche alla realizzazione concreta delle funzionalità principali. In particolare, ho lavorato sulle componenti *backend*, microservizio e *frontend*, occupandomi sia delle funzionalità applicative sia dell'integrazione tra le diverse parti del sistema. In questa fase la pianificazione è servita a mantenere allineati i componenti sviluppati e i rispettivi canali di comunicazione, evitando che le singole attività procedessero in modo isolato.
- **Librerie, verifica e documentazione**: la fase conclusiva ha riguardato il completamento delle librerie di integrazione, la verifica del funzionamento complessivo e la preparazione della documentazione tecnica. Le attività finali sono state quindi orientate a consolidare quanto implementato e a rendere la soluzione più comprensibile e riutilizzabile nel contesto aziendale.

La pianificazione iniziale ha quindi fornito una traccia di riferimento, mentre il confronto periodico con l'azienda ha permesso di adattare le attività al grado di maturazione della soluzione.

2.4.4.3 Strumenti e tracciamento

Gli strumenti utilizzati durante lo *stage* hanno supportato sia il coordinamento con l'azienda sia l'organizzazione tecnica del lavoro. In particolare, si possono distinguere tre ambiti principali:

- **Comunicazione**: per il confronto con il tutor e con il *team* sono stati utilizzati principalmente Slack e Microsoft Teams, già introdotti come strumenti aziendali nella sezione dedicata all'ambiente di lavoro. Questi canali hanno permesso di mantenere un contatto continuo, sia per aggiornamenti rapidi sia per richieste di supporto più specifiche.
- **Gestione del codice**: il codice è stato organizzato tramite *repository* Git private su GitHub, che hanno consentito di tracciare l'evoluzione delle modifiche e mantenere ordinato il lavoro sulle diverse componenti. Questa impostazione ha facilitato la separazione tra microservizio, librerie e parti di integrazione, rendendo più chiaro lo sviluppo progressivo della soluzione.
- **Tracciamento**: accanto al codice sono stati prodotti documenti di supporto, diagrammi e appunti tecnici utili a motivare alcune scelte progettuali e a rendere più comprensibile il funzionamento del sistema. Il tracciamento dei requisiti e delle attività è stato mantenuto anche attraverso *checklist* tecniche e verifiche progressive, così da collegare le implementazioni realizzate agli obiettivi concordati e controllare l'avanzamento delle funzionalità previste.

2.5 Scelta della proposta e obiettivi personali

Motivazione della scelta
Fin dalla pubblicazione della lista delle aziende partecipanti a Stage-IT, la proposta di RiskAPP ha attirato in modo particolare il mio interesse. Tra i progetti presentati, quello dell'azienda mi è apparso stimolante sia per le tecnologie coinvolte sia per la possibilità di lavorare su un problema concreto, collegato a un prodotto esistente e non a un esercizio separato dal contesto professionale.
Durante l'evento ho sostenuto diversi colloqui con varie aziende, cercando di valutare non solo il contenuto tecnico delle singole proposte, ma anche il tipo di esperienza formativa che ciascuna avrebbe potuto offrirmi. Questo confronto mi ha aiutato a chiarire meglio le mie priorità: più che la vicinanza geografica della sede o la comodità logistica, per me era importante scegliere uno *stage* capace di farmi acquisire competenze tecniche, metodo di lavoro e maggiore consapevolezza del funzionamento di un ambiente aziendale.
RiskAPP è rimasta fin dall'inizio la scelta che ritenevo più adatta al mio percorso. Questa impressione è stata confermata nel successivo incontro presso la sede aziendale, durante il quale ho potuto approfondire il progetto, comprendere meglio l'ambiente in cui mi sarei inserito e valutare il rapporto tra autonomia personale e confronto con i referenti aziendali. La scelta finale è quindi maturata dalla combinazione di più fattori: interesse per l'azienda e per la proposta tecnica, coerenza con il

mio percorso formativo e convinzione che l'esperienza potesse offrirmi un contributo significativo in vista del lavoro futuro.

Obiettivi personali

Accanto agli obiettivi iniziali definiti dall'azienda, ho individuato alcuni obiettivi personali (OP) legati alla crescita tecnica e professionale, che prima di iniziare lo *stage* mi ero prefissato di realizzare nel loro insieme. La Tabella 2.3 li riporta in forma misurabile, descrivendo per ciascuno il risultato concreto attraverso cui sarebbe stato possibile valutarne il raggiungimento nella retrospettiva finale. La tabella è utile perché distingue gli aspetti formativi che intendevo sviluppare dalle funzionalità richieste dall'azienda: a differenza degli obiettivi aziendali, gli obiettivi personali non sono organizzati secondo una scala di priorità, ma rappresentano un insieme di risultati formativi che intendevo perseguire durante l'intera esperienza.

Codice	**Descrizione**
OP1	Rafforzare l'autonomia nell'analisi delle esigenze iniziali e nella scelta delle possibili soluzioni tecniche, motivando le principali decisioni progettuali durante i confronti con il tutor aziendale e con i referenti tecnici.
OP2	Approfondire la progettazione e l'implementazione di soluzioni asincrone distribuite attraverso la realizzazione di un flusso funzionante basato su Django Channels, Celery e comunicazione tramite WebSocket$_G$.
OP3	Migliorare la capacità di progettare componenti riutilizzabili attraverso la produzione di librerie Python e TypeScript integrabili con altri *backend* Django e *frontend* React.
OP4	Consolidare il metodo di verifica e documentazione del lavoro svolto mediante la predisposizione di *test*, documentazione tecnica e materiali di supporto utili alla manutenzione e al riuso della soluzione.

Tabella 2.3: Obiettivi personali dello *stage*, indicati con codice e descrizione misurabile.

Capitolo 3

Il progetto

3.1 Analisi e progettazione

Questa sezione descrive il passaggio **dalla comprensione del problema applicativo alla definizione dell'architettura della soluzione**. L'attenzione è posta soprattutto sulla separazione delle responsabilità tra *frontend*, *backend* applicativo e microservizio, perché proprio questa distinzione ha guidato le scelte successive di implementazione e integrazione.

3.1.1 Analisi del problema applicativo

La prima parte dello *stage* è stata dedicata alla comprensione del problema e del contesto applicativo in cui la soluzione avrebbe dovuto inserirsi. L'attività non è stata condotta in modo astratto, bensì a partire da una necessità concreta: gestire operazioni asincrone di lunga durata, come la generazione di *report* PDF_A, evitando che l'utente dovesse attendere passivamente o che il sistema dipendesse da continue interrogazioni periodiche.

Difficoltà affrontate
In questa fase ho analizzato i limiti di una gestione basata su $polling_G$, cioè sull'invio ripetuto di richieste dal *client* al *backend* per verificare lo stato di un'elaborazione. Questo approccio può risultare **semplice da implementare**, ma **introduce traffico** anche quando non sono disponibili aggiornamenti e può **ritardare la comunicazione** dello stato effettivo all'utente. La difficoltà principale non è stata tanto la comprensione del singolo caso d'uso, quanto la necessità di **ricondurre esigenze applicative diverse a un modello comune**. La generazione dei *report* costituiva il caso più immediato da cui partire, ma la soluzione doveva essere coerente anche con l'idea di gestire in modo riutilizzabile altri processi asincroni.

Scelte adottate
Per evitare di procedere sulla base di interpretazioni implicite, ho prodotto alcune bozze di documentazione personale, usate come strumento di lavoro durante l'analisi. Tali bozze descrivevano il problema, gli attori coinvolti, il flusso previsto

degli aggiornamenti e i punti ancora da chiarire. Il loro scopo non era costituire documentazione finale di progetto, ma rendere esplicita la mia comprensione del dominio e permettere ai referenti aziendali di verificarla progressivamente.

Risultati conseguiti
Questo confronto ha avuto un ruolo pratico nella prosecuzione del lavoro: mi ha permesso di distinguere gli aspetti già sufficientemente chiari, come l'esigenza di superare il *polling*$_G$, dagli aspetti da definire in progettazione, come la distribuzione delle responsabilità tra *backend* applicativo, microservizio e *frontend*.

3.1.2 Studio delle tecnologie

Difficoltà affrontate
Lo studio delle tecnologie è stato orientato dalle esigenze emerse nell'analisi iniziale e dal perimetro tecnico già presente in RiskAPP. La difficoltà principale non era soltanto imparare la sintassi o le funzionalità dei singoli strumenti, ma comprenderne l'uso in un contesto applicativo reale, nel quale le scelte tecniche dovevano rispettare vincoli già presenti, convenzioni di progetto e modalità di lavoro aziendali. Alcune tecnologie, come Django Channels, Celery e il protocollo di comunicazione WebSocket$_G$, non facevano parte delle competenze che possedevo già in modo pratico; altre erano note solo a livello generale e richiedevano quindi un approfondimento più operativo.

Scelte adottate
Per evitare uno studio puramente teorico, ho seguito un percorso di tre passaggi:

- ho consultato la documentazione delle tecnologie principali che delimitavano il perimetro della soluzione, concentrandomi sulle parti effettivamente utili al progetto;
- ho confrontato quanto studiato con le convenzioni già presenti in RiskAPP, così da non introdurre soluzioni tecnicamente corrette ma incoerenti con il modo di lavorare dell'azienda;
- ho realizzato piccoli prototipi e prove isolate, utili a verificare in pratica il comportamento delle tecnologie prima di inserirle nel flusso applicativo completo.

Sul lato *server* mi sono concentrato su Django, Django *REST*$_A$ Framework e Django Channels, studiando il ruolo delle *API*$_A$ *REST*$_{AG}$ nella comunicazione tra servizi e quello dei WebSocket$_G$ nella trasmissione di aggiornamenti in tempo reale. Ho inoltre approfondito Celery, RabbitMQ, Redis e PostgreSQL, poiché questi strumenti erano rilevanti per l'esecuzione delle elaborazioni asincrone, l'instradamento dei messaggi e la persistenza degli stati.
Ho esteso l'analisi anche al *frontend*, perché il valore del progetto non dipendeva soltanto dalla corretta esecuzione delle operazioni lato *server*, ma anche dalla possibilità di mostrare all'utente aggiornamenti chiari e tempestivi. Per questo motivo ho

studiato come integrare la comunicazione in tempo reale in un'applicazione React e TypeScript, valutando il modo in cui notifiche, stati di avanzamento e messaggi provenienti dal microservizio potessero essere rappresentati in modo coerente nell'interfaccia.

Risultati conseguiti
Il risultato di questa fase non è stato una scelta tecnologica completamente libera, ma una comprensione più precisa di come usare strumenti già coerenti con il perimetro aziendale per costruire una soluzione integrabile e manutenibile.

3.1.3 Architettura della soluzione

La progettazione ha trasformato l'analisi iniziale in una struttura operativa del sistema. In questa fase l'obiettivo non era soltanto rappresentare graficamente i componenti, ma definire un'architettura capace di guidare la successiva codifica e di rendere espliciti i confini tra responsabilità applicative, gestione asincrona e comunicazione in tempo reale.

Difficoltà affrontate
Poiché il problema applicativo e il contesto aziendale erano già stati inquadrati nelle fasi precedenti del progetto, in questa parte del lavoro l'attenzione si è spostata soprattutto sul **confine tra i componenti**. La generazione dei *report* era un caso d'uso utile per rendere concreto il problema, ma non esauriva l'obiettivo: la soluzione doveva introdurre un modello più generale per registrare, aggiornare e notificare lo stato di operazioni asincrone. La difficoltà progettuale principale consisteva quindi nel decidere **dove collocare le responsabilità**, evitando due estremi:

- concentrare troppa logica nel *backend* applicativo, rendendo il microservizio un elemento marginale e poco riutilizzabile;
- spostare nel microservizio dettagli specifici del dominio dei *report*, aumentando l'accoppiamento e riducendo la generalità della soluzione.

