Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Informatica

Marzo 2026

Modernizzazione di sistemi core banking legacy: strategia di migrazione incrementale verso architetture a microservizi event-driven

Relatore: Antonio Vetrò

Candidato: Francesca Villanova

Sommario

Negli ultimi anni, il settore bancario ha affrontato una crescente necessità di modernizzazione dei propri sistemi core, storicamente basati su architetture mainframe che, pur garantendo elevata affidabilità nell'elaborazione transazionale, presentano limiti significativi in termini di scalabilità, integrazione e agilità. Con la diffusione del digital banking e la domanda di servizi accessibili in tempo reale, le architetture monolitiche sincrone, basate su comunicazione request-response e dipendenti dalla scalabilità verticale, risultano inadeguate a sostenere le esigenze operative moderne, evidenziando inefficienze nell'utilizzo delle risorse, accumulo di latenze e costi di manutenzione crescenti.

La presente tesi affronta la modernizzazione di un sistema core banking mediante migrazione verso un'architettura distribuita a microservizi asincroni event-driven, progettati secondo pattern di idempotenza e resilienza. La strategia prevede una migrazione incrementale in cui sistema legacy e sistema modernizzato operano in parallelo, con meccanismi di traffic splitting e rollback, garantendo la continuità operativa e mitigando i rischi legati a interruzioni del servizio e perdita di dati rispetto ad approcci di migrazione in un'unica fase.

La valutazione sperimentale, condotta secondo il paradigma Goal-Question-Metric, confronta le configurazioni as-is e to-be mediante metriche rilevate in produzione, analizzando disponibilità dei servizi e correttezza funzionale attraverso il confronto sistematico degli output. I risultati dimostrano la fattibilità della transizione incrementale e indicano che l'architettura asincrona garantisce prestazioni comparabili o migliorative rispetto alla configurazione sincrona, con benefici evidenti in termini di efficienza, capacità di elaborazione e continuità del servizio.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Il settore bancario ha storicamente fatto affidamento su sistemi mainframe per le funzioni core, adottando architetture monolitiche e linguaggi di programmazione legacy per automatizzare l'elaborazione di grandi volumi transazionali e ridurre i costi operativi. Tali soluzioni hanno garantito elevata affidabilità e capacità di elaborazione per lunghi periodi, ma presentano limiti nell'integrazione con ecosistemi digitali moderni: la struttura monolitica, la dipendenza da modelli di scalabilità prevalentemente verticale e la crescente complessità del codice rendono onerose le attività di manutenzione e poco agili i processi di evoluzione [1].

Con la diffusione del digital banking e l'aumento della domanda di servizi accessibili in tempo reale tramite canali online e mobile, emerge la necessità di rendere i servizi core più integrabili, scalabili e manutenibili. In questo contesto, la modernizzazione punta sull'adozione di architetture scalabili e orientate al lungo periodo, impiegando paradigmi quali il cloud computing, l'architettura a microservizi e le soluzioni di storage distribuito, per adattare dinamicamente le risorse alla domanda, ridurre colli di bottiglia prestazionali e migliorare la resilienza complessiva del sistema [2].

La presente tesi affronta la modernizzazione di servizi core bancari mediante un porting tecnologico da un ambiente mainframe intrinsecamente sincrono basato su COBOL/CICS, a una moderna soluzione a microservizi basata sul paradigma asincrono event-driven, implementato tramite Spring e Apache Kafka. La transizione è finalizzata a garantire livelli di performance e affidabilità compatibili con un dominio ad alta frequenza transazionale, preservando al contempo la continuità operativa. La sostituzione delle funzionalità legacy avviene in modo graduale, introducendo microservizi che rimpiazzano progressivamente parti del sistema esistente senza interromperne l'esercizio, con particolare attenzione al mantenimento di elevate prestazioni e alla stabilità complessiva del sistema. Il lavoro presentato è stato sviluppato a partire dall'esperienza maturata durante un tirocinio curricolare svolto presso Reply.

1.1 Obiettivi

I percorsi di modernizzazione verso microservizi comprendono sfide concrete, tra cui la complessità della migrazione incrementale, la gestione delle dipendenze e l'adozione di pratiche di osservabilità e validazione in ambienti reali. Gli obiettivi specifici della tesi sono:

1. Esaminare le scelte architetturali e le soluzioni tecniche adottate per la transizione da un sistema sincrono legacy a un'architettura a microservizi asincroni, progettati per garantire scalabilità, resilienza e alte prestazioni.
2. Definire una strategia di migrazione incrementale e controllata, basata su coesistenza, meccanismi di traffic splitting e rollback, con l'obiettivo di garantire continuità operativa e riduzione del rischio rispetto a transizioni *big-bang*.
3. Progettare i servizi critici considerando requisiti di idempotenza e pattern di resilienza, al fine di ridurre rischi di inconsistenza e duplicazione dei dati in presenza di retry/riconsegne tipiche dei flussi asincroni.
4. Valutare sperimentalmente l'impatto dell'intervento, confrontando le due configurazioni (as-is sincrona vs to-be asincrona) mediante metriche selezionate rilevate in produzione durante una fase di coesistenza controllata.

La valutazione adottata ha carattere empirico-sperimentale: il confronto tra le due configurazioni viene effettuato su servizi funzionalmente equivalenti (as-is sincrona e to-be asincrona) esposte in coesistenza controllata, dove il vecchio sistema e il nuovo sistema operano in parallelo per un periodo transitorio. Le misure sono state definite secondo il paradigma Goal-Question-Metric (GQM), che collega obiettivi di valutazione a domande operative e a metriche misurabili. In questo contesto, la validazione durante la transizione non riguarda solo la disponibilità del servizio, ma anche la correttezza dei risultati prodotti: il confronto sistematico tra output di sistemi differenti rappresenta un approccio razionale per individuare discrepanze e ridurre il rischio di errori.

1.2 Struttura della tesi

La tesi è organizzata come segue:

- **Capitolo 1 – Introduzione:** inquadra il problema della modernizzazione dei sistemi core bancari, definisce obiettivi e perimetro del lavoro.
- **Capitolo 2 – Contesto as-is:** descrive l'architettura legacy (mainframe COBOL/CICS), il modello sincrono, gli aspetti transazionali e i principali vincoli che motivano la migrazione.

- **Capitolo 3 – Design to-be:** presenta l'architettura target a microservizi asincroni event-driven, includendo la rappresentazione con modello C4 e le scelte tecnologiche principali.
- **Capitolo 4 – Implementazione e coesistenza:** illustra la strategia di migrazione incrementale e la coesistenza legacy/to-be, descrivendo i meccanismi operativi adottati per ridurre il rischio e garantire continuità di servizio.
- **Capitolo 5 – Valutazione empirico-sperimentale:** definisce l'impostazione sperimentale e la valutazione empirica basata su GQM, confrontando le configurazioni as-is e to-be mediante metriche raccolte durante la fase di coesistenza.
- **Capitolo 6 – Limitazioni e possibili miglioramenti futuri:** discute i principali limiti dell'approccio e del contesto sperimentale e propone evoluzioni e interventi migliorativi.
- **Capitolo 7 – Conclusioni:** presenta i risultati principali e i contributi della tesi.

Capitolo 2

Contesto “as-is” di riferimento: core bancari legacy e transazioni sincrone

Nel contesto bancario contemporaneo, i servizi mainframe, comunemente definiti come “sistemi legacy”, svolgono un ruolo centrale nella gestione dei processi core dell’organizzazione, supportando le principali funzioni di elaborazione transazionale. Essi sono caratterizzati da una struttura monolitica, adatta a gestire carichi di lavoro molto elevati, ma sono generalmente considerati sistemi ad alta criticità, poiché un loro malfunzionamento può avere impatti significativi sulla continuità dei servizi bancari. Tali sistemi sono stati progettati in un periodo storico in cui l’integrazione con applicazioni esterne non rappresentava un requisito fondamentale; questa impostazione ha contribuito, nel tempo, all’evoluzione di un codice sorgente mal strutturato, rendendo le attività di manutenzione particolarmente onerose in termini di costi e complessità operativa [1].

Il progetto oggetto di questa tesi affronta la modernizzazione di un sistema core banking legacy, il cui stato as-is è basato su un’architettura mainframe con linguaggio di programmazione COBOL e modello di comunicazione sincrono. Al giorno d’oggi, infatti, l’evoluzione del settore bancario ha reso prioritaria l’integrazione dei sistemi core con servizi terzi. In tale contesto, un’architettura legacy come quella presa in esame rappresenta un collo di bottiglia significativo, poiché le interazioni sincrone tra i componenti introducono latenze accumulative, impattando negativamente le prestazioni complessive del sistema.

2.1 COBOL e infrastruttura mainframe nel core banking

Alla base dei sistemi legacy presi in analisi in questa tesi si trova COBOL, acronimo di *Common Business Oriented Language*, un linguaggio di programmazione ad alto livello sviluppato alla fine degli anni '50 specificamente per applicazioni aziendali. COBOL si distingue per l'efficienza nella gestione di grandi volumi di dati e transazioni, risultando particolarmente appropriato in settori che richiedono elevata affidabilità e rapidità di elaborazione, caratteristiche fondamentali nel settore bancario [3].

COBOL utilizza una sintassi verbosa basata sulla lingua inglese che, al contrario dei linguaggi macchina, lo rende di facile comprensione e ne permette la manutenzione a posteriori, aspetto rilevante per infrastrutture destinate a operare per lunghi lassi di tempo. Inoltre, privilegia le specifiche esigenze aziendali, risultando efficiente nella gestione di insiemi di dati di grandi dimensioni e supportando l'aritmetica decimale, che consente l'esecuzione di calcoli finanziari con elevata precisione, altra proprietà molto importante in ambito bancario [4], [5].

La robustezza e l'affidabilità del linguaggio COBOL sono strettamente legate all'infrastruttura mainframe su cui esso opera. Il mainframe IBM z/OS fornisce ridondanza, tolleranza ai guasti ed elevata disponibilità, anche per carichi di lavoro ad alta criticità operativa. Questa sinergia tra il linguaggio e la piattaforma mainframe ha reso COBOL dominante in ambiti critici come quello bancario, assicurativo e governativo, in cui l'indisponibilità o la compromissione dei dati e delle operazioni potrebbero avere ripercussioni significative [6], [7].

2.2 Elaborazione transazionale a modello sincrono

2.2.1 Pattern request/response e call chain

Nel servizio mainframe in esame, l'interazione tra componenti software segue un paradigma di comunicazione sincrono basato sul pattern request/response.

In questo paradigma, quando un componente (client) richiede un servizio a un altro componente (server), il client sospende la propria esecuzione e resta in attesa di una risposta da parte del server prima di proseguire, garantendo una sincronizzazione temporale rigorosa, idonea per operazioni critiche come transazioni bancarie dove la conferma del completamento è essenziale [8].

Si può evidenziare la natura bloccante del pattern request/response attraverso il seguente scenario tipico:

1. Un cliente accede all'applicazione di online banking.
2. L'applicazione invia al servizio di core banking una richiesta (sincrona) di consultazione del saldo per il conto del cliente.
3. La richiesta viene ricevuta dal sistema e processata.
4. Il programma COBOL esegue una query sul database per recuperare il saldo.
5. Il servizio mainframe ritorna la risposta all'applicazione cliente.
6. Solo a questo punto, l'applicazione di online banking può presentare il saldo all'utente.

Questo approccio causa una latenza di comunicazione non trascurabile, in particolare per quanto riguarda i sistemi transazionali, nei quali le operazioni devono concludersi con un riscontro (successo o fallimento) in tempi compatibili con l'operatività del servizio [8].

Un aspetto particolarmente problematico emerge quando il pattern request/response si ripete attraverso molteplici livelli di componenti, formando una catena di chiamate (*call chain*). In tale catena, le interazioni hanno una forte dipendenza temporale, per cui la disponibilità complessiva del sistema è vincolata da quella di ogni singolo anello della catena: per esempio, se il servizio A attende il servizio B, che a sua volta attende il servizio C, l'esecuzione rimane bloccata finché l'ultimo nodo non risponde. Generalmente, una call chain può estendersi su diversi moduli del sistema, ciascuno dei quali esegue operazioni e genera latenza, oltre alla latenza di comunicazione tra i moduli. Di conseguenza, il tempo di risposta percepito dal client corrisponde alla somma dei tempi di elaborazione dei singoli componenti e delle latenze di rete tra di essi. L'accumulo di latenze lungo una call chain di profondità significativa può determinare un tempo di risposta totale elevato, soprattutto in caso di carico elevato [8], [9].

Inoltre, un ritardo o un guasto in un servizio a valle può generare dei fallimenti che si propagano a cascata nel sistema, poiché la disponibilità complessiva di una call chain è governata dal principio della "catena debole" (*weakest link*): se uno qualsiasi dei componenti diviene indisponibile a causa di un fallimento, di un timeout o di un sovraccarico, l'intera catena fallisce, causando un errore percepito dal client finale [10], [8].

2.2.2 Middleware transazionale CICS e z/OS Connect per architettura sincrona

Nell'ambito dei sistemi mainframe dedicati all'elaborazione transazionale ad alta frequenza, tipici del settore bancario e finanziario, il software utilizzato prevalentemente per l'orchestrazione delle transazioni è CICS (*Customer Information Control*

System). Esso è stato sviluppato da IBM e si occupa di coordinare l'esecuzione di programmi applicativi COBOL, agendo come middleware tra i sistemi esterni e l'architettura mainframe [10].

Nel sistema core banking oggetto di questa tesi, una transazione ha inizio quando una richiesta esterna viene inviata al sistema mainframe.

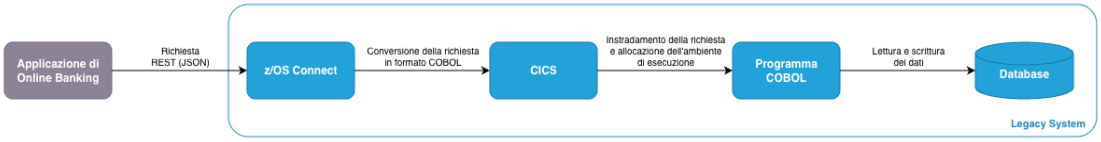

Figura 2.1: Flusso di elaborazione di una richiesta nel sistema legacy.

Le richieste provenienti da applicazioni o sistemi esterni vengono ricevute da z/OS Connect Enterprise Edition, una soluzione IBM utilizzata per esporre e invocare servizi CICS tramite interfacce API, fungendo da punto di integrazione tra client moderni e programmi CICS/COBOL [10], [11]. z/OS Connect si occupa di tradurre le richieste in formato REST, tipicamente espresse in JSON, in un formato comprensibile alla logica CICS/COBOL sul mainframe, identificando il *transaction code* appropriato e predisponendo i dati necessari all'esecuzione della transazione. Il *transaction code* è un identificativo univoco utilizzato per determinare il programma COBOL da eseguire e consente di instradare la richiesta verso il componente applicativo corretto.

Una volta identificato il programma COBOL, CICS alloca uno spazio di esecuzione isolato e dedicato per quella specifica transazione. Tale ambiente comprende la memoria privata (*working storage*) dove il programma COBOL memorizza variabili e dati temporanei, un'area di comunicazione (COMMAREA) dove vengono depositati i dati della richiesta e scritti i risultati dell'elaborazione, nonché le connessioni alle risorse critiche del sistema, come database e file. Il mantenimento dell'isolamento transazionale è essenziale per evitare interferenze tra richieste concorrenti [12]. Al termine dell'elaborazione, z/OS Connect trasforma la risposta prodotta dal mainframe in formato JSON e la restituisce al client esterno [11].

L'utilizzo di z/OS Connect introduce un vantaggio significativo in una fase di transizione, in quanto consente di esporre i servizi core bancari verso canali moderni senza modificare la logica COBOL sottostante. Questo consente di allinearsi agli standard di integrazione moderni, evitando la riscrittura immediata dei programmi [10], [13].

Un aspetto critico di CICS è mantenere l'integrità dei dati quando molteplici transazioni accedono simultaneamente alle stesse informazioni. Per fare questo, CICS implementa un meccanismo di *concurrency control* basato sul protocollo *two-phase locking* (2PL) [14], [15]. In questo paradigma, quando una transazione vuole accedere a un dato critico (in lettura o scrittura), il sistema acquisisce un

lock esclusivo sulla risorsa interessata, impedendo ad altre transazioni di effettuare operazioni incompatibili sullo stesso dato. Qualora una transazione richieda l'accesso a una risorsa già protetta da un lock in conflitto, la richiesta viene sospesa fino al rilascio del lock da parte dell'altra transazione. Il 2PL garantisce la correttezza dell'esecuzione, prevenendo anomalie di concorrenza (*race conditions*), ma introduce inevitabilmente latenze [14].

Sebbene z/OS Connect consenta di esporre i servizi mainframe con interfacce moderne, non risolve i problemi intrinseci del modello di esecuzione sincrono sottostante. Le comunicazioni rimangono di tipo bloccante e le criticità legate all'accumulo di latenza nelle call chain, all'inefficienza nell'utilizzo delle risorse e alla contesa dei lock, che può portare all'accumulo significativo di code di attesa, persistono. Proprio queste limitazioni motivano l'evoluzione verso un'architettura a microservizi asincroni [10].

2.2.3 Modello transazionale: Unit of Work, syncpoint e semantica ACID

Nel contesto della progettazione transazionale, una *Unit of Work* (UOW) definisce il perimetro logico di una transazione, ossia l'insieme di operazioni da effettuare sui dati che devono essere considerate come un'unica unità indivisibile dal punto di vista della consistenza [16]. In termini operativi, una UOW raggruppa una sequenza di operazioni che devono completarsi con successo e le cui modifiche vengono rese permanenti nel database, oppure, in caso di fallimento, non producono alcun effetto, secondo il principio di "all-or-nothing" alla base dell'atomicità delle transazioni, particolarmente importante in ambito bancario [16], [17].

L'implementazione concreta di una UOW in CICS avviene attraverso il meccanismo di *syncpoint*, che identifica un istante temporale durante l'esecuzione di una transazione in cui tutte le modifiche effettuate ai dati vengono consolidate in modo permanente, oppure annullate [16]. Tale comportamento viene realizzato tramite due operazioni fondamentali: il `COMMIT`, che conferma le modifiche effettuate nella UOW rendendole visibili e durevoli, e il `ROLLBACK`, che annulla le modifiche riportando i dati allo stato precedente l'inizio della UOW [16], [17]. Un syncpoint rappresenta perciò un punto di non ritorno dal quale le modifiche non possono più essere revocate mediante rollback (fatta eccezione per transazioni future).

L'architettura transazionale di CICS garantisce il rispetto delle proprietà ACID, che assicurano l'atomicità, la consistenza, l'isolamento e la durabilità delle transazioni, principi fondamentali per la correttezza della loro esecuzione [18].