Il punto critico non era quindi decidere se eseguire un'elaborazione in modo asincrono, ma stabilire **quale componente dovesse possedere lo stato del *task***, quale invece dovesse conoscere le regole del dominio applicativo e quale infine dovesse comunicare gli aggiornamenti all'interfaccia.

Scelte adottate
Il lavoro si è concentrato sulla **separazione delle responsabilità** tra i tre componenti principali: *backend*, *frontend* e microservizio.
La presenza di un terzo componente, il microservizio, rientrava nel perimetro richiesto dall'azienda. La scelta progettuale più rilevante ha riguardato però il **ruolo da assegnargli**: non era sufficiente introdurre un servizio aggiuntivo, perché un confine sbagliato avrebbe potuto spostare complessità senza ridurla. Per questo motivo ho confrontato con i referenti aziendali diverse possibilità di ripartizione

delle responsabilità, sintetizzate nella Tabella 3.1. L'inserto è utile perché esplicita il ragionamento progettuale alla base dell'architettura, distinguendo la scelta adottata dalle alternative che avrebbero prodotto un maggiore accoppiamento tra i componenti.

Ipotesi considerata	Effetto sulla separazione delle responsabilità	Valutazione progettuale
Microservizio come semplice tramite degli eventi	Il *backend* applicativo avrebbe mantenuto quasi tutta la gestione dello stato, mentre il microservizio avrebbe inoltrato soltanto notifiche verso il *frontend*.	Soluzione poco robusta: avrebbe ridotto il microservizio a un canale tecnico, senza introdurre una rappresentazione condivisa del ciclo di vita dei *task*.
Microservizio come componente centrale anche per il dominio dei *report*	Il microservizio avrebbe conosciuto dettagli specifici del caso d'uso applicativo, come struttura dei *report*, regole di annullamento e logica collegata ai dati gestiti dal *backend*.	Soluzione scartata perché avrebbe aumentato l'accoppiamento, rendendo il componente meno riutilizzabile in altri contesti.
Microservizio come gestore dello stato asincrono e degli eventi	Il *backend* mantiene la logica di dominio e avvia le elaborazioni; il microservizio conserva lo stato dei *task* e notifica gli eventi; il *frontend* riceve aggiornamenti senza interrogazioni periodiche.	**Soluzione adottata**: consente di isolare la gestione asincrona, mantenere chiari i confini applicativi e predisporre il componente al riuso.

Tabella 3.1: Valutazione delle principali alternative nella ripartizione delle responsabilità tra *backend* applicativo e microservizio.

Risultati conseguiti

Il risultato della progettazione è un'architettura in cui:

- il *frontend* è responsabile dell'interazione con l'utente e della visualizzazione degli aggiornamenti;
- il *backend* applicativo è responsabile delle funzionalità di dominio e dell'avvio delle elaborazioni asincrone;
- il microservizio è responsabile della registrazione dei *task*, della gestione del loro stato e dell'invio degli eventi in tempo reale.

In questo modo il microservizio non assume il controllo della generazione dei *report*, ma fornisce un'infrastruttura applicativa riutilizzabile per rappresentare l'avanzamento dei processi asincroni.
La Figura 3.1 riassume l'architettura progettata e permette di leggere il sistema secondo due livelli: nella parte superiore sono rappresentati i tre componenti applicativi principali, mentre nella parte inferiore sono collocati i servizi di supporto necessari all'esecuzione asincrona e alla persistenza dei dati. Il diagramma evidenzia

inoltre la distinzione tra i canali di comunicazione: le API_A $REST_{AG}$ sono usate per la comunicazione tradizionale tra *frontend*, *backend* e microservizio, mentre il canale WebSocket_G collega direttamente microservizio e *frontend* per la trasmissione degli aggiornamenti in tempo reale. La presenza di RabbitMQ e del *worker* Celery permette invece di leggere il percorso dell'elaborazione asincrona in tre passaggi:

1. il *backend* applicativo avvia il lavoro e lo inserisce nella coda;
2. il *worker* Celery esegue l'elaborazione e produce il PDF_A;
3. il microservizio riceve gli aggiornamenti e mantiene lo stato del *task* allineato con gli eventi notificati al *frontend*.

In questo modo la figura chiarisce il motivo della separazione architetturale adottata, senza confondere la logica applicativa con la gestione degli eventi.

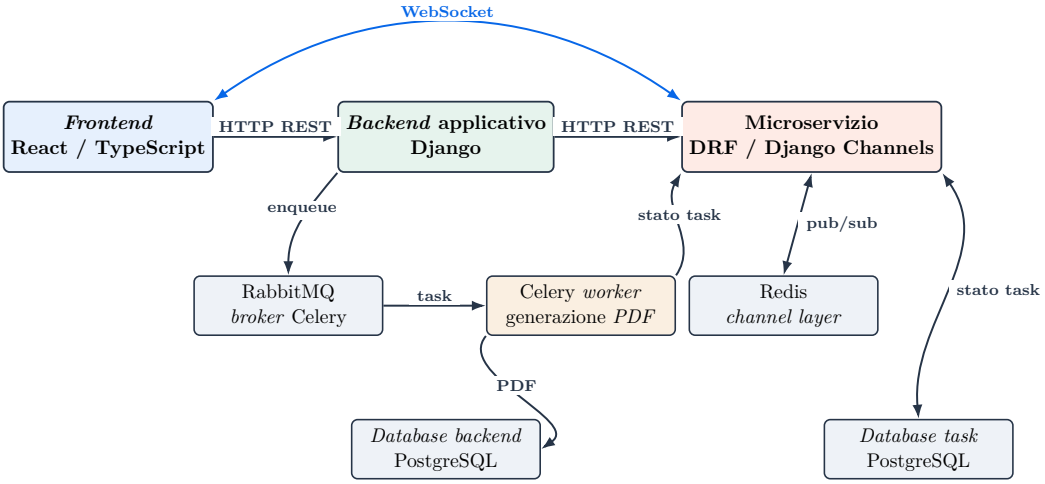
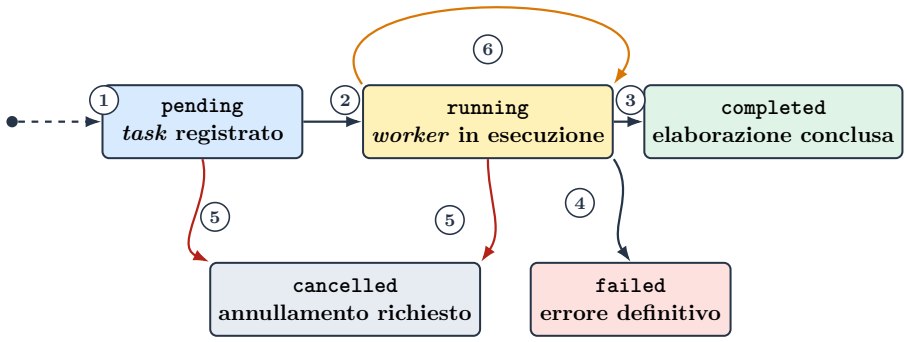


Figura 3.1: Architettura progettata per la gestione delle operazioni asincrone.

3.1.4 Ciclo di vita del *task*

A completamento della vista architetturale, ho definito il ciclo di vita del singolo *task* asincrono. La Figura 3.2 mostra gli stati principali registrati dal microservizio e gli eventi prodotti verso il *frontend*. Questa rappresentazione è utile perché chiarisce come il sistema distingue le diverse situazioni possibili: creazione del *task*, esecuzione, completamento, fallimento, nuovo tentativo e annullamento.



1 `create_async_task` / `task.updated` **2** `before_start` / `task.updated`
3 `on_success` / `task.ready` **4** `on_failure` / `task.failed`
5 annullamento / `task.cancelled` **6** *retry* / `task.retry`, stato ancora `running`

Figura 3.2: Ciclo di vita di un *task* asincrono e relativi eventi notificati al *frontend*.

Nel flusso progettato, il *task* viene creato dal *backend* applicativo presso il microservizio e parte dallo stato `pending`. Quando il *worker* Celery inizia l'elaborazione, lo stato passa a `running`; da qui il *task* può completarsi con `task.ready`, fallire definitivamente con `task.failed`, rimanere in esecuzione dopo un nuovo tentativo tramite `task.retry` oppure essere annullato con `task.cancelled`. Ogni passaggio rilevante produce un evento che il microservizio inoltra al *frontend*, così da aggiornare l'interfaccia senza ricorrere a interrogazioni periodiche.

3.2 Codifica e integrazione dei componenti

La fase di codifica ha trasformato l'architettura descritta nella sezione precedente in componenti funzionanti e collegati tra loro. Il lavoro non si è limitato all'implementazione isolata dei singoli moduli, ma ha richiesto di mantenere coerente il flusso complessivo. Per questo motivo le attività sono state organizzate seguendo i confini principali della soluzione: microservizio, *backend* applicativo, *frontend* e librerie di integrazione.

3.2.1 Microservizio per processi asincroni

La realizzazione del microservizio ha riguardato la gestione dei *task* asincroni e degli eventi associati.

Difficoltà affrontate
La difficoltà principale è stata trasformare il ruolo definito in progettazione in un componente effettivamente utilizzabile dagli altri moduli. Il microservizio doveva mantenere una rappresentazione affidabile dello stato dei *task*, ma senza assumere responsabilità proprie del dominio applicativo. Inoltre, gli aggiornamenti dovevano essere comunicati in tempo reale solo ai *client* interessati, evitando sia notifiche non autorizzate sia dipendenze eccessive dal *backend* applicativo.

Scelte adottate
Il lavoro si può riassumere in tre aspetti principali:

- ho modellato le informazioni necessarie per identificare un *task*, associarlo all'utente che lo aveva generato e mantenerne lo stato nel tempo. In questo modo il microservizio non si limita a ricevere aggiornamenti dal *backend*, ma diventa il punto in cui viene mantenuta la rappresentazione condivisa dello stato dell'elaborazione;
- ho implementato le API_A $REST_{AG}$ interne per creare, leggere, aggiornare ed eliminare i *task* ($CRUD_A$). Le operazioni esposte sono state pensate per essere usate soltanto da *backend* applicativi autorizzati, tramite autenticazione basata su JWS_A, in particolare dal *backend* che avvia l'elaborazione e dal *worker* che ne aggiorna l'esito. In particolare:
  - il JWS_A viene firmato con crittografia asimmetrica, mantenendo la chiave privata nel *backend* autorizzato;
  - la chiave pubblica viene fornita manualmente al microservizio, che la usa per verificare la firma e accettare solo richieste provenienti da componenti riconosciuti.
- ho gestito il canale in tempo reale tramite WebSocket$_G$, inviando gli aggiornamenti solo all'utente corretto tramite gruppi associati all'utente autenticato. Ogni modifica rilevante dello stato può così produrre un evento da inoltrare ai *client* interessati, evitando che il *frontend* debba interrogare periodicamente il sistema. Anche l'autenticazione del *frontend* sul canale WebSocket$_G$ segue lo stesso criterio di fiducia adottato tra *backend* e microservizio:
  - il *backend* firma un JWS_A con la propria chiave privata e lo restituisce al *frontend*;
  - il *frontend* invia il JWS_A al microservizio quando richiede l'iscrizione al proprio canale WebSocket$_G$;
  - il microservizio verifica la firma tramite la medesima chiave pubblica e, se la verifica ha esito positivo, accetta la richiesta e stabilisce la connessione.

Il modello dati$_G$ del microservizio, cioè la struttura che definisce quali informazioni vengono memorizzate per ogni *task*, è stato progettato quindi per contenere solo le informazioni necessarie alla gestione asincrona. Per questo motivo non include riferimenti diretti al contenuto del *report*, ma registra l'identificativo del *task*, lo stato corrente, l'utente che lo ha avviato, i tempi dell'elaborazione e le informazioni utili a rappresentare errori o tentativi falliti. Il Codice 3.1 riporta un estratto del modello, limitato ai campi principali: l'inserto è utile perché mostra concretamente quali dati sono stati resi persistenti e chiarisce il confine tra stato tecnico dell'elaborazione e contenuto applicativo del *report*.

```
class AsyncTask(models.Model):
    id_task = models.UUIDField(primary_key=True, default=uuid.
        uuid4)
```

```
    status = models.CharField(max_length=20, choices=
       STATUS_CHOICES)
    user_creation = models.PositiveBigIntegerField()
    date_creation = models.DateTimeField()
    duration = models.PositiveIntegerField(default=0)
    attempts_failed = models.PositiveIntegerField(default=0)
    error_code = models.CharField(max_length=80, blank=True)
    error_message = models.TextField(blank=True)
```

Codice 3.1: Estratto del modello dati$_G$ usato dal microservizio per rappresentare un *task* asincrono.

Gli stati ammessi dal modello descrivono le principali fasi del ciclo di vita del *task*:

- `pending`, per i *task* registrati ma non ancora avviati;
- `running`, per i *task* in esecuzione;
- `completed`, `failed` e `cancelled`, per rappresentare gli esiti conclusivi.

A questi si aggiungono i campi automatici di creazione e aggiornamento del *record*. Questa struttura consente al microservizio di descrivere il ciclo di vita del *task* senza conoscere la logica specifica dell'elaborazione eseguita dal *backend*.

Risultati conseguiti
Queste scelte hanno permesso di **separare la persistenza dello stato dalla sua comunicazione verso l'interfaccia**, mantenendo il microservizio responsabile del ciclo di vita dei *task*. Il componente ottenuto costituisce quindi il punto comune attraverso cui gli **altri moduli autorizzati** possono registrare elaborazioni asincrone, aggiornarne l'esito e notificare il *frontend* senza introdurre logica specifica, come quella dei *report*.

3.2.2 *Backend* applicativo

Sul *backend* applicativo ho sviluppato le funzionalità necessarie al *frontend* per l'autenticazione, la gestione delle entità applicative e la gestione dei *report* PDF_A. Questa parte del lavoro ha rappresentato il punto di raccordo tra il dominio applicativo e la nuova infrastruttura asincrona: il *backend*, infatti, non doveva limitarsi a inviare richieste al microservizio, ma doveva mantenere la coerenza tra utenti, clienti assicurati, *report* richiesti e stato tecnico dell'elaborazione.

Difficoltà affrontate
La difficoltà principale è stata integrare la nuova gestione asincrona senza confondere le responsabilità del *backend* con quelle del microservizio. Il *backend* doveva continuare a essere il punto di riferimento per utenti, dati applicativi e regole di dominio, ma doveva anche coordinare la generazione dei *report* con il ciclo di vita del *task* registrato nel microservizio.
Le criticità principali hanno riguardato tre aspetti:

- **coerenza tra dati applicativi e stato asincrono**: la richiesta di generazione doveva produrre un *report* tracciabile nel dominio applicativo e, allo stesso tempo, un *task* riconosciuto dal microservizio;
- **gestione delle operazioni ammesse**: non tutte le azioni hanno senso in ogni stato del processo. Ad esempio, un *report* può essere scaricato solo quando il file è disponibile, mentre l'annullamento è significativo solo quando l'elaborazione non è ancora conclusa;
- **isolamento tra utenti**: le operazioni esposte al *frontend* dovevano rispettare i permessi dell'utente autenticato, evitando che dati o *report* appartenenti ad altri utenti fossero consultabili o modificabili.

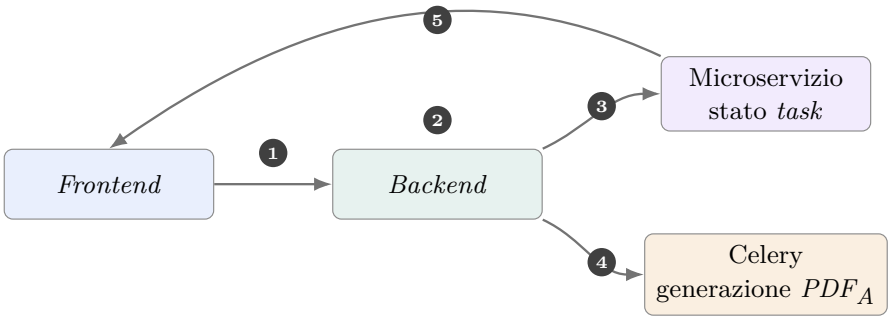
In questa parte, quindi, il problema non era soltanto avviare un'elaborazione Celery, ma inserirla in un flusso applicativo controllato, in cui ogni passaggio avesse una conseguenza coerente sia nel *backend* sia nel microservizio.

Scelte adottate

Questa parte del sistema è stata implementata mantenendo **nel *backend* le responsabilità legate al dominio applicativo** e delegando al microservizio la rappresentazione tecnica dello stato asincrono. In concreto, le scelte adottate possono essere ricondotte a tre funzioni principali:

- **mantenere le responsabilità legate al dominio applicativo**: il *backend* continua a gestire utenti, clienti assicurati, dati necessari alla produzione dei *report*, regole di accesso e controllo delle operazioni consentite. In questo modo il microservizio non deve conoscere la struttura dei dati applicativi né le regole che determinano quali *report* un utente può generare o consultare;
- **collegare la generazione di un *report* al microservizio asincrono**: quando l'utente richiede una nuova generazione, il *backend* crea il riferimento applicativo al *report*, registra un *task* presso il microservizio, avvia l'elaborazione tramite Celery e conserva il collegamento tra il *report* e il *task* corrispondente. Il *worker*, al termine dell'elaborazione, aggiorna l'esito e rende disponibile il file PDF_A;
- **esporre le operazioni collegate al ciclo di vita del *report***: le API_A applicative permettono al *frontend* di consultare lo stato, richiedere una nuova generazione, annullare un'elaborazione quando previsto, scaricare il *file* prodotto ed eliminare il *report* nei casi consentiti.

Il flusso applicativo risultante è stato organizzato in modo sequenziale, così da rendere esplicito il ruolo del *backend* in ogni passaggio. La Figura 3.3 sintetizza questa sequenza: l'inserto è utile perché mostra come il *backend* faccia da raccordo tra richiesta utente, dati applicativi, microservizio asincrono ed elaborazione Celery.



1 richiesta autenticata di generazione
2 verifica permessi e creazione delle informazioni applicative (*report* PDF_A)
3 registrazione del *task* asincrono presso il microservizio
4 elaborazione demandata a Celery e produzione del PDF_A
5 notifica dello stato dal microservizio al *frontend*

Figura 3.3: Flusso coordinato dal *backend* per generare un *report* PDF_A e rendere visibile lo stato dell'elaborazione al *frontend*.

Questa organizzazione consente di distinguere chiaramente il significato applicativo del *report* dal significato tecnico del *task*: il primo appartiene al dominio del *backend*, il secondo descrive l'avanzamento dell'elaborazione asincrona.

Risultati conseguiti
Il risultato ottenuto è un *backend* che mantiene il **collegamento tra le funzionalità applicative ordinarie e la nuova gestione asincrona**, evitando che il microservizio debba conoscere dettagli specifici del dominio dei *report*. La generazione del PDF_A rimane quindi una funzionalità applicativa, mentre lo stato dell'elaborazione viene rappresentato e notificato tramite il componente dedicato.
Dal punto di vista del progetto, questa scelta ha prodotto tre effetti principali:

- il flusso di generazione è diventato più controllabile, perché ogni *report* è collegato a uno stato tecnico esplicito e consultabile;
- le operazioni esposte al *frontend* risultano più coerenti con lo stato effettivo dell'elaborazione: l'utente non interagisce più con un processo opaco, ma con un insieme di azioni abilitate o meno in base alla situazione del *report*;
- il microservizio resta riutilizzabile anche per altri processi asincroni, perché il *backend* si fa carico della traduzione tra dominio applicativo e infrastruttura comune di gestione dei *task*.

3.2.3 *Frontend* e interazione utente

Sul lato *frontend* ho realizzato le viste e le interazioni necessarie per permettere all'utente di gestire le entità applicative, avviare la generazione dei *report*, consultarne lo stato, annullare le elaborazioni quando previsto e scaricare il PDF_A al termine

del processo.

Difficoltà affrontate
La difficoltà non riguardava soltanto la costruzione delle schermate, ma la traduzione di un processo asincrono in un'esperienza comprensibile per l'utente. Stati intermedi, errori temporanei, annullamenti e completamento del *task* dovevano essere rappresentati in modo chiaro, senza costringere l'utente a ricaricare manualmente la pagina o a interpretare dettagli tecnici interni al sistema.

Scelte adottate
L'obiettivo non era solo rendere disponibile la funzionalità tecnica, ma presentare l'evoluzione dell'elaborazione in modo comprensibile per l'utente. Per questo motivo il flusso è stato organizzato intorno a due viste principali.

Gestione clienti assicurati
La gestione dei *report* è stata organizzata a partire dalla schermata dei clienti assicurati. Da questa vista l'utente può:

- creare un nuovo cliente;
- visualizzare l'elenco dei clienti disponibili;
- visualizzare i dettagli propri dell'utente;
- accedere alla pagina di gestione dei *report* di un certo cliente tramite l'azione `Gestisci report`.

I dati mostrati nelle seguenti schermate sono dimostrativi e non corrispondono a clienti reali.
La Figura 3.4 mostra questa vista iniziale: l'inserto è utile perché evidenzia il punto di ingresso del flusso, prima che l'utente passi alla gestione dei singoli *report*.