Atomicity (Atomicità): Una UOW è eseguita in modo atomico, il che significa che tutte le operazioni sono applicate oppure nessuna di esse, evitando stati intermedi incoerenti [18]. CICS garantisce questa proprietà tramite un sistema

di *journaling* (recovery log), che consente al database di determinare, anche in caso di failure immediatamente prima del `COMMIT`, se la UOW sia stata committata e debba essere ripristinata, oppure se debba essere annullata [16].

Consistency (Coerenza): Ogni transazione mantiene il database in uno stato coerente, ossia assicura che tutte le regole di integrità dei dati siano rispettate, portando il database da uno stato valido a un altro stato valido [18].

Isolation (Isolamento): Nel modello di CICS, l'isolamento tra transazioni concorrenti è garantito mediante il protocollo di *concurrency control two-phase locking* (2PL) descritto precedentemente [14]. Questo meccanismo assicura che l'esecuzione simultanea di più transazioni produca risultati equivalenti a quelli che si otterrebbero eseguendo le transazioni in sequenza.

Durability (Durabilità): Una volta che una transazione raggiunge il syncpoint di commit, viene garantito che le modifiche siano permanentemente memorizzate su un supporto duraturo e che sopravvivano a qualsiasi guasto del sistema [18]. CICS implementa questa proprietà attraverso un sistema di journaling strutturato, in cui ogni modifica è scritta in un log di recovery prima del completamento del commit. In caso di guasti, durante la fase di recovery CICS utilizza tale journal per ripristinare le transazioni completate, evitando la perdita di qualsiasi movimento già committato [16].

Nel caso di transazioni distribuite che coinvolgono molteplici *resource manager*, CICS coordina il commit distribuito utilizzando il protocollo *two-phase commit* (2PC) [15], [19]. Il 2PC è un algoritmo di coordinamento che assicura che tutte le risorse partecipanti raggiungano un consenso sul commit prima che le modifiche diventino permanenti, prevenendo inconsistenze distribuite. Durante la prima fase, il coordinatore transazionale contatta tutti i resource manager coinvolti per verificare che le operazioni possano essere completate correttamente, senza applicare ancora le modifiche. Nella seconda fase, se tutti i partecipanti confermano la disponibilità, il coordinatore invia il comando definitivo di commit, rendendo le modifiche permanenti; in caso di fallimento di uno dei partecipanti, il coordinatore ordina il rollback a tutti gli altri [15], [19]. Il 2PC garantisce così l'atomicità delle transazioni che interessano più risorse, sebbene introduca una latenza aggiuntiva dovuta al coordinamento tra le entità coinvolte [16].

2.3 Batch processing

Oltre alle transazioni sincrone in real-time gestite da CICS, i sistemi mainframe supportano un'importante categoria di elaborazione denominata *batch processing*. Esso è un modello di esecuzione in cui un insieme di richieste viene raccolto ed

elaborato in un'unica sessione, tipicamente durante finestre temporali predefinite in cui il carico transazionale è ridotto [20], [21].

Nei sistemi mainframe IBM z/OS, i job batch vengono specificati e schedulati utilizzando il linguaggio procedurale JCL (*Job Control Language*), che definisce la sequenza di operazioni da eseguire. Un job batch, infatti, può contenere molteplici step, ciascuno eseguito in sequenza, con la possibilità di condizionare l'esecuzione di step successivi in base al risultato di quelli precedenti [21].

In particolare, il batch processing è adatto a operazioni di volume elevato che non richiedono interattività. Tale modello offre un vantaggio in termini di efficienza, poiché i job vengono schedulati in momenti in cui le risorse del mainframe sono maggiormente disponibili e l'assenza di interattività consente di ottenere un throughput superiore rispetto al processing transazionale real-time [20].

Tuttavia, la natura del modello batch implica una latenza intrinseca: i dati elaborati in batch non sono disponibili fino al suo completamento, rendendo impossibile vedere real-time i risultati. Nel core banking moderno, dove le aspettative dei clienti includono accesso real-time ai dati e visibilità istantanea dei movimenti, il modello batch rappresenta un collo di bottiglia sempre più rilevante.

2.4 Limiti e vincoli operativi

La solidità strutturale che caratterizza i sistemi basati su COBOL ha contribuito nel tempo alla generazione di architetture fortemente centralizzate e poco inclini al cambiamento. La dipendenza dalle proprietà del linguaggio e dall'ambiente mainframe ha infatti scoraggiato l'attuazione di investimenti in modernizzazione, dando vita a sistemi che, pur garantendo elevati livelli di stabilità e affidabilità, risultano oggi limitati in termini di agilità e capacità di integrazione con tecnologie esterne [10]. Le limitazioni del sistema legacy, parzialmente descritte nelle sezioni precedenti, si articolano su più livelli.

2.4.1 Vincoli linguistici e architetturali

La rigidità strutturale di COBOL rappresenta uno dei principali limiti del linguaggio. Esso è stato progettato per l'elaborazione di flussi di dati lineari e sequenziali, con una struttura di programma monolitica, che rende difficile isolare componenti logicamente distinte e riutilizzarle come servizi modulari e indipendenti. Inoltre, la struttura stessa del linguaggio contribuisce ad aumentare la probabilità di errori di programmazione e a rendere più onerose le attività di manutenzione. I sistemi COBOL tendono, nel tempo, ad accumulare basi di codice sorgente altamente complesse, spesso scarsamente documentate e con una struttura architetturale degradata nel tempo. Questo fenomeno è comunemente indicato come *spaghetti code*, ovvero codice disorganizzato caratterizzato da flussi di controllo intrecciati,

privo di una struttura logica chiara, in cui è complesso comprendere, modificare o isolare singole funzionalità. La manutenzione di tali sistemi diventa quindi sempre più onerosa, con un impatto diretto sui costi operativi e sulla capacità di evoluzione del sistema [22].

2.4.2 Limitazioni del modello sincrono e dell'architettura CICS

Il modello di esecuzione sincrono adottato da CICS introduce una serie di vincoli che si amplificano con l'aumento della complessità dei servizi e del grado di concorrenza del sistema. In particolare, l'utilizzo del pattern request–response bloccante comporta un accumulo di latenza che diventa critico quando le transazioni si articolano lungo call chain profonde, in cui ogni componente introduce tempi di elaborazione e di attesa aggiuntivi. Analogamente, quando molteplici transazioni concorrenti accedono alle medesime risorse di dati, i meccanismi di locking possono causare un accumulo significativo di code di attesa, con un impatto diretto sul throughput complessivo del sistema [14], [9].

Sebbene z/OS Connect consenta di esporre i servizi mainframe tramite interfacce REST conformi agli standard di integrazione moderni, esso agisce esclusivamente come strato di integrazione e non risolve i problemi intrinseci del modello di esecuzione sincrono sottostante [10].

Un'ulteriore limitazione deriva dall'isolamento transazionale: per garantirlo, a ogni transazione attiva viene assegnato uno spazio di esecuzione dedicato. Tuttavia, questo modello di allocazione delle risorse risulta inefficiente in scenari ad elevata concorrenza, dove numerosi task COBOL rimangono sospesi in attesa del completamento di operazioni di I/O, continuando a consumare risorse [12].

2.4.3 Scalabilità e infrastruttura

La natura monolitica e sincrona dei programmi COBOL, particolarmente in ambito CICS, limita la possibilità di distribuire i componenti su più nodi o di sfruttare architetture cloud native, poiché il linguaggio e l'ambiente di esecuzione non offrono primitive di alto livello per parallelismo, concorrenza asincrona o calcolo distribuito. Nel sistema legacy analizzato, l'aumento del carico transazionale viene gestito principalmente attraverso il potenziamento dell'hardware mainframe (*scaling verticale*), strategia economicamente meno sostenibile rispetto ai modelli di *scaling orizzontale*, che consentono di distribuire il carico tra più nodi e aumentare la resilienza del sistema, ovvero la capacità di continuare a funzionare correttamente anche in caso di guasti o picchi di carico [8], [9].

Inoltre, l'integrazione di tutte le funzionalità in un unico sistema genera dipendenze temporali rigide tra i componenti: anche una piccola modifica a un servizio

può richiedere la ricompilazione e il redeployment dell'intero sistema, limitando la flessibilità operativa e la rapidità degli aggiornamenti [23].

2.4.4 Limitazioni di integrazione e incompatibilità con standard moderni

L'esposizione di servizi COBOL verso client moderni comporta vincoli operativi significativi. La struttura rigida dei dati mainframe, basata su record a lunghezza fissa, è incompatibile con formati moderni semi-strutturati come JSON e XML, di conseguenza sono necessari intermediari che si occupano della conversione dei dati. Questi layer introducono overhead di elaborazione, complessità aggiuntiva e potenziali punti di failure, poiché il mainframe dipende da essi per comunicare con i sistemi esterni [10]. Strumenti come z/OS Connect consentono di esporre i servizi CICS come API REST, ma agiscono esclusivamente come strato di integrazione e traduzione, senza eliminare vincoli imposti da COBOL [10], [11].

2.4.5 Vincoli operativi e di costo

I vincoli tecnologici evidenziati nei paragrafi precedenti si riflettono direttamente sui vincoli operativi e sui costi dei sistemi legacy. In particolare, il modello batch introduce rigidità nei processi e aumenta la latenza delle elaborazioni, limitando la capacità del sistema di rispondere rapidamente alle richieste. Tali caratteristiche risultano incompatibili con le aspettative dei clienti moderni, che richiedono disponibilità continua ed elaborazioni in tempo reale.

Dal punto di vista economico, l'utilizzo dei mainframe comporta costi operativi significativi, legati sia all'hardware sia alla manutenzione del software. Le architetture moderne basate su piattaforme cloud-native offrono invece modelli di costo più flessibili, grazie alla possibilità di scalare risorse in modo dinamico e distribuire il carico tra più nodi, riducendo significativamente le spese operative a parità di capacità di elaborazione [24].

2.5 Raccordo verso l'architettura to-be

Le limitazioni evidenziate nella Sezione 2.4 motivano la ricerca di un'architettura alternativa in grado di affrontare specificamente questi vincoli, mantenendo al contempo i vantaggi di affidabilità e robustezza che i sistemi mainframe hanno consolidato nel tempo.

L'architettura proposta, trattata nel capitolo successivo, si articola su tre pilastri: l'adozione di un modello di esecuzione event-driven asincrono, la decomposizione dell'architettura monolitica in microservizi indipendenti e la migrazione da COBOL

a linguaggi di programmazione moderni che supportano nativamente i pattern necessari per architetture cloud-native.

Queste scelte, sebbene rappresentino un cambiamento paradigmatico significativo rispetto al sistema legacy, sono motivate dalla necessità di soddisfare requisiti non-funzionali quali performance, scalabilità e manutenibilità, che il sistema legacy non è più in grado di garantire in maniera sostenibile.

Capitolo 3

Design “to-be”: architettura a microservizi asincroni

La migrazione del sistema di core banking da un’architettura legacy sincrona a una moderna architettura a microservizi rappresenta una trasformazione significativa sia dal punto di vista tecnologico sia organizzativo. Questa trasformazione coinvolge diversi aspetti dell’architettura, dal modello di comunicazione tra i componenti fino alle tecnologie di sviluppo utilizzate.

In particolare, il passaggio da un modello request–response sincrono, tipico degli ambienti CICS, a un paradigma asincrono event-driven basato su Apache Kafka permette di eliminare le latenze bloccanti e di favorire il disaccoppiamento tra i componenti applicativi. L’adozione di un sistema di messaggistica ad alto throughput (Kafka), cioè in grado di gestire flussi di eventi ad elevata intensità, abilita l’elaborazione asincrona delle transazioni e supporta scenari a elevata concorrenza, rendendo l’architettura intrinsecamente più scalabile e resiliente [25], [26].

Parallelamente, la decomposizione dell’architettura monolitica in microservizi indipendenti consente di ridurre il forte accoppiamento tipico delle architetture legacy, permettendo a ciascun servizio di essere sviluppato, distribuito e scalato in modo autonomo. Questo approccio consente di ridurre i tempi di sviluppo e riduce i tempi di rilascio di nuove funzionalità [23], [26].

Infine, la migrazione tecnologica include il superamento del linguaggio COBOL a favore di linguaggi e framework moderni, come Java con Spring Boot, che offrono accesso a librerie consolidate, strumenti di sviluppo avanzati e supporto nativo a paradigmi fondamentali per le architetture cloud-native, come programmazione reattiva, asincronia e gestione avanzata della concorrenza [27].

3.1 Modello C4 e struttura architetturale

Le decisioni architetturali relative al sistema to-be sono rappresentate tramite diagrammi C4, che offrono una visione gerarchica e organizzata del sistema su quattro livelli di astrazione, ciascuno con una prospettiva specifica.

Il C4 Model consente di partire da una visione di alto livello del sistema e, attraverso livelli progressivamente più dettagliati, di descrivere la struttura e le responsabilità dei suoi elementi costitutivi, fino al livello del codice (System Context → Container → Component → Code) [28].

Livello 1 Il *System Context Diagram* mostra il sistema software in esame, gli attori che interagiscono con esso (persone o sistemi esterni) e le relazioni esterne. A questo livello, i dettagli interni dell'architettura non vengono rappresentati: l'obiettivo è identificare chiaramente i confini del sistema e le sue interazioni con l'ambiente esterno. Questo diagramma è progettato per comunicare con stakeholder come executive e product manager, fornendo informazioni su cosa fa il sistema e chi lo utilizza [28].

Livello 2 Il *Container Diagram* decompone il sistema in container, ossia entità logiche o esecutive autonome che eseguono una specifica funzione all'interno del sistema. Ogni container ha responsabilità specifiche, comunica con altri container tramite interfacce o protocolli definiti e mantiene il proprio storage persistente. Questo livello è utile principalmente ad architetti di sistema e sviluppatori senior, in quanto fornisce una visione macroscopica della struttura del sistema senza scendere nei dettagli dei singoli componenti [28].

Livello 3 Il *Component Diagram* decompone container selezionati nei loro componenti costituenti, cioè moduli o elementi funzionali con responsabilità definite. Questo livello mostra le interazioni interne tra componenti e serve come guida per gli sviluppatori, indicando linee guida per lo sviluppo del codice e quali interfacce usare per la comunicazione interna [28].

Livello 4 Il livello *Code* descrive le classi, le interfacce e le relazioni dettagliate dei singoli componenti. Nel caso specifico del sistema aziendale in esame, questo livello non viene rappresentato, in quanto la trattazione si concentra sull'architettura ad alto livello e sulla decomposizione in container e componenti, ritenendo sufficiente fornire indicazioni implementative generali senza scendere nei dettagli del codice sorgente [28].

In conformità alle linee guida del C4 Model di Simon Brown, nei diagrammi è stata adottata una codifica cromatica volta a distinguere gli elementi oggetto dell'analisi da quelli esterni al perimetro del progetto. In particolare, gli elementi rappresentati in blu costituiscono il sistema in esame e sono approfonditi nei

diversi livelli di dettaglio, mentre gli elementi in grigio rappresentano sistemi o componenti preesistenti, utilizzati dal sistema ma non oggetto di progettazione o sviluppo nell'ambito del presente lavoro. Tale distinzione consente di rendere immediatamente visibili i confini architetturali e le responsabilità analizzate.

3.1.1 System Context Diagram

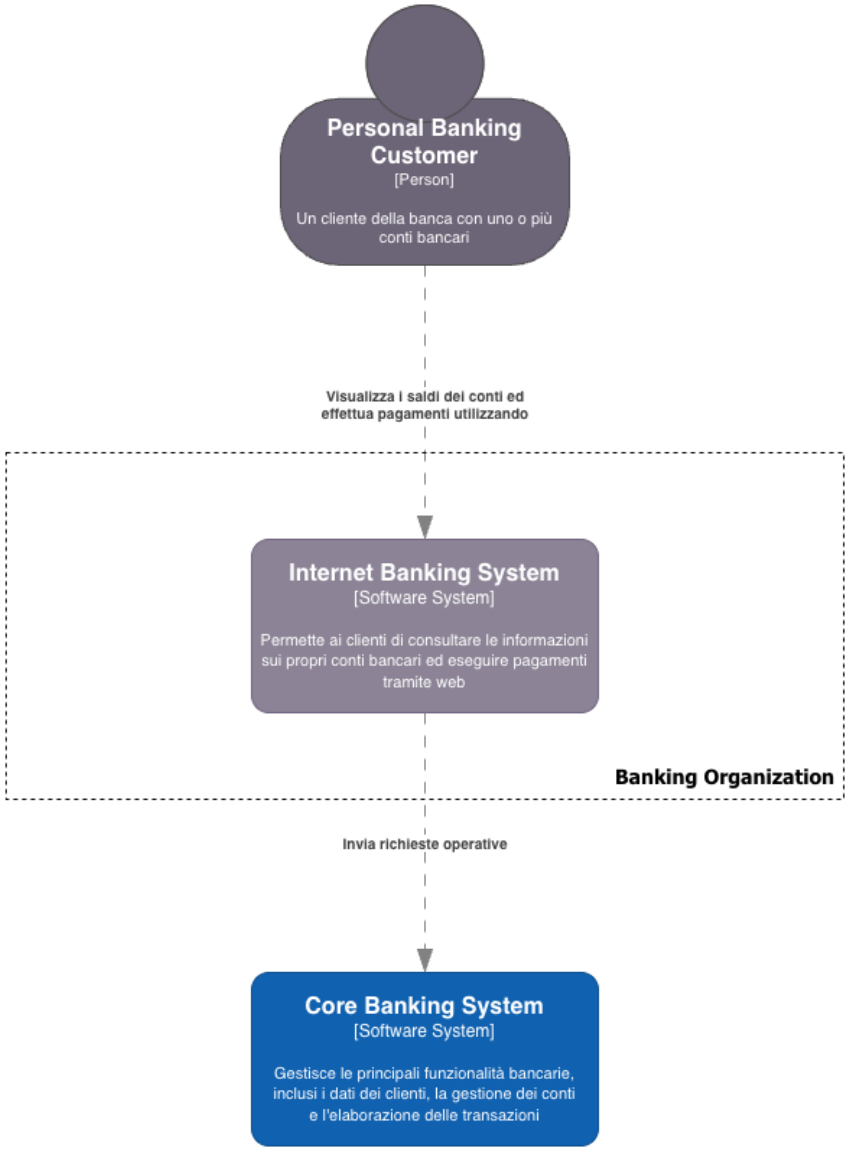

Figura 3.1: Relazioni tra il sistema core banking e gli attori esterni.

Il System Context Diagram fornisce una visione di alto livello del sistema di Internet Banking, evidenziandone i confini e le principali interazioni con gli attori esterni. In questo contesto, il sistema consente ai clienti della banca (*Personal Banking Customer*) di consultare le informazioni relative ai propri conti correnti ed eseguire operazioni dispositive, come i pagamenti, tramite interfaccia web. Il sistema di Internet Banking interagisce direttamente con il Core Banking System, che rappresenta il sistema centrale responsabile della gestione dei dati dei clienti, dei conti e dell'elaborazione delle transazioni bancarie. Utilizzando la notazione cromatica del C4 Model per evidenziare il perimetro di analisi, lo studio architetturale presentato in questa tesi si concentra sul Core Banking System e sulle sue interazioni con il sistema di Internet Banking.