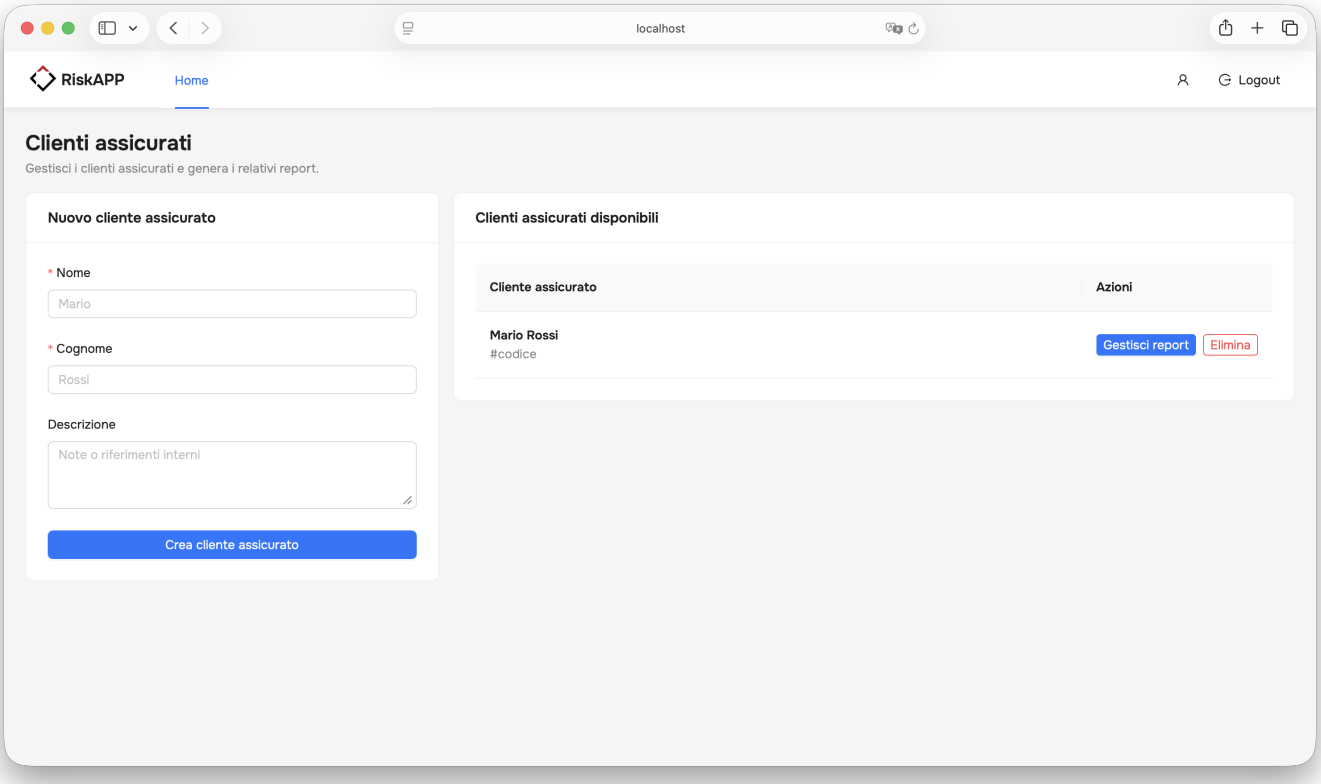


Figura 3.4: Schermata dei clienti assicurati. Fonte: schermata acquisita dalla piattaforma sviluppata durante lo *stage*.

Gestione *report* del cliente assicurato

All'interno del dettaglio, quindi dopo aver selezionato il bottone `Gestisci report` di un certo cliente, sono mostrati i dati del cliente in questione e una tabella dei *report* PDF_A associati. Per ogni *report* vengono visualizzati:

- l'identificativo ($UUID_A$);
- il titolo;
- lo stato corrente;
- la data dell'ultimo aggiornamento.

Le azioni disponibili permettono di avviare una nuova generazione, annullare un'elaborazione ancora in attesa o in esecuzione, aprire il dettaglio del *report*, scaricare il PDF_A quando il *file* è disponibile ed eliminare il *report* nei casi consentiti. Gli aggiornamenti ricevuti tramite WebSocket$_G$ invalidano i dati locali e mostrano notifiche all'utente, ad esempio quando:

- il *task* passa in esecuzione;
- il PDF_A è pronto;
- avviene un errore durante la generazione;

- avviene un nuovo tentativo a seguito di un errore;
- il *task* viene annullato.

Questa parte ha quindi **collegato la gestione asincrona alla percezione dell'utente**, rendendo visibili stati che altrimenti sarebbero rimasti interni al sistema. La Figura 3.5 mostra un esempio della schermata di dettaglio di un *report*: oltre allo stato finale completato, l'interfaccia conserva il messaggio `TASK_RETRY`, relativo a un errore temporaneo simulato al primo tentativo. L'inserto è utile perché rende visibile il valore della comunicazione in tempo reale: l'utente non vede soltanto il *file* disponibile, ma anche gli eventi intermedi che hanno caratterizzato l'elaborazione.

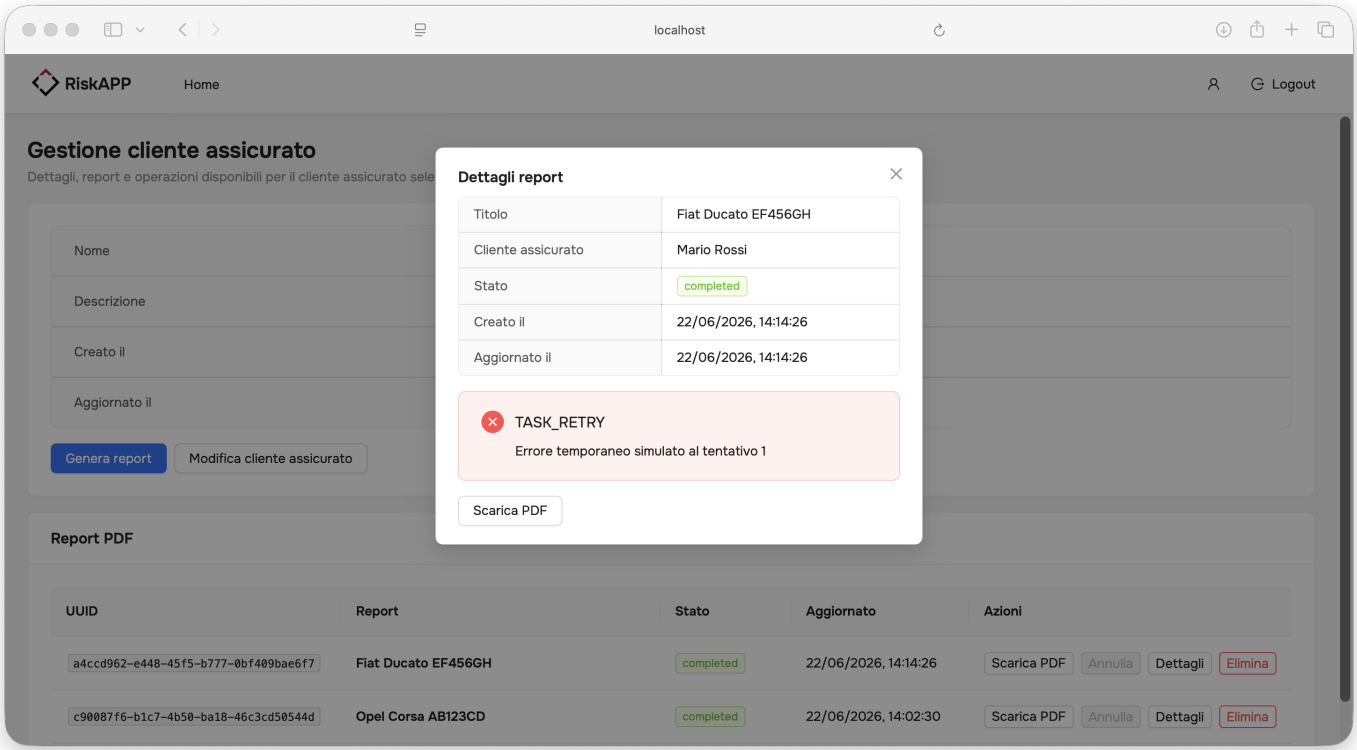


Figura 3.5: Interfaccia di dettaglio di un *report* PDF_A generato. Fonte: schermata acquisita dalla piattaforma sviluppata durante lo *stage*.

Risultati conseguiti

Il risultato è un'interfaccia che rende visibile il ciclo di vita dell'elaborazione asincrona: l'utente può avviare un *report*, seguirne l'evoluzione, ricevere notifiche sugli eventi principali e scaricare il PDF_A quando disponibile. Le schermate mostrate evidenziano quindi non solo l'aspetto finale dell'applicazione, ma anche il modo in cui la comunicazione in tempo reale viene resa percepibile nell'uso ordinario.

3.2.4 Librerie di integrazione

Nella terza fase ho realizzato due librerie pensate per rendere più semplice l'integrazione della soluzione in altri contesti applicativi.

Difficoltà affrontate
La difficoltà principale era evitare che la logica di comunicazione con il microservizio venisse duplicata nei diversi progetti o nei diversi punti dell'applicazione. Senza un livello di integrazione dedicato, ogni *backend* avrebbe dovuto implementare manualmente le chiamate alle API_A del microservizio e ogni *frontend* avrebbe dovuto gestire direttamente connessione WebSocket$_G$, autenticazione, eventi e riconnessione.

Scelte adottate
L'obiettivo era pertanto evitare che ogni applicazione dovesse ricostruire da zero la comunicazione con il microservizio e la gestione degli eventi, concentrando invece la logica ripetibile in componenti riutilizzabili. Per questo sono state realizzate due librerie, una per il lato *backend* e una per il lato *frontend*.

Libreria Python per *backend*
La prima è una libreria Python per *backend* Django, che espone funzioni per creare, recuperare, aggiornare ed eliminare i *task* asincroni tramite API_A $REST_{AG}$.
Su questo livello ho costruito un decoratore$_G$ per *task* Celery, cioè un costrutto usato per aggiungere comportamento a una funzione senza modificarne direttamente il corpo. L'idea era sfruttare i meccanismi messi a disposizione dalla classe dei *task* Celery per agganciare comportamenti personalizzati all'esecuzione del processo: avvio, completamento, errore, nuovo tentativo e annullamento. In questo modo il decoratore$_G$ può occuparsi automaticamente delle chiamate $REST_A$ verso il microservizio e della logica di aggiornamento degli stati principali, lasciando al futuro utilizzatore soltanto la responsabilità di implementare l'elaborazione specifica legata al proprio dominio applicativo. Questa scelta ha reso la libreria più utile rispetto a un semplice insieme di funzioni di supporto: chi integra un nuovo processo asincrono può dichiarare il proprio *task* Celery e applicare il decoratore, ottenendo una gestione coerente dello stato senza dover riscrivere ogni volta le stesse chiamate al microservizio.

Libreria TypeScript per *frontend*
La seconda è una libreria TypeScript destinata a *frontend* React. In questo caso il problema non era soltanto aprire una connessione WebSocket$_G$, ma rendere riutilizzabile l'intero comportamento necessario a ricevere aggiornamenti in tempo reale dal microservizio. Senza un componente dedicato, ogni interfaccia avrebbe dovuto occuparsi direttamente della costruzione dell'URL_A di connessione, dell'autenticazione, della gestione degli eventi ricevuti, della riconnessione e della sincronizzazione con lo stato locale dell'applicazione.
Per questo motivo la libreria espone un hook$_G$ React, cioè una funzione riutiliz-

zabile che incapsula stato e comportamento di un componente. L'*hook* fornisce al *frontend* applicativo un punto di integrazione più semplice: la schermata che lo utilizza non deve conoscere i dettagli del protocollo di comunicazione, ma può reagire a eventi già trasformati in informazioni applicative. In particolare, la libreria gestisce:

- apertura della connessione;
- autenticazione;
- sottoscrizione agli eventi;
- riconnessione automatica;
- trasformazione dei messaggi ricevuti in eventi applicativi più semplici da usare nell'interfaccia.

Questa impostazione ha permesso di separare la logica tecnica della comunicazione in tempo reale dalla logica specifica delle schermate. Nel caso dei *report*, il *frontend* può quindi aggiornare la tabella, mostrare notifiche o invalidare i dati locali senza duplicare in ogni componente la gestione della connessione WebSocket$_G$.

Risultati conseguiti
Le librerie prodotte hanno reso più ordinata l'integrazione tra applicazioni e microservizio, riducendo la quantità di codice ripetitivo necessario per creare *task*, aggiornarne lo stato e ricevere eventi in tempo reale. Durante questa attività ho cercato di evitare un accoppiamento eccessivo con il solo caso d'uso dimostrativo dei *report*: il risultato non è quindi soltanto un supporto alla funzionalità sviluppata nello *stage*, ma una base più adatta al riuso, sulla quale l'azienda può costruire eventuali estensioni future e collegare altri processi asincroni allo stesso modello di gestione.