3.1.2 Container Diagram

Il Container Diagram illustra la struttura interna complessiva dell'architettura, mostrando l'apertura sia dell'Internet Banking System sia del Core Banking System, e descrivendo le modalità di interazione tra i rispettivi container applicativi.

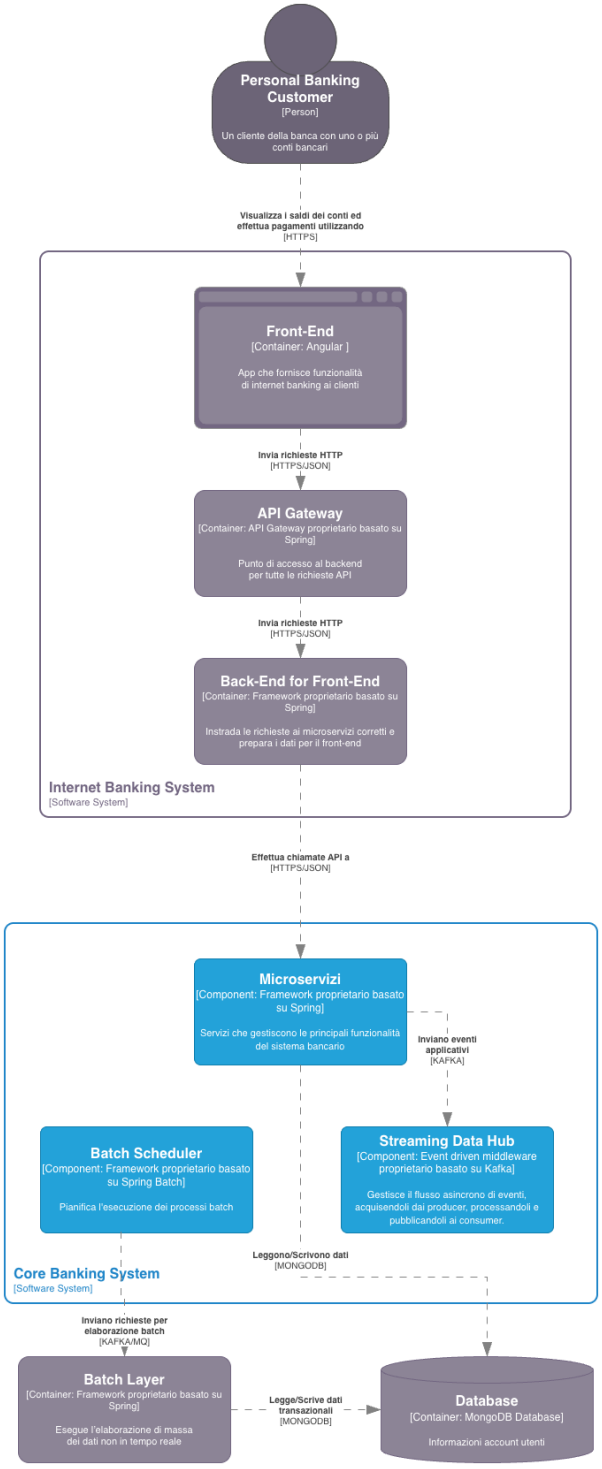

Figura 3.2: Decomposizione architetturale dei container logici principali dell'Internet Banking System e del Core Banking System.

Per quanto riguarda l'Internet Banking System, l'interazione dell'utente avviene tramite un'interfaccia web Front-End sviluppata in Angular, che invia richieste HTTP al backend attraverso un API Gateway. Quest'ultimo rappresenta l'unico punto di accesso al backend per tutti i client ed è responsabile dell'instradamento delle richieste verso i servizi appropriati, garantendo uniformità di accesso e disaccoppiamento tra client e servizi interni.

Le richieste vengono quindi gestite da un *Back-End for Front-End* (BE4FE), ovvero un backend applicativo progettato per servire un determinato client o piattaforma (ad esempio web o mobile), che funge da livello di intermediazione tra il front-end e il Core Banking System. Esso non introduce una molteplicità di backend, ma funge da strato di adattamento tra client eterogenei e un unico backend centrale.

Il BE4FE interroga un *Session Manager* per ottenere le informazioni di sessione dell'utente; quest'ultimo recupera i dati dalla propria cache e li restituisce sotto forma di session context. Tali informazioni vengono utilizzate dal BE4FE per filtrare le richieste dirette al core e per formattare correttamente i parametri di input e output.

A seguito dell'elaborazione da parte del Core Banking System, le risposte vengono ritrasmesse al BE4FE, che provvede infine a inoltrarle al front-end per la presentazione all'utente.

Il BE4FE è concepito come un backend *stateful*, in quanto gestisce il mantenimento dello stato tra operazioni successive (ad esempio autenticazione, compilazione di form o avanzamento di transazioni), mentre il Core Banking System consiste in un backend di tipo *stateless*, favorendo una maggiore scalabilità dei servizi in quanto non è necessario che mantenga informazioni tra richieste successive.

Il Core Banking System è composto da un insieme di microservizi, ciascuno responsabile di uno specifico dominio funzionale. La comunicazione tra i microservizi può avvenire secondo due modalità principali: sincrona o asincrona. La comunicazione sincrona avviene tramite un REST Connector, basato su chiamate HTTP con risposta immediata, ed è utilizzata nei casi in cui sia necessario ottenere un risultato in tempo reale, ad esempio per la validazione di una transazione. La comunicazione asincrona, invece, è basata su Apache Kafka e viene adottata per operazioni che non richiedono una risposta immediata, come la propagazione di eventi, l'aggiornamento di sistemi secondari o l'attivazione di elaborazioni successive non sincrone, che vengono eseguite indipendentemente dal flusso principale di richiesta, consentendo un maggiore disaccoppiamento tra i servizi.

All'interno del Core Banking System opera lo *Streaming Data Hub* (SDH), un middleware event-driven basato su Kafka, responsabile della raccolta, elaborazione e distribuzione degli eventi applicativi. Lo SDH riceve dati da diversi producer, li filtra e li rende disponibili ai consumer in modalità near real-time, ovvero con una latenza ridotta ma non istantanea, adeguata a scenari di elevata concorrenza e

grandi volumi di dati.

Completano l'architettura alcuni elementi esterni ai due sistemi principali. Il *Batch Layer* è dedicato all'elaborazione massiva dei dati non in tempo reale e consente di raggruppare e processare in modalità batch richieste quali aggiornamenti periodici o calcoli aggregati. La persistenza dei dati è gestita tramite un unico database (MongoDB), utilizzato dai microservizi per le operazioni di consultazione e di aggiornamento. Dal punto di vista infrastrutturale, il database è distribuito su più nodi/server per garantire alta disponibilità e tolleranza ai guasti, mantenendo tuttavia un unico datastore logico per l'applicazione.

3.1.3 Component Diagram

Il Component Diagram fornisce una visione dettagliata della struttura interna di un microservizio, evidenziando una struttura in layer che separa le responsabilità. Il punto di ingresso del microservizio è rappresentato dal *Controller*, che riceve le richieste in arrivo, valida i dati e le inoltra ai componenti sottostanti. Esso può occuparsi anche di funzionalità trasversali come autenticazione, autorizzazione e gestione degli errori.

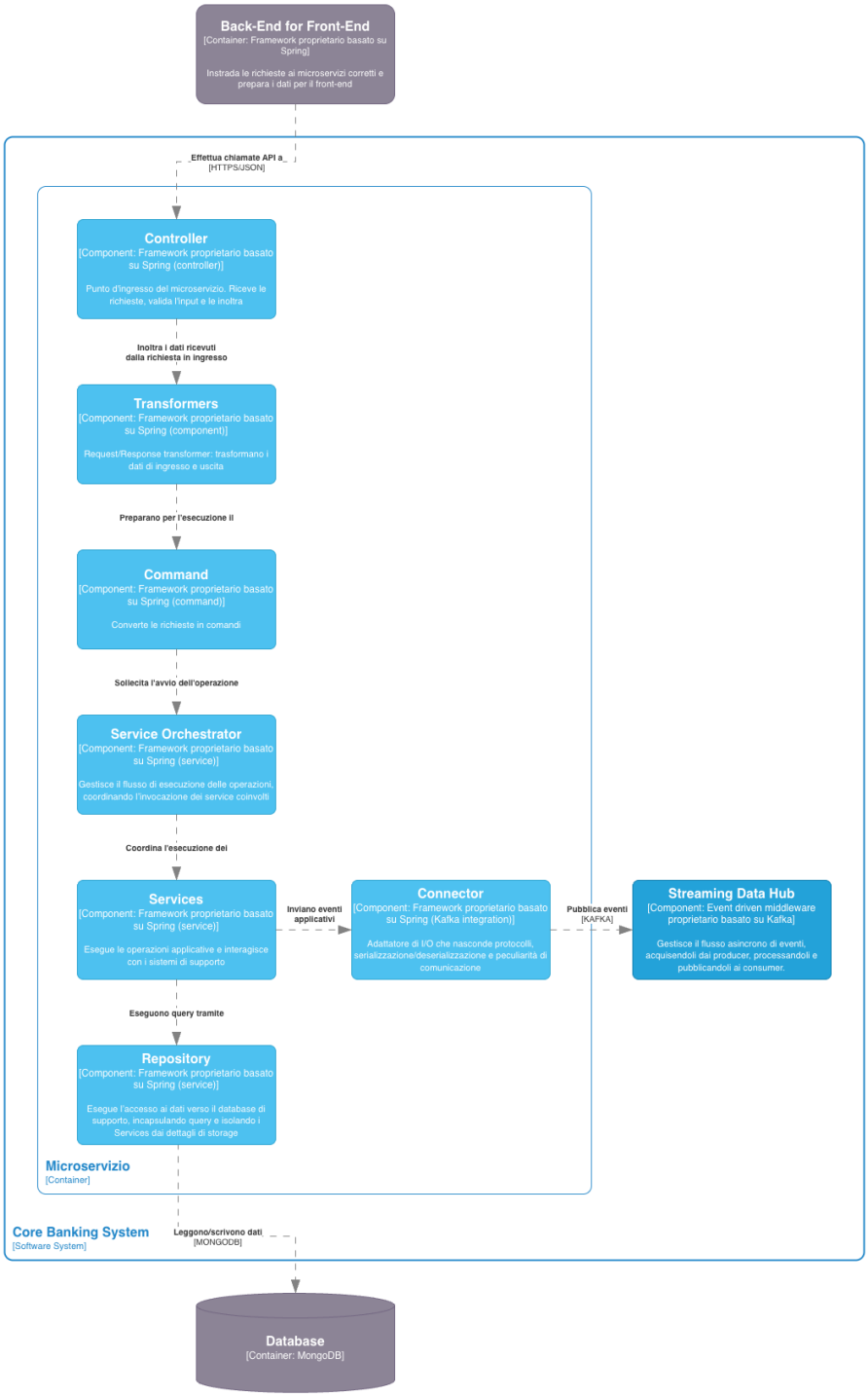

Figura 3.3: Decomposizione in componenti di un microservizio del Core Banking System.

I dati elaborati dal Controller passano quindi ai *Transformer*, incaricati di convertire le informazioni di input e output tra i formati esterni e quelli utilizzati internamente dal microservizio. Successivamente, i dati vengono rappresentati come comandi applicativi dal *Command*, componente che incapsula le operazioni da eseguire e le passa al livello applicativo. All'interno del Command viene implementato il flusso di esecuzione del caso d'uso, delegando a uno o più componenti di tipo *Service* l'implementazione della logica di business, così da mantenere il codice di esecuzione più chiaro e leggibile. La separazione del Controller dalla logica di business (Command/Service) consente di mantenere le regole applicative indipendenti dai dettagli del canale di esposizione e dai formati di scambio. Di conseguenza, il medesimo caso d'uso può essere riutilizzato ed esposto tramite interfacce differenti (ad es. REST o consumo di eventi), introducendo adapter dedicati senza modificare la logica di business sottostante.

Nel microservizio si distinguono tre rappresentazioni dei dati con responsabilità diverse. Il *DTO* (Data Transfer Object) rappresenta il contratto di scambio verso l'esterno ed è la struttura utilizzata dalle API in ingresso/uscita (request/response); su di esso si applicano le validazioni effettuate dal Controller (ad esempio vincoli su campi obbligatori, formati e range). L'*Object Model* (OM) rappresenta il modello interno di dominio/applicativo del microservizio: è la forma con cui Command, Orchestrator e Services manipolano i dati in modo stabile rispetto alle variazioni del contratto esterno. La *Resource* è la rappresentazione dell'output esposta dall'API, costruita a partire dall'OM per restituire al client un contenuto coerente con l'interfaccia esposta.

Le trasformazioni tra tali rappresentazioni avvengono in punti ben definiti. In ingresso, il Controller riceve un DTO e delega ai Transformer la mappatura DTO $\rightarrow$ OM, mantenendo nel Controller le responsabilità di validazione e gestione del flusso. In uscita, l'OM risultante dall'elaborazione viene convertito tramite Transformer (OM $\rightarrow$ DTO) e/o assemblato in una Resource tramite un *Resource Assembler* (OM $\rightarrow$ Resource) prima di essere restituito al chiamante, evitando che dettagli di rappresentazione si propaghino in Command e Services.

Le integrazioni verso sistemi esterni sono incapsulate nei *Connector*, intesi come adattatori di I/O: essi nascondono protocolli, serializzazione/deserializzazione e peculiarità di comunicazione (ad es. REST, eventi). All'interno dei connector, eventuali transformer specifici di integrazione (request/response transformer) convertono tra OM e formati/protocolli esterni (OM $\rightarrow$ external request; external response $\rightarrow$ OM), così che Command e Services rimangano focalizzati sulla logica applicativa, mentre orchestrazione e integrazione restano separate.

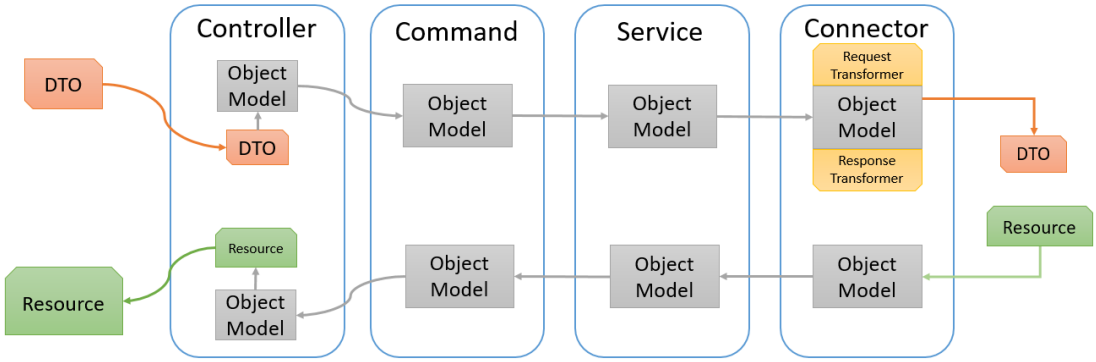

Figura 3.4: Request lifecycle interno del microservizio.

Il *Service Orchestrator* coordina l'esecuzione delle operazioni, gestendo l'ordine e le modalità con cui vengono invocati i Services, che a loro volta accedono ai sistemi di supporto tramite componenti dedicati: *Repository* per la consultazione del database e *Connector* per la pubblicazione di eventi asincroni verso lo Streaming Data Hub (Kafka). L'orchestratore può inoltre gestire transazioni distribuite, tentativi di ripetizione in caso di errore e aggregazione dei risultati provenienti da più Services.

Il layer di Repository fornisce un'astrazione dell'accesso ai dati, separando la logica di persistenza dalle operazioni di business e riducendo l'accoppiamento tra logica applicativa e tecnologia di storage. In termini di portabilità, tale astrazione concentra nel layer di accesso ai dati la maggior parte dei cambiamenti legati a una possibile evoluzione della persistenza (ad es. la sostituzione di MongoDB con un diverso database), limitando l'impatto sui componenti applicativi che consumano il repository. Inoltre, esponendo un'interfaccia uniforme per l'accesso ai dati il repository rende l'interrogazione dei dati coerente e stabile, mascherando i dettagli implementativi del database sottostante.

Va sottolineato che il diagramma rappresenta un singolo microservizio, sebbene nell'architettura reale i microservizi siano numerosi. Essi possono interagire tra loro, invocandosi reciprocamente per reperire dati o completare operazioni complesse. Tale interconnessione permette di realizzare un'architettura distribuita, flessibile e resiliente, in cui ogni microservizio mantiene la propria responsabilità pur collaborando con gli altri per il corretto funzionamento dell'intero sistema.

In fase di migrazione, un elemento determinante della progettazione riguarda l'individuazione dei confini di servizio nella transizione dal monolite legacy COBOL/CICS all'insieme di microservizi, poiché tali confini influenzano direttamente coesione, accoppiamento e strategie di migrazione. La decomposizione di un sistema monolitico legacy in microservizi costituisce infatti una sfida architetturale di rilievo:

a differenza della progettazione greenfield, in cui i confini possono essere definiti sin dall'inizio sulla base delle esigenze di business, la trasformazione di un sistema esistente richiede di bilanciare molteplici vincoli, tra cui l'allineamento ai domini di business attuali, la riduzione dell'accoppiamento, la fattibilità tecnica della migrazione e la gestione del rischio operativo [23]. Nei sistemi legacy, inoltre, i confini funzionali raramente coincidono con la modularizzazione del codice (ad esempio in COBOL), a causa di accoppiamenti impliciti tra moduli, dipendenze cicliche e una limitata separazione delle responsabilità. Di conseguenza, l'identificazione di confini di servizio adeguati richiede una comprensione approfondita dei flussi informativi e delle dipendenze transazionali [29].

3.2 Comunicazione asincrona ed Event-Driven Architecture

La scelta più significativa nell'architettura to-be riguarda il paradigma di comunicazione: il passaggio da un modello sincrono request/response bloccante a un modello asincrono basato su eventi, in grado di migliorare performance e scalabilità.

Nel modello sincrono del sistema as-is, quando un componente invia una richiesta a un altro, rimane bloccato in attesa della risposta. In scenari di alta concorrenza, con molteplici call chain che si estendono attraverso diversi livelli di servizi, l'accumulo di latenze diventa critico. Inoltre, la contesa su risorse condivise dovuta al two-phase locking introduce ulteriori code di attesa.

Per superare tali criticità, l'architettura to-be adotta una *Event-Driven Architecture* (EDA), in cui un servizio produttore pubblica un evento che descrive un fatto accaduto nel sistema, senza conoscere né attendere i servizi che lo elaboreranno.