3.3 Verifica e validazione

3.3.1 Verifica dei flussi principali

La verifica ha avuto lo scopo di controllare che il comportamento del sistema fosse coerente con gli obiettivi definiti e con l'architettura progettata. Mi sono concentrato sui flussi che coinvolgevano *frontend*, *backend* e microservizio, così da accertare che le integrazioni funzionassero non solo a livello dei singoli componenti, ma anche nella loro interazione complessiva.
Gli scenari applicativi verificati sono stati in particolare:

- creazione di un *report* e avvio dell'elaborazione asincrona;
- aggiornamento progressivo dello stato del *task*;
- gestione di errori temporanei con nuovo tentativo;
- gestione degli errori persistenti;

- annullamento della generazione;
- conclusione del processo con disponibilità del PDF_A.

La verifica automatizzata più estesa è stata concentrata sul *backend* applicativo, dove sono stati realizzati 27 *test* basati su $TestCase_G$ di Django, cioè classi usate per definire scenari di *test* controllati. Questi *test* sono stati considerati soprattutto come verifiche di integrazione applicativa, perché coinvolgono viste, modelli, autenticazione, persistenza dei dati e comunicazione simulata con il microservizio. La Tabella 3.2 riassume le principali attività di verifica: l'inserto permette di leggere gli aspetti verificati in relazione alle modalità effettivamente adottate, distinguendo *test* automatici, controlli di $build_G$, controlli di $lint_G$ e prove manuali documentate.

Componente	Aspetti verificati	Modalità di verifica
***Backend* applicativo**	• generazione del PDF_A e calcolo dei dati dimostrativi di rischio; • registrazione, autenticazione e protezione delle API_A; • isolamento dei dati tra utenti; • creazione, cancellazione e annullamento dei *report*; • coerenza del ciclo di vita rispetto allo stato registrato nel microservizio.	• 27 *test* di integrazione applicativa basati su $TestCase_G$ di Django; • uso di *mock* per alcune chiamate verso il microservizio asincrono, così da isolare la dipendenza esterna senza perdere la verifica del flusso del *backend*.
Microservizio per *task* asincroni	• modello dei *task* asincroni e stati ammessi; • creazione, aggiornamento e cancellazione dei *task* tramite API_A interne; • composizione dei messaggi di evento inviati al *frontend*; • autenticazione della connessione WebSocket_G e controllo dei *token* di servizio.	• 15 *test* di unità sulle parti isolate della logica del microservizio; • controlli mirati sul comportamento delle API_A interne e del canale WebSocket_G.
***Frontend* React**	• correttezza della compilazione; • qualità minima del codice dell'interfaccia.	• controlli leggeri di $lint_G$ previsti dalla *repository*; • controlli leggeri di $build_G$ previsti dalla *repository*.

Tabella 3.2: Sintesi delle verifiche svolte sui componenti principali.

La distribuzione delle verifiche non è stata uniforme tra tutti i componenti, ma è stata orientata dai rischi principali del progetto. Sul *backend* applicativo ho privilegiato *test* di integrazione, perché in quel punto si concentravano autenticazione, persistenza dei dati, regole applicative e comunicazione con il microservizio. Sul microservizio ho invece concentrato i *test* di unità sulle funzioni isolate di gestione dello stato dei *task* e di composizione degli eventi. Per il *frontend*, non essendo previsto lo sviluppo di una *suite* completa di *test* automatici, ho verificato almeno la correttezza della compilazione e dei controlli statici previsti dalla *repository*.

3.3.2 Validazione con i referenti aziendali

La validazione è avvenuta tramite confronto progressivo con i referenti aziendali. Le funzionalità realizzate sono state discusse durante l'avanzamento del progetto, verificando che il comportamento ottenuto fosse coerente con il problema di partenza e con le modalità operative attese.
Dal punto di vista operativo, la validazione si è basata sulla prova dei flussi principali e sulla revisione progressiva delle scelte tecniche.
Il confronto con il tutor ha riguardato sia il comportamento visibile all'utente sia la struttura interna della soluzione, con particolare attenzione alla separazione tra *frontend*, *backend* applicativo, microservizio e librerie di integrazione. La documentazione finale ha completato questa validazione, perché ha reso esplicite le modalità di integrazione, configurazione e riuso dei componenti prodotti.

3.4 Documentazione

La documentazione tecnica è stata organizzata distinguendo due destinatari principali: lo sviluppatore, che deve integrare librerie e microservizio in un nuovo progetto, e il futuro manutentore, che dovrà modificarli o estenderli. Una difficoltà di questa attività è stata quindi individuare il futuro lettore di ciascun documento e centrare le informazioni realmente utili per quel profilo, evitando sia descrizioni troppo generiche sia dettagli non necessari rispetto allo scopo operativo. La Tabella 3.3 riassume la documentazione prodotta, chiarendone destinatari e utilità. In particolare, distingue in modo chiaro i documenti finalizzati a fornire indicazioni operative per l'uso dei componenti da quelli destinati a raccogliere le informazioni necessarie alla futura manutenzione.

Documento	Destinatari	Contenuto e utilità
Documentazione dell'architettura del sistema	Sviluppatori e manutentori che devono comprendere l'integrazione tra *backend*, microservizio, librerie e servizi di supporto.	Descrive la separazione delle responsabilità tra componenti, il flusso di gestione dei *task* asincroni, la configurazione tramite Docker Compose, le variabili d'ambiente e gli *endpoint* principali coinvolti nella comunicazione con il microservizio.
Guida all'utilizzo della libreria *frontend*	Sviluppatori che devono integrare la libreria in un'applicazione React.	Descrive lo scopo della libreria, il contesto d'uso, le funzionalità esposte e le modalità principali con cui ricevere e rappresentare gli aggiornamenti provenienti dal microservizio.
Manuale di sviluppo della libreria *frontend*	Futuri sviluppatori di RiskAPP incaricati di aggiornare o manutenere la libreria.	Illustra l'organizzazione interna della libreria, il ruolo dell'hook$_G$ React, la gestione degli stati della connessione e le procedure di verifica locale, così da rendere più semplice modificare la libreria senza alterarne il comportamento atteso.
Documentazione della libreria Python per *backend*	Sviluppatori che devono collegare un *backend* applicativo al microservizio.	Presenta le funzioni esposte dalla libreria per gestire lo stato di *task* asincroni presso il microservizio, insieme all'utilizzo del decoratore$_G$ per *task* Celery.
Documentazione di configurazione del microservizio	Sviluppatori e manutentori che devono installare o integrare il microservizio in un ambiente aziendale.	Spiega come inserire il microservizio in un ambiente Docker Compose, quali variabili d'ambiente valorizzare e quali impostazioni collegare agli *endpoint* del servizio.

Tabella 3.3: Documentazione tecnica prodotta, distinta per destinatari, contenuto e utilità.

3.5 Risultati

3.5.1 Visione d'insieme del prodotto

Al termine delle attività ho ottenuto un sistema composto da più moduli integrati tra loro:

- un microservizio per la gestione dei *task* asincroni;
- un *backend* applicativo per la gestione delle entità e dei *report* PDF_A;
- un *frontend* React per l'interazione con l'utente;
- due librerie *client* pensate per semplificare l'integrazione della soluzione in altri contesti.

Dal punto di vista dell'utente, il risultato principale è il **passaggio da un modello basato su interrogazioni ripetute del *client***, cioè sul polling$_G$, **a un flusso in cui lo stato delle elaborazioni viene aggiornato lato *server* e comunicato in tempo reale al *frontend***. In questo modo l'utente può seguire l'avanzamento delle operazioni asincrone e ricevere notifiche sul loro esito, mentre i componenti applicativi mantengono responsabilità separate.

3.5.2 Risultati quantitativi

Il risultato quantitativo può essere riassunto distinguendo copertura dei requisiti, copertura delle verifiche e quantità di prodotti consegnati. La lettura della copertura dei requisiti tiene come riferimento gli obiettivi aziendali definiti nella Tabella 2.1, così da collegare i risultati ottenuti al perimetro iniziale dello *stage*. La Tabella 3.4 raccoglie questi dati in forma sintetica: l'inserto è utile perché mette in relazione il lavoro svolto con i risultati quantitativi.

Ambito	Risultato quantitativo
Copertura dei requisiti	• OB1-OB10: soddisfatti. • D1: non realizzato. • F1: non realizzato.

Ambito	Risultato quantitativo
Copertura dei *test*	• 27 *test* automatici di integrazione presenti nella *repository* del *backend* applicativo, tutti superati; • copertura del codice applicativo della *repository* del *backend* pari al 77%, calcolata con *coverage.py* sul modulo applicativo, escludendo *test* e migrazioni; • 15 *test* automatici di unità per il microservizio, tutti superati, dedicati alla verifica delle funzioni principali di gestione dello stato dei *task*; • scenari principali coperti: generazione dei *report*, autenticazione e protezione delle API_A, isolamento dei dati tra utenti, comunicazione con il microservizio e coerenza del ciclo di vita dei *task*; • controlli di $lint_G$ e $build_G$ eseguiti su *frontend* e libreria TypeScript, tutti superati.
Documentazione tecnica	• Documentazione dell'architettura del sistema; • guida all'utilizzo della libreria *frontend*; • manuale di sviluppo della libreria *frontend*; • documentazione della libreria *backend*; • documentazione di configurazione del microservizio;
Repository software	• *Repository* del *frontend* React; • *repository* del *backend* Django; • *repository* del microservizio per *task* asincroni; • *repository* della libreria Python per *backend*; • *repository* della libreria TypeScript per *frontend* React.
Codice sviluppato	• 108 *file* rilevanti tra codice, configurazione e documentazione; • circa 5576 righe di codice Python, TypeScript e JavaScript.

Tabella 3.4: Risultati quantitativi del progetto, distinti per requisiti, verifiche, documentazione, *repository* e codice prodotto.

Capitolo 4

Valutazione retrospettiva

4.1 Soddisfacimento degli obiettivi

In questa sezione valuto il raggiungimento degli obiettivi concordati con l'azienda e degli obiettivi personali, basandomi sui dati di fatto emersi dal progetto, sui risultati quantitativi riportati nella Tabella 3.4, sulle verifiche svolte e sui riscontri ricevuti durante lo *stage* nel confronto con il tutor aziendale e con i referenti tecnici. Il bilancio che propongo non riguarda soltanto la presenza degli artefatti prodotti, ma anche il grado con cui tali artefatti rispondono al problema iniziale e al percorso formativo che mi ero prefissato.
La proposta di *stage* aveva infatti una doppia natura: da un lato richiedeva la realizzazione di componenti *software* utili all'azienda, dall'altro rappresentava per me un'occasione di crescita tecnica e professionale. Per questo motivo distinguo gli obiettivi aziendali, legati al prodotto consegnato, dagli obiettivi personali, legati alle competenze che ho acquisito durante il lavoro. Nella valutazione, raggruppo gli obiettivi affini per evitare ripetizioni, ma esplicito comunque i codici a cui mi riferisco e l'esito ottenuto.

Obiettivi aziendali

- **Realizzazione del microservizio (OB1)**: l'obiettivo principale dell'azienda era introdurre un componente dedicato alla gestione di operazioni asincrone, capace di separare questa responsabilità dal resto dell'applicazione. Ho raggiunto questo risultato realizzando un microservizio basato su Django, Django $REST_A$ Framework e Django Channels, poi integrato con il *backend* applicativo e con il *frontend*.
  Sulla base degli artefatti consegnati e del flusso verificato nel caso d'uso dei *report* PDF_A, ho raggiunto l'obiettivo; di conseguenza, è **soddisfatto**. Lo dimostra la presenza di un componente autonomo, organizzato in una *repository* dedicata, integrato con il *backend* e con il *frontend*: il *backend* può avviare e seguire le elaborazioni, il microservizio può gestire lo stato dei *task* e il *frontend* può ricevere aggiornamenti senza dipendere soltanto da richieste periodiche. Il passaggio da un modello basato su $polling_G$ a un modello con notifiche tramite

WebSocket$_G$ risponde quindi al problema tecnico individuato all'inizio dello *stage*.