La comunicazione avviene secondo il modello *publish-subscribe*, nel quale un servizio produttore (*producer*) emette eventi verso un intermediario di messaggistica (*message broker*), mentre altri servizi consumatori (*consumer*) si sottoscrivono agli eventi di interesse e li elaborano in modo asincrono, secondo i propri tempi. Questo approccio introduce un disaccoppiamento temporale, poiché il servizio produttore non attende l'elaborazione dell'evento da parte dei consumatori, e un disaccoppiamento logico, in quanto i servizi non comunicano direttamente ma tramite eventi [30].

Numerosi studi confermano i benefici di questo paradigma. In particolare, confronti sperimentali mostrano che architetture a microservizi basate su eventi presentano tempi di risposta medi inferiori di circa il 30% rispetto a soluzioni sincrone API-driven, oltre a una significativa riduzione del tasso di errore e a un miglior utilizzo delle risorse computazionali [10], [9].

3.2.1 Apache Kafka come message broker

In questo contesto, il message broker svolge il ruolo di intermediario tra i servizi produttori e i servizi consumatori, occupandosi della ricezione, memorizzazione e distribuzione dei messaggi. I servizi non comunicano direttamente tra loro, ma delegano al broker la gestione della consegna degli eventi ai consumatori interessati. Tale disaccoppiamento abilita comunicazioni asincrone e favorisce un'elevata scalabilità del sistema, risultando particolarmente adatto ad architetture a microservizi [25].

L'architettura to-be analizzata adotta come message broker Apache Kafka, una piattaforma di event streaming progettata per gestire flussi di dati ad alta intensità in ambienti distribuiti. A differenza dei message broker tradizionali, che gestiscono i messaggi come unità transitorie destinate a essere rimosse dopo il consumo, Kafka è costruito come un log distribuito persistente. Gli eventi vengono scritti su disco in modo sequenziale, caratteristica che consente di ottenere elevate prestazioni in scrittura, e sono replicati su più nodi del cluster per garantire durabilità e tolleranza ai guasti. I messaggi non vengono eliminati immediatamente dopo il consumo, ma restano disponibili per un intervallo di tempo configurabile, consentendo ai consumer di rileggere eventi passati per rielaborazioni e recupero da errori applicativi [25]. In aggiunta a questa conservazione, Kafka applica anche la *log compaction*, che mantiene per ciascuna chiave l'ultimo valore noto all'interno della singola partizione, eliminando progressivamente i record precedenti relativi alla stessa chiave. Il principale vantaggio è la riduzione dello storage e una conservazione più granulare, perché viene trattenuto almeno il valore più aggiornato per ogni chiave, producendo di fatto uno snapshot dello stato. Di contro, più eventi relativi alla stessa chiave possono essere soppressi, quindi non è possibile ricostruire la storia completa dell'event log per quella chiave.

3.2.2 Topic, partizioni e modello di consumo

All'interno di Kafka, gli eventi vengono organizzati all'interno di *topic*, che possono essere interpretati come canali tematici attraverso i quali vengono veicolati messaggi dello stesso tipo o relativi allo stesso dominio applicativo. Ogni topic è suddiviso internamente in più *partizioni*, che costituiscono sequenze ordinate e immutabili di messaggi salvati sul disco. L'ordine degli eventi è garantito esclusivamente all'interno della singola partizione, mentre non è assicurato a livello globale del topic. La suddivisione in partizioni rappresenta un elemento centrale del modello Kafka, in quanto consente di distribuire il carico di lavoro e di abilitare l'elaborazione parallela: più istanze di uno stesso servizio possono consumare eventi contemporaneamente leggendo da partizioni diverse, favorendo la scalabilità orizzontale del sistema [25].

Dal punto di vista del bilanciamento del carico, il producer invia i record direttamente al broker che è leader della partizione di destinazione, senza un livello

di instradamento intermedio. Per supportare questo meccanismo, i nodi del cluster forniscono al produttore metadati aggiornati (broker attivi e leader delle partizioni dei topic), consentendo di indirizzare correttamente le richieste. In questo modello, un evento pubblicato su un topic è scritto una sola volta nel log; tale evento può però essere letto da più consumer differenti.

I producer sono responsabili della pubblicazione degli eventi sui topic, senza avere conoscenza diretta dei servizi che li elaboreranno, mentre i consumer si iscrivono ai topic di interesse ed elaborano gli eventi in modo asincrono. Per gestire in modo efficiente il consumo parallelo e la distribuzione del carico, Kafka introduce il concetto di *consumer group*, ovvero un insieme di consumer che collaborano per consumare gli eventi di uno stesso topic. All'interno di un gruppo, ogni partizione è assegnata a un solo consumer, garantendo che ciascun evento venga elaborato una sola volta per gruppo. In caso di guasto o arresto di un consumer, Kafka riassegna automaticamente le partizioni ai consumer di quel gruppo ancora attivi, assicurando la continuità dell'elaborazione [25].

In questo modello, tenere traccia di ciò che è stato consumato è un aspetto chiave per garantire prestazioni e affidabilità. Kafka rappresenta tale avanzamento tramite l'*offset*, un intero che identifica la posizione del prossimo messaggio da leggere all'interno di una partizione, mantenuto per ciascuna partizione e per ciascun consumer group; di conseguenza lo stato di avanzamento può essere aggiornato in modo efficiente.

3.2.3 Affidabilità e prestazioni

Sul piano dell'efficienza, per ridurre l'overhead di I/O sia nelle operazioni client-server sia nelle operazioni persistenti interne al server, Kafka raggruppa più messaggi nella stessa richiesta (*batching*), ammortizzando i cicli di andata/ritorno, e permettendo operazioni di append/fetch su blocchi grandi e sequenziali.

Dal punto di vista infrastrutturale, invece, Kafka è composto da uno o più *broker*, ovvero nodi del cluster che si occupano della memorizzazione fisica delle partizioni e della gestione delle operazioni di lettura e scrittura. Ogni partizione è associata a un broker leader, responsabile della gestione delle richieste di I/O, e a una o più repliche mantenute su broker differenti. Questo meccanismo di replicazione consente di garantire alta disponibilità e tolleranza ai guasti: in caso di fallimento del broker leader, una delle repliche viene automaticamente promossa a nuovo leader, riducendo al minimo l'impatto sul servizio. Poiché però una replica può essere guasta o in ritardo rispetto al leader, Kafka limita l'elezione ai soli nodi dell'insieme dinamico delle repliche *in sync* (ISR), cioè sufficientemente allineate; un nodo che fallisce deve essere completamente risincronizzato prima di poter rientrare nell'ISR e tornare eleggibile. Inoltre, il fattore di replicazione, che consiste nel numero di repliche per partizione leader compreso, è tipicamente impostato

a tre e permette di proteggere il sistema dalla perdita di dati anche in presenza di guasti hardware multipli. In scenari in cui il collo di bottiglia è la banda di rete (ad esempio per flussi tra data center geograficamente distanti), Kafka sfrutta batching e compressione lato producer: più record diretti alla stessa partizione vengono aggregati e compressi in un unico batch. Il batch può essere memorizzato e trasferito in forma compressa; broker e consumer effettuano la decompressione quando necessario.

Grazie a questa architettura distribuita e alla combinazione di persistenza su disco, partizionamento e replicazione, Kafka è in grado di gestire throughput molto elevati, mantenendo una latenza contenuta, caratteristica fondamentale per sistemi event-driven basati su microservizi [25].

3.2.4 Idempotenza e semantica di consegna

Nei sistemi di messaggistica distribuita, la semantica di consegna definisce come i messaggi sono garantiti di essere elaborati dai consumatori. In particolare, esistono tre semantiche possibili: *at-most-once* (il messaggio potrebbe non essere elaborato), *at-least-once* (il messaggio potrebbe essere elaborato più volte), ed *exactly-once* (il messaggio viene elaborato esattamente una volta) [25], [31].

Nel contesto bancario, è fondamentale garantire che gli effetti delle transazioni risultino applicati una sola volta, evitando sia la perdita di messaggi sia la loro duplicazione, che potrebbero causare addebiti o accrediti errati. In architetture distribuite, questo obiettivo si ottiene attraverso il principio di *idempotenza*, ovvero la proprietà per cui l'esecuzione ripetuta di un'operazione con gli stessi parametri produce lo stesso risultato osservabile di una singola esecuzione [25], [31]. Nel sistema reale presentato, la piattaforma opera con semantica at-least-once: poiché i consumer producono effetti su sistemi esterni come database relazionali, la garanzia exactly-once end-to-end nativa di Kafka risulta inapplicabile. La correttezza viene quindi garantita a livello applicativo tramite idempotenza e scritture deterministiche, ottenendo di fatto *exactly-once effects*.

Kafka, infatti, supporta la semantica exactly-once grazie all'uso di producer e consumer idempotenti combinati con meccanismi transazionali. I producer associano a ciascun messaggio un identificativo sequenziale univoco e il broker Kafka mantiene nella cache i sequence number più alti ricevuti per ogni producer. Se il produttore ritenta l'invio dello stesso messaggio, il broker lo identifica tramite il sequence number e scarta il duplicato automaticamente. Tale meccanismo mitiga efficacemente le duplicazioni introdotte dai retry di invio sul tratto producer→broker; tuttavia, per garantire la semantica exactly-once sull'intera catena di elaborazione (end-to-end), è necessario gestire anche la consistenza sul lato consumer e l'allineamento tra stato applicativo ed eventi emessi [25], [31]. Sul lato consumer,

l'elaborazione dei messaggi è collegata alla lettura progressiva della partizione, permettendo al consumer di riprendere l'elaborazione dal punto in cui si era interrotta in caso di fallimenti. Grazie all'idempotenza applicativa, eventuali riprocessamenti dello stesso messaggio producono sempre lo stesso risultato, evitando duplicazioni o incoerenze nello stato finale [25], [31].