- **Librerie di integrazione (OB2, OB3)**: la proposta prevedeva la produzione di una libreria Python per l'integrazione con altri *backend* Django (OB2) e di una libreria TypeScript per l'utilizzo da parte di *frontend* React (OB3). Ho raggiunto entrambi gli obiettivi; di conseguenza, sono **soddisfatti**. L'evidenza non è soltanto la presenza del codice: ho sviluppato le due librerie come componenti separati, con *repository* dedicate, documentazione tecnica e un ruolo preciso nel flusso applicativo.
  Nel caso della libreria Python, ho soddisfatto OB2 centralizzando le chiamate verso il microservizio e supportando la gestione automatica degli stati dei *task* Celery. Per quanto riguarda la libreria TypeScript, ho soddisfatto OB3 realizzando un punto di accesso riutilizzabile per la comunicazione in tempo reale lato *frontend*, separato dalla logica specifica delle schermate. In questo modo il risultato non si limita a duplicare codice applicativo, ma rende più chiara e riutilizzabile la responsabilità di integrazione con il microservizio.

- **Comunicazione tra i componenti (OB4, OB5)**: una parte rilevante degli obiettivi obbligatori riguardava la definizione dei canali di comunicazione tra *backend*, microservizio e *frontend*. Questo aspetto era importante perché la soluzione doveva inserirsi in un contesto applicativo già esistente e non poteva essere pensata come un prototipo separato dal prodotto aziendale.
  Il bilancio è positivo: ho raggiunto e reso **soddisfatti** entrambi gli obiettivi, realizzando la comunicazione tra *backend* e microservizio (OB4) tramite API_A $REST_{AG}$ e gestendo la comunicazione tra microservizio e *frontend* (OB5) tramite WebSocket$_G$. Lo dimostrano i flussi in cui tali comunicazioni sono effettivamente usate: registrazione del *task*, aggiornamento dello stato, generazione del *report* e notifica verso l'interfaccia. Le prove svolte e i *test* di integrazione sul *backend* confermano quindi che i componenti sviluppati sono integrabili e coerenti con l'architettura prevista.

- **Funzionalità applicative e gestione dei *report* (OB6)**: tra gli obiettivi obbligatori rientrava anche l'implementazione delle funzionalità applicative necessarie alla gestione delle principali entità e dei *report* PDF_A collegati al *frontend*. Ho realizzato un risultato che comprende il sistema $CRUD_A$, l'avvio della generazione dei *report* e il tracciamento dello stato delle elaborazioni.
  Sulla base delle funzionalità implementate e verificate, ho raggiunto l'obiettivo; di conseguenza, è **soddisfatto**. Lo dimostrano le operazioni $CRUD_A$, l'avvio della generazione, il tracciamento dello stato e la possibilità per l'utente di ricevere aggiornamenti sull'elaborazione. Ho dunque integrato la generazione dei *report*, che rappresentava un caso concreto di operazione asincrona, nel flusso complessivo della soluzione, evitando che rimanesse una funzionalità isolata o opaca per l'utente.

- **Documentazione, verifiche e manutenibilità (OB7, OB8, OB9, OB10)**: ho affrontato gli obiettivi relativi alla documentazione e ai *test* nella parte

conclusiva dello *stage*, quando la soluzione aveva raggiunto una forma sufficientemente stabile. Ho raggiunto gli obiettivi documentali; di conseguenza, sono **soddisfatti**: ho prodotto la documentazione dell'architettura del sistema, delle comunicazioni tra componenti e delle istruzioni di integrazione del microservizio (OB7); ho redatto la guida all'utilizzo della libreria *frontend*, il manuale di sviluppo della stessa libreria, la documentazione della libreria Python per *backend* e la documentazione di configurazione del microservizio (OB8). Ho raggiunto anche gli obiettivi di verifica, completando le relative attività: ho realizzato 15 *test* di unità sul microservizio (OB9) e 27 *test* di integrazione sul *backend* applicativo (OB10). Inoltre, ho eseguito controlli di $lint_G$ e $build_G$ su *frontend* e libreria TypeScript con esito positivo e ho misurato una copertura del codice applicativo pari al 77%.
In questo caso il soddisfacimento è dimostrato soprattutto dai dati quantitativi: la presenza di *test* automatici superati, controlli di $lint_G$ e $build_G$ con esito positivo e la copertura misurata dei *test* rende il bilancio misurabile. Gli scenari verificati hanno riguardato generazione dei *report*, autenticazione e protezione delle API_A, isolamento dei dati tra utenti, comunicazione con il microservizio e coerenza del ciclo di vita dei *task*. Dunque, con la documentazione ho esplicitato le scelte principali e le modalità di integrazione, mentre con i *test* ho ottenuto un riscontro sul comportamento delle parti più importanti del sistema.

- **Obiettivi desiderabili e facoltativi (D1, F1)**: non ho raggiunto gli obiettivi non obbligatori relativi alla *dashboard* di monitoraggio (D1) e a strumenti aggiuntivi di *monitoring* lato *frontend* (F1); di conseguenza, **non sono soddisfatti**. Questo esito non rappresenta però un fallimento del nucleo progettuale, perché tali attività erano state definite fin dall'inizio come estensioni subordinate allo svolgimento delle parti principali.
  Questo bilancio resta coerente con la priorità assegnata agli obiettivi: D1 e F1 erano estensioni non obbligatorie, mentre il nucleo OB1–OB10 è stato raggiunto e verificato. La scelta di rimandarle è stata concordata con l'azienda, non solo per bilanciare il tempo disponibile, ma anche per approfondire meglio gli obiettivi obbligatori e consolidare il valore del lavoro già prodotto. Per questo motivo restano possibili come sviluppi futuri, senza compromettere in alcun modo il raggiungimento degli obiettivi fondamentali dello *stage*.

La Figura 4.1 sintetizza il bilancio dei soli obiettivi aziendali in base alla priorità con cui erano stati definiti nel piano di lavoro. L'inserto è utile perché mostra subito che ho raggiunto il nucleo obbligatorio, mentre non ho svolto le attività desiderabili e facoltative entro il perimetro effettivamente consegnato.

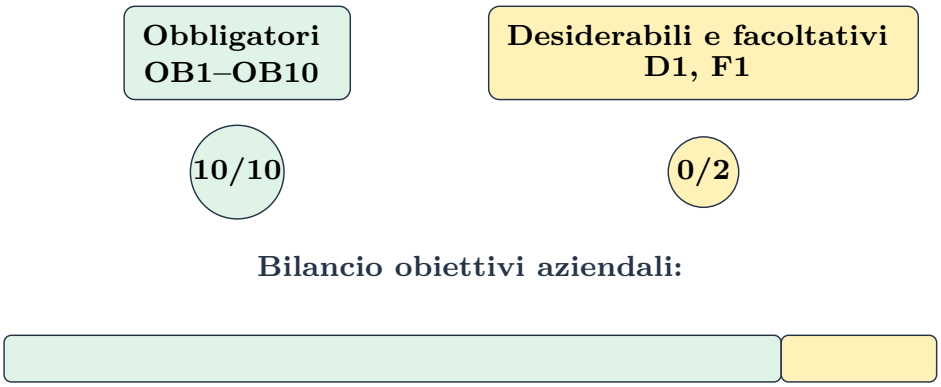


Figura 4.1: Bilancio degli obiettivi aziendali dello *stage*, distinguendo quelli obbligatori da quelli desiderabili e facoltativi.

Obiettivi personali

- **Autonomia nell'analisi e nelle scelte tecniche (OP1)**: il primo obiettivo personale riguardava la capacità di affrontare un problema reale con un margine di autonomia maggiore rispetto ai progetti universitari. Durante lo *stage* non ho ricevuto soltanto una lista di istruzioni da implementare, ma un'esigenza generale da comprendere, tradurre in scelte architetturali e discutere con il tutor aziendale.
  Il bilancio è positivo: ho raggiunto l'obiettivo perché ho dovuto motivare e discutere scelte progettuali concrete, in particolare nella definizione del ruolo del microservizio, nella separazione delle responsabilità tra componenti e nell'impostazione delle librerie di integrazione. Questo mi ha aiutato a distinguere meglio tra requisito, vincolo tecnico e scelta implementativa. In particolare, il confronto con l'azienda mi ha permesso di capire che una soluzione corretta non è solo quella che funziona localmente, ma quella che si inserisce in modo comprensibile nel contesto esistente e rimane sostenibile per chi dovrà mantenerla.

- **Approfondimento delle soluzioni asincrone distribuite (OP2)**: un secondo obiettivo personale era approfondire la gestione di processi asincroni e la comunicazione in tempo reale, tema che conoscevo solo parzialmente prima dello *stage*. La realizzazione del flusso basato su Django Channels, Celery e WebSocket$_G$ mi ha permesso di affrontare concretamente problemi legati allo stato dei *task*, alla propagazione degli aggiornamenti e alla separazione tra elaborazione e interfaccia utente.
  La retrospettiva è positiva: ho raggiunto l'obiettivo perché non mi sono limitato a uno studio teorico, ma ho utilizzato queste tecnologie dentro un flusso applicativo completo, collegato alla generazione dei *report*, all'aggiornamento dello stato dei *task* e alla notifica verso il *frontend*. Questo mi ha portato a comprendere meglio le difficoltà tipiche dei sistemi distribuiti, come la coerenza dello stato, la gestione degli errori e la necessità di rendere osservabile l'avanzamento di un'operazione non immediata.

- **Progettazione di componenti riutilizzabili (OP3)**: un altro obiettivo personale era migliorare la capacità di progettare componenti pensati per essere riutilizzati da altri sviluppatori. Lo sviluppo delle librerie Python e TypeScript mi ha costretto a ragionare non solo sul comportamento interno della soluzione, ma anche sull'interfaccia offerta verso l'esterno.
  Ho raggiunto l'obiettivo realizzando due librerie distinte, una per l'integrazione lato *backend* e una per l'integrazione lato *frontend*, accompagnate dalla relativa documentazione tecnica. Il lavoro sulle librerie mi ha fatto comprendere meglio l'importanza di API_A chiare, nomi coerenti, configurazioni esplicite e documentazione di supporto. Una funzionalità può essere tecnicamente corretta, ma può diventare difficile da adottare se non espone un punto di integrazione comprensibile.

- **Verifica, documentazione e responsabilità sul risultato (OP4)**: l'ultimo obiettivo personale riguardava il consolidamento del metodo di verifica e documentazione. Nel corso dello *stage* ho prodotto *test*, documentazione tecnica e materiali di supporto, cercando di rendere il lavoro valutabile e riprendibile anche dopo la conclusione dell'esperienza.
  Ho raggiunto questo obiettivo attraverso attività verificabili: *test* automatici, controlli di $lint_G$ e $build_G$, misurazione della copertura e documentazione tecnica prodotta a supporto dell'integrazione. Ho considerato la verifica non come una fase separata e finale, ma come un'attività necessaria per controllare progressivamente la coerenza tra obiettivi, implementazione e comportamento effettivo del sistema. Allo stesso modo, la documentazione mi ha aiutato a rendere esplicite le decisioni prese e a lasciare all'azienda un riferimento utile per eventuali sviluppi successivi.