Per scenari più complessi, che coinvolgono operazioni critiche con più eventi, Kafka supporta scritture transazionali. Attraverso le transazioni, si garantisce che un insieme di messaggi venga reso visibile ai consumer solo se l'intera operazione va a buon fine. In caso contrario, nessun evento viene pubblicato. Questo viene realizzato usando il *pattern outbox*, nel quale gli eventi della transazione vengono prima inseriti in una tabella dedicata all'interno del database transazionale e successivamente, solo dopo che le operazioni di salvataggio sul database sono andate a buon fine, vengono pubblicati su Kafka da un servizio separato [32]. Questo approccio sfrutta le proprietà ACID del database per garantire l'allineamento tra stato applicativo ed eventi emessi, evitando situazioni di inconsistenza in caso di fallimenti parziali del sistema. Qualora il processo di pubblicazione fallisca, gli eventi rimangono memorizzati nella tabella di outbox e possono essere riprocessati senza perdita di informazioni [33], [32].

3.2.5 Resilienza e pattern di fault tolerance

In un'architettura distribuita, il fallimento di un singolo servizio non deve causare il fallimento dell'intero sistema. Il sistema as-is soffre del problema della "catena debole", dove il fallimento di qualsiasi componente nella call chain causa il fallimento dell'intera transazione. L'architettura to-be affronta questo problema attraverso diversi pattern di resilienza.

Il *retry pattern* con backoff esponenziale consente di riprovare automaticamente un'operazione fallita per cause transitorie, come il timeout di un servizio esterno, applicando ritardi crescenti tra i tentativi e aggiungendo variazione casuale, in modo da impedire che più client che stanno ritentando l'operazione lo facciano tutti nello stesso istante. Tuttavia, il retry è efficace unicamente quando applicato a operazioni idempotenti [34].

Il *bulkhead pattern* isola le risorse dedicate a specifiche categorie di operazioni, evitando che un singolo tipo di richiesta esaurisca tutte le risorse disponibili e congesti il sistema [35].

Il *timeout* e la *deadline propagation* assicurano che ogni richiesta attraverso la catena di servizi sia consapevole del tempo massimo disponibile per completarsi. Quando una richiesta attraversa diversi servizi, il timeout viene comunicato downstream (propagazione deadline), permettendo ai servizi di interrompere l'elaborazione anticipatamente se la deadline è prossima. Questo evita che il sistema investa risorse in richieste destinate a fallire a causa del timeout [34].

Il *pattern circuit breaker* interrompe temporaneamente le chiamate verso un servizio che manifesta errori ripetuti, prevenendo cascate di fallimenti; al ripristino della stabilità il circuito viene gradualmente riaperto [35]. Nel contesto bancario, il circuit breaker è fondamentale per isolare i fallimenti dei servizi e garantire fallback rapidi, evitando timeout prolungati che potrebbero bloccare le operazioni.

L'integrazione di questi pattern consente a un'architettura distribuita di tollerare fallimenti transitori, degradazioni temporanee e guasti parziali, garantendo al contempo la disponibilità e la coerenza delle transazioni bancarie.

3.2.6 Pattern Saga e coordinamento delle transazioni distribuite

In un'architettura a microservizi, le operazioni che coinvolgono più componenti applicativi non possono essere gestite tramite transazioni ACID distribuite. In questo contesto, per garantire consistenza dei dati e atomicità viene utilizzato il *Saga pattern*, attraverso il quale un'operazione complessa viene modellata come una sequenza di transazioni locali coordinate, ciascuna associata a un'eventuale azione di compensazione in caso di fallimento [36], [37]. Nello specifico, un'operazione complessa viene suddivisa in una sequenza di step, ciascuno gestito da un microservizio indipendente, preservando l'ordine degli step; se uno step fallisce, la saga esegue transazioni di compensazione per annullare gli step precedenti, ripristinando la coerenza dei dati [36].

Il Saga pattern può essere implementato secondo due approcci distinti: orchestrazione e choreografia.

Nell'approccio orchestrato, un componente centrale (orchestratore) coordina l'esecuzione della sequenza di step della transazione e gestisce esplicitamente le compensazioni, offrendo tracciabilità centralizzata dell'intero flusso [36]. Tuttavia, introduce un potenziale punto di fragilità del sistema, poiché il fallimento dell'orchestratore può compromettere l'esecuzione dell'intera saga [37].

Nella saga basata su choreografia, invece, i servizi reagiscono a eventi pubblicati sul sistema di messaggistica, senza un coordinatore centrale, favorendo disaccoppiamento, flessibilità e resilienza. Tuttavia, senza un coordinamento centrale, diventa più difficile seguire l'intero flusso della saga, e l'ordine in cui gli eventi vengono elaborati può variare, aumentando la complessità del monitoraggio [36], [37].

Kafka è particolarmente adatto alla saga basata su choreografia, poiché fornisce un canale affidabile e persistente per la pubblicazione e il consumo di eventi. La durabilità dei messaggi e la possibilità di riprocessamento consentono di gestire fallimenti dei consumer e di mantenere coerenza applicativa, caratteristiche particolarmente rilevanti nel contesto bancario [36].

Nella pratica, molte architetture moderne adottano entrambi gli approcci Saga, combinando l'orchestrazione per gestire le operazioni critiche e la choreografia basata su eventi tramite Kafka per il coordinamento tra servizi [37].

3.2.7 Sintesi e motivazioni

Kafka è stato scelto come message broker per diverse ragioni specifiche al contesto bancario:

- **Alto throughput:** gestisce milioni di messaggi al secondo, necessario per sistemi core banking ad alta frequenza transazionale [25].
- **Durabilità e replicazione:** i messaggi sono replicati su più broker, riducendo il rischio di perdita dati in caso di guasto di un nodo.
- **Modello a partizioni:** la suddivisione dei topic in partizioni consente il consumo parallelo da più consumer, favorendo scalabilità orizzontale.
- **Offset management:** Kafka tiene traccia della posizione di lettura per ogni consumer group, permettendo il ripristino dell'elaborazione in caso di interruzione.
- **Efficienza I/O e rete:** batching e compressione riducono overhead e richieste di rete, migliorando l'utilizzo di rete e disco.
- **Semantica at-least-once con idempotenza applicativa:** exactly-once effects end-to-end, cruciale per transazioni bancarie senza duplicazioni/perdite.

3.3 Framework e strumenti per l'implementazione: Spring

Il framework applicativo adottato nell'architettura to-be, proprietario dell'organizzazione, si basa sull'ecosistema Spring, in particolare su Spring Boot e Spring Cloud, che rappresentano oggi uno standard per lo sviluppo architetture a microservizi. Queste tecnologie forniscono un insieme di strumenti adatti alla realizzazione di sistemi distribuiti, asincroni e cloud-native, rispondendo alle esigenze di scalabilità, resilienza e manutenibilità tipiche dei moderni sistemi bancari.

Spring Boot è utilizzato per l'implementazione dei singoli microservizi, all'interno dei quali vengono definite la logica di business, le API REST e l'accesso ai dati tramite meccanismi di auto-configurazione e *Inversion of Control* che semplificano la gestione delle dipendenze e delle configurazioni applicative. Spring Cloud estende tali funzionalità introducendo i meccanismi necessari al coordinamento tra servizi

in ambienti distribuiti, includendo la configurazione centralizzata, il bilanciamento del carico e l’esposizione controllata delle API tramite gateway applicativi [38].

Un concetto centrale di Spring è la *Dependency Injection*, un pattern che consente di disaccoppiare i componenti di un’applicazione. Invece di istanziare manualmente le dipendenze, ogni componente dichiara ciò di cui ha bisogno e il container IoC (*Inversion of Control*) di Spring provvede a crearle e iniettarle automaticamente. Nel contesto dei microservizi bancari, la dependency injection favorisce la testabilità, poiché le dipendenze possono essere facilmente sostituite con mock, la flessibilità nell’aggiornamento delle dipendenze e il disaccoppiamento tra servizi, rendendo l’architettura più modulare, scalabile e manutenibile [38], [39], [40].

Ad esempio, nel contesto di un microservizio transaction-oriented i repository vengono iniettati automaticamente da Spring:

```
1 @Service
2 public class TransactionService {
3     @Autowired // Iniettato da Spring
4     private TransactionRepository repository;
5 }
```

3.3.1 Spring Boot per i microservizi

Spring Boot viene utilizzato come base per lo sviluppo dei microservizi grazie alla sua capacità di semplificare la creazione di applicazioni Java standalone. A differenza delle tradizionali applicazioni Java EE, che richiedevano estese configurazioni XML e deployment su server centralizzati, Spring Boot offre meccanismi che velocizzano lo sviluppo e riducono la complessità: esegue automaticamente la configurazione dei bean in base alle librerie presenti nel classpath, integra un web server direttamente nell’applicazione eliminando la necessità di un application server esterno, e mette a disposizione “starter dependencies”, ovvero pacchetti preconfigurati di librerie correlate che semplificano la gestione delle dipendenze e riducono i problemi di versioning, ovvero le incompatibilità tra diverse versioni delle librerie utilizzate [38], [39].

Nel contesto della modernizzazione di un core banking system, questi aspetti permettono di sviluppare rapidamente nuovi microservizi senza affrontare la complessità configurativa tipica dei sistemi legacy. Ogni servizio è un’applicazione Spring Boot indipendente, eseguibile e scalabile autonomamente [40], [41].

Rispetto ai sistemi legacy COBOL/CICS, l’adozione di Spring Boot comporta miglioramenti significativi sia tecnici sia organizzativi. Il ciclo di sviluppo diventa più rapido e flessibile, grazie