Riepilogo quantitativo

La Figura 4.2 riassume il bilancio complessivo degli obiettivi, distinguendo il nucleo aziendale obbligatorio, le estensioni rimandate e gli obiettivi personali. L'inserto è utile perché consente di leggere a colpo d'occhio l'esito del soddisfacimento degli obiettivi, senza sostituire la valutazione argomentata dei punti precedenti.

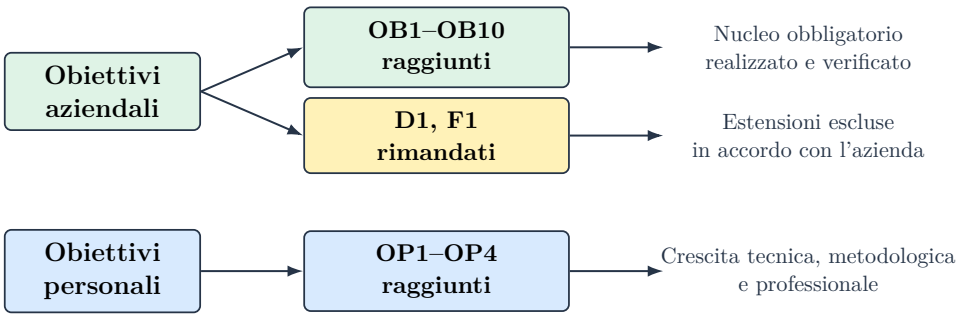


Figura 4.2: Sintesi del raggiungimento degli obiettivi aziendali e personali dello *stage*.

4.2 Maturazione professionale

Lo *stage* ha rappresentato un passaggio significativo nel mio percorso di maturazione professionale, perché mi ha permesso di lavorare su un problema applicativo reale, nel quale il risultato non dipendeva soltanto dal funzionamento del singolo componente, ma anche dall'integrazione con il contesto aziendale. La crescita è maturata su tre piani principali: tecnico, metodologico e organizzativo. Per evitare una descrizione solo discorsiva, accompagno il bilancio con tre tabelle distinte: ciascuna collega le attività effettivamente svolte alle competenze sviluppate e alle responsabilità assunte. Questa forma è utile perché rende più chiaro il passaggio tra ciò che ho realizzato nel progetto e ciò che ne ho ricavato in termini professionali.

Ambito tecnico
Dal punto di vista tecnico ho consolidato competenze già presenti e ne ho acquisite di nuove, soprattutto nella gestione di flussi asincroni e nella comunicazione tra componenti distribuiti. Il lavoro non si è limitato allo sviluppo del microservizio, ma ha richiesto di collegare microservizio, *backend*, *frontend*, librerie di integrazione, generazione dei *report* e verifiche automatiche in un unico sistema coerente.
La Tabella 4.1 riporta le principali attività tecniche svolte durante lo *stage* e le mette in relazione con le competenze maturate e con le responsabilità operative assunte. L'inserto è utile perché mostra che la crescita tecnica non ha riguardato un solo strumento, ma l'intero flusso applicativo: dalla modellazione dei *task* asincroni fino all'integrazione tra componenti e alla verifica del comportamento prodotto.

Attività svolte	Competenze maturate	Responsabilità assunte
• Studio e sperimentazione iniziale delle tecnologie previste.	• Comprensione pratica di Django, Django Channels, Celery, React e TypeScript; • capacità di verificare i concetti studiati tramite prototipi mirati.	• Arrivare allo sviluppo con un perimetro tecnico più chiaro; • rispettare lo *stack* e le modalità di lavoro già presenti in azienda.
• Progettazione e implementazione del microservizio per la gestione dei *task* asincroni.	• Uso di Django, Django $REST_A$ Framework e Django Channels; • modellazione dello stato e degli eventi di un'elaborazione asincrona.	• Garantire l'autonomia del componente; • assicurare funzionalità coerenti con il ruolo assegnato al microservizio.
• Integrazione tra *backend*, microservizio e *frontend* React.	• Progettazione di un sistema modulare; • gestione di comunicazioni tramite API_A $REST_{A\,G}$ e WebSocket_G.	• Garantire il funzionamento *end-to-end* del flusso; • preservare la separazione dei compiti tra componenti.

Attività svolte	Competenze maturate	Responsabilità assunte
• Sviluppo del *backend* applicativo per entità e *report*.	• Modellazione delle entità applicative; • gestione di funzionalità $CRUD_A$ e regole di accesso.	• Collegare la gestione asincrona a un caso d'uso applicativo completo; • mantenere separati dati di dominio e stato tecnico dei *task*.
• Implementazione del flusso dei *report* PDF_A.	• Gestione di avvio, stati, annullamento e *download*; • collegamento tra elaborazione Celery e percezione lato utente.	• Rendere osservabile il ciclo di vita dei *report*; • gestire casi di errore, completamento e disponibilità del *file*.
• Sviluppo delle librerie Python e TypeScript.	• Progettazione di componenti riutilizzabili e configurabili; • definizione di punti di integrazione chiari per altri progetti.	• Evitare accoppiamento con il solo caso dimostrativo; • lasciare una base tecnica riusabile dall'azienda.
• Realizzazione di *test*, controlli di $lint_G$ e $build_G$.	• Progettazione di verifiche automatiche e controlli statici; • interpretazione dei risultati quantitativi: 27 *test* di integrazione, 15 *test* di unità e copertura del 77% sul modulo principale del *backend*.	• Documentare evidenze oggettive sul funzionamento; • ridurre il rischio di regressioni sulle parti principali.

Tabella 4.1: Attività, competenze e responsabilità maturate nell'ambito tecnico dello *stage*.

Ambito metodologico

Sul piano metodologico ho maturato una maggiore capacità di passare dall'analisi di un'esigenza generale a scelte progettuali motivate. In particolare, il confronto tra alternative, la separazione delle responsabilità e la verifica progressiva mi hanno aiutato a controllare meglio il rapporto tra requisiti, progettazione, implementazione e risultati conseguiti. Questo aspetto è stato importante perché il progetto non richiedeva soltanto di scrivere codice funzionante, ma anche di giustificare le scelte, verificare di aver compreso correttamente il problema e produrre documentazione utile a chi avrebbe ripreso il lavoro.

La Tabella 4.2 sintetizza quindi le attività più legate al metodo di lavoro: analisi iniziale, progettazione dell'architettura e documentazione tecnica. Il suo scopo è evidenziare come, durante lo *stage*, abbia imparato a trasformare un'esigenza applicativa in decisioni tracciabili, confrontabili con i referenti aziendali e coerenti

con gli obiettivi del progetto.

Attività svolte	Competenze maturate	Responsabilità assunte
• Analisi del problema applicativo e studio del contesto.	• Traduzione di un'esigenza generale in vincoli e obiettivi; • uso di bozze personali per verificare la comprensione.	• Chiarire il perimetro prima della codifica; • confrontare la comprensione con tutor e referenti tecnici.
• Definizione dell'architettura della soluzione.	• Separazione delle responsabilità tra *backend*, microservizio e *frontend*; • valutazione di alternative progettuali.	• Motivare le scelte tecniche; • mantenere coerenza tra architettura, vincoli aziendali e obiettivi.
• Produzione della documentazione tecnica.	• Scrittura orientata al futuro lettore; • sintesi di configurazione, uso e sviluppo dei componenti; • organizzazione della documentazione per destinatario e componente.	• Rendere il lavoro riprendibile; • spiegare decisioni e modalità di integrazione senza dipendere dalla sola memoria del progetto.

Tabella 4.2: Attività, competenze e responsabilità maturate nell'ambito metodologico dello *stage*.

Ambito organizzativo

Sul piano organizzativo ho imparato a gestire un lavoro meno guidato rispetto ai progetti universitari, mantenendo l'allineamento con il tutor aziendale e con i referenti tecnici. La gestione delle priorità è stata rilevante soprattutto nella fase conclusiva, quando ho dovuto consolidare il nucleo obbligatorio del progetto, completare le verifiche e lasciare documentazione utile per il lavoro successivo. Questo mi ha portato a ragionare non solo sull'attività immediata, ma anche sull'impatto delle scelte sul lavoro dell'azienda e sulla possibilità di permettere il proseguimento dello sviluppo a RiskAPP dopo la fine dello *stage*.

La Tabella 4.3 raccoglie gli aspetti legati alla pianificazione, al confronto periodico, alla gestione delle priorità e all'organizzazione degli artefatti prodotti. L'utilità dell'inserto è mostrare che la maturazione organizzativa non coincide solo con la gestione del tempo, ma comprende anche la capacità di comunicare l'avanzamento, recepire *feedback* e lasciare un risultato ordinato e riprendibile.

Attività svolte	Competenze maturate	Responsabilità assunte
• Pianificazione delle attività e confronto periodico.	• Maggiore autonomia nella gestione del lavoro; • comunicazione di avanzamento, dubbi e risultati.	• Mantenere l'allineamento con tutor e referenti tecnici; • recepire *feedback* e vincoli emersi durante lo sviluppo.
• Gestione delle priorità nella fase conclusiva.	• Distinzione tra nucleo obbligatorio ed estensioni; • valutazione del rapporto tra tempo disponibile, qualità e riuso.	• Raggiungere prima gli obiettivi obbligatori; • rimandare consapevolmente attività desiderabili e facoltative.
• Organizzazione del lavoro in *repository* e componenti separati.	• Organizzazione ordinata degli artefatti prodotti; • consapevolezza della manutenibilità oltre il proprio intervento; • gestione di componenti distribuiti su cinque *repository*.	• Lasciare un risultato verificabile, documentato e riprendibile; • facilitare eventuali estensioni future da parte dell'azienda.

Tabella 4.3: Attività, competenze e responsabilità maturate nell'ambito organizzativo dello *stage*.

4.3 Rapporto con il percorso di studi

Il rapporto tra il percorso di studi e lo *stage* non può essere letto come una corrispondenza diretta tra singoli insegnamenti e singole tecnologie utilizzate. Nel progetto di *stage*, infatti, ho lavorato con strumenti che non conoscevo in modo specifico prima dell'esperienza, come Django Channels, Celery e la comunicazione tramite WebSocket$_G$. Questo però non riduce il ruolo della formazione universitaria: il corso di laurea mi ha fornito una base tecnica, metodologica e organizzativa che si è rivelata essenziale per comprendere strumenti nuovi e inserirli in una soluzione coerente. Per questo motivo, il confronto viene presentato partendo dalle conoscenze già possedute, passando poi dalle lacune emerse nel contesto aziendale, fino alle competenze che ho sviluppato autonomamente per colmare quella distanza. Le attività svolte durante il tirocinio riprendono infatti fasi riguardanti il ciclo di vita del *software* già incontrate nel percorso universitario, come analisi, progettazione, codifica, verifica e documentazione; la differenza principale riguarda il grado di autonomia, la concretezza del contesto aziendale e la responsabilità diretta sul risultato consegnato.

Conoscenze già possedute
Le conoscenze più direttamente riutilizzate sono state quelle legate allo sviluppo del *software* e alla progettazione di sistemi composti da più parti. Non si è trattato soltanto di una base teorica: alcune competenze tecniche erano già state esercitate nei progetti universitari, anche se in contesti più controllati rispetto a quello aziendale. In particolare, mi sono stati utili:

- la capacità di analizzare un problema e scomporlo in componenti;
- la modellazione dei dati e delle entità applicative;
- la separazione delle responsabilità tra moduli;
- la progettazione di interfacce e API_A per far comunicare sistemi distinti;
- la verifica progressiva del comportamento tramite *test* e controlli.

Queste basi non mi hanno fornito una soluzione pronta, ma hanno permesso di orientarmi nel progetto e di apprendere più rapidamente le tecnologie richieste. Per esempio, la distinzione tra dati di dominio e stato tecnico dei *task* deriva da un modo di ragionare già incontrato nel percorso universitario: separare concetti diversi, assegnare responsabilità chiare ai componenti e verificare che l'integrazione complessiva resti coerente. Nello *stage* ho applicato lo stesso approccio in un ambiente meno guidato, nel quale le decisioni dovevano essere confrontate con referenti aziendali e mantenute coerenti con un possibile riuso futuro.
La Tabella 4.4 chiude questo primo passaggio mettendo in relazione le basi fornite dal percorso universitario con la loro applicazione nel progetto di *stage*. L'inserto è utile perché evita un confronto diretto tra singoli progetti e mostra invece il passaggio più rilevante: competenze già costruite nel percorso di studi sono state applicate in un contesto aziendale più concreto, meno guidato e maggiormente orientato alla consegna di un risultato riutilizzabile.

Ambito	Base fornita dal percorso universitario	Applicazione nello *stage*
Analisi e progettazione	• Analisi di esigenze e vincoli; • definizione di requisiti; • progettazione architetturale; • metodo orientato a obiettivi verificabili.	• Applicare il metodo con maggiore autonomia; • definire il ruolo del microservizio nel sistema; • motivare le scelte rispetto a vincoli e aspettative aziendali.

Ambito	Base fornita dal percorso universitario	Applicazione nello *stage*
Modellazione e persistenza	• Rappresentazione di entità e relazioni; • progettazione dei dati a supporto delle funzionalità; • coerenza tra modello, requisiti e comportamento atteso.	• Modellare clienti assicurati, *report* e *task*; • mantenere separato il dominio applicativo dallo stato dell'elaborazione asincrona.
Integrazione tra componenti	• Progettazione e uso di API_A; • separazione tra interfaccia, logica applicativa e servizi.	• Applicare la separazione a un sistema distribuito; • collegare microservizio, *backend*, *frontend* e librerie; • integrare comunicazioni $REST_{A\,G}$, WebSocket$_G$, Django Channels e Celery.
Verifica e qualità	• Controllo progressivo del comportamento; • attenzione a *test*, casi limite e regressioni; • abitudine a collegare implementazione e verifica.	• Decidere in autonomia dove concentrare le verifiche; • produrre *test* di unità e integrazione; • combinare copertura, $lint_G$, $build_G$ e prove manuali documentate.
Documentazione e riuso	• Capacità di descrivere scelte tecniche e risultati; • attenzione alla comprensibilità del lavoro svolto; • produzione di materiali coerenti con le fasi del progetto.	• Scrivere documentazione per utilizzatori e manutentori reali; • rendere riutilizzabili le librerie Python e TypeScript; • lasciare una base riprendibile dall'azienda dopo lo *stage*.

Tabella 4.4: Confronto tra basi fornite dal percorso universitario e loro applicazione nel progetto di *stage*.

Lacune personali emerse

Dopo aver ricondotto le attività dello *stage* alle basi già possedute, il confronto con il mondo del lavoro ha messo in evidenza anche alcune lacune personali. Non le considero come mancanze assolute del percorso universitario, ma come distanze concrete che, nella fase iniziale, hanno reso alcune attività meno immediate e hanno richiesto uno sforzo di adattamento. Questi aspetti sono emersi soprattutto perché il progetto si collocava in un contesto reale, con vincoli aziendali, tecnologie già scelte,

aspettative di riuso e necessità di coordinamento:

- **Valutazione dei compromessi tecnici**: durante lo *stage* ho dovuto imparare a cercare un equilibrio tra la libertà nelle scelte progettuali e la necessità di rimanere coerente con il contesto RiskAPP, con le sue convenzioni, le sue tecnologie e le sue modalità operative. La difficoltà non stava tanto nell'individuare una soluzione funzionante, quanto più nel capire quale soluzione fosse adeguata dentro un sistema già esistente. Nei progetti universitari questo compromesso era meno marcato, perché lo spazio decisionale risultava generalmente più libero e meno vincolato da pratiche aziendali già esistenti.

- **Mancanza di conoscenze tecniche specifiche**: all'inizio dello *stage* non conoscevo in modo operativo alcune tecnologie centrali per il progetto, come Django Channels e il protocollo di comunicazione WebSocket$_G$. Questo ha rappresentato una difficoltà iniziale, perché prima di poter contribuire in modo efficace è stato necessario comprenderne il funzionamento e verificarne l'applicazione nel caso concreto. Si tratta comunque di una lacuna marginale e fisiologica, perché il percorso universitario non può coprire tutte le tecnologie usate nei diversi contesti aziendali; la base ricevuta mi ha però permesso di studiarle autonomamente e di inserirle progressivamente nella soluzione.

- **Responsabilità operative**: con l'avanzare dello *stage* ho percepito un passaggio graduale da un'attività guidata a una gestione più autonoma del lavoro. Questa transizione non è stata immediata, perché richiedeva di assumere maggiore sicurezza nel portare avanti decisioni e risultati senza un controllo costante su ogni passaggio. Rispetto ai progetti universitari, in cui il percorso è generalmente più scandito da indicazioni, revisioni e consegne intermedie, il progetto aziendale mi ha richiesto di assumere maggiore responsabilità sulle decisioni prese, sulla qualità del risultato e sulla sua effettiva utilità per l'azienda. Questo mi ha fatto avvicinare a una modalità di lavoro più simile a quella di una figura inserita operativamente nel progetto.

Competenze acquisite autonomamente

Le lacune emerse non sono rimaste elementi separati dalla realizzazione del progetto, ma sono diventate il punto di partenza per nuove competenze. Durante lo *stage* è stato necessario trasformare le basi già possedute in capacità operative più mature. Ho dovuto studiare autonomamente tecnologie e modalità di lavoro nuove, non in modo astratto, perché servivano a risolvere problemi concreti del progetto: gestire elaborazioni asincrone, notificare aggiornamenti in tempo reale, collegare *backend*, microservizio e *frontend*, produrre librerie riutilizzabili e lasciare documentazione utile. Questo percorso si è concretizzato soprattutto in tre aspetti:

- ho imparato a valutare le scelte tecniche non solo in base al fatto che funzionassero, ma anche rispetto al contesto RiskAPP, alle sue convenzioni e alla possibilità di mantenerle dopo la fine dello *stage*;

- ho studiato le tecnologie richieste con l'obiettivo di applicarle a problemi concreti, anche attraverso prove iniziali e programmi di *test* prima dell'integrazione nel progetto;
- ho acquisito maggiore autonomia nel pianificare le attività, verificare progressivamente i risultati e rendere il lavoro comprensibile anche per chi lo avrebbe ripreso in seguito, attraverso codice ordinato, librerie riutilizzabili e documentazione.

La Figura 4.3 sintetizza il passaggio descritto nel paragrafo: le basi universitarie hanno fornito il metodo, mentre l'apprendimento personale ha permesso di adattarlo alle tecnologie e al contesto RiskAPP. L'inserto è volutamente essenziale, perché non sostituisce il confronto svolto nel testo, ma rende immediata la relazione tra formazione ricevuta, studio autonomo e applicazione nel progetto di *stage*.

Figura 4.3: Relazione tra basi universitarie, apprendimento personale e conoscenze applicate nel progetto di *stage*.

Acronimi e abbreviazioni

API Application Programming Interface. 7, 16, 22, 25, 27, 29, 34, 36, 40, 42, 43, 45, 46, 50, 51

CRUD Create, Read, Update, Delete. 15, 27, 42, 47

HTTP HyperText Transfer Protocol. iii, 7, 14

IT Information Technology. 1

JWS JSON Web Signature. 27

PDF Portable Document Format. ix, 15, 21, 25, 28–30, 32, 33, 36, 39, 41, 42, 47

PMI Piccole e Medie Imprese. 3

REST Representational State Transfer. 7, 14–16, 22, 25, 27, 34, 41, 42, 46, 51

URL Uniform Resource Locator. 34

UUID Universally Unique Identifier. 32

Glossario

API REST interfaccia che permette a componenti *software* diversi di comunicare tramite richieste *HTTP* secondo i principi architetturali *REST*, esponendo funzionalità e dati tramite regole condivise. 7, 16, 22, 25, 27, 34, 42, 46, 51

background modalità di esecuzione di un'attività che viene svolta senza bloccare la risposta immediata del sistema verso l'utente. 8

build processo con cui il codice sorgente viene preparato per l'esecuzione o la distribuzione, tramite compilazione, controllo delle dipendenze e generazione degli artefatti necessari. 36, 40, 43, 45, 47, 51

cache area di memorizzazione temporanea utilizzata per conservare dati ad accesso frequente e ridurre i tempi di recupero delle informazioni. 7

Commercial Lines insieme delle coperture assicurative rivolte ad aziende, professionisti e organizzazioni, in contrapposizione alle polizze destinate a clienti privati. 1, 3

decoratore costrutto di programmazione che permette di estendere o modificare il comportamento di una funzione o di un metodo senza intervenire direttamente sul suo codice interno. In Python viene spesso usato per aggiungere logica ricorrente, come controlli, tracciamento o aggiornamenti di stato, mantenendo più pulita la funzione principale. 34, 38

funnel sequenza di passaggi attraverso cui un'opportunità commerciale viene seguita, filtrata e progressivamente avvicinata alla conclusione. 3

hook funzione di React che permette a un componente funzionale di gestire stato, effetti e logica riutilizzabile senza ricorrere a classi. Gli *hook* sono utili per isolare comportamenti comuni, come la gestione di una connessione, l'aggiornamento dei dati o la reazione a eventi esterni, rendendoli riutilizzabili in più componenti. 34, 38

insurtech settore che applica tecnologie digitali, automazione e analisi dei dati ai processi assicurativi, con l'obiettivo di semplificare la gestione delle polizze, la valutazione del rischio e l'interazione tra operatori, intermediari e clienti. iii, 1

lead contatto commerciale potenzialmente interessato a un prodotto o servizio, che può essere seguito e qualificato nelle fasi successive del processo. iii, 3

lint analisi automatica del codice sorgente usata per individuare errori, possibili problemi di qualità e violazioni delle convenzioni stilistiche definite dal progetto. 36, 40, 43, 45, 47, 51

modello dati componente che descrive la struttura di un'entità gestita dall'applicazione, indicando i campi da memorizzare, il loro tipo e alcune regole di validazione o comportamento. In un *framework* come Django il modello dati viene usato anche per collegare il codice applicativo alla tabella corrispondente nel *database*. ix, 27, 28

polling tecnica di comunicazione in cui un componente interroga periodicamente un altro servizio per verificare la presenza di aggiornamenti o il completamento di un'operazione. 13, 14, 21, 22, 39, 41

pull request proposta di modifica al codice sorgente, utilizzata per sottoporre un'implementazione a revisione prima dell'integrazione nel progetto. 7

reverse proxy componente che riceve le richieste dirette a un'applicazione e le inoltra al servizio interno appropriato. 8

TestCase classe fornita dal *framework* di *test* di Django per organizzare e automatizzare verifiche sul comportamento dell'applicazione. Ogni TestCase definisce uno scenario controllato, prepara i dati necessari, esegue una o più operazioni e controlla che il risultato ottenuto sia coerente con quello atteso. 36

WebSocket protocollo di comunicazione che mantiene aperto un canale bidirezionale tra *client* e *server*, permettendo lo scambio di messaggi in tempo reale. 14–16, 19, 22, 25, 27, 32, 34–36, 42, 44, 46, 49, 51, 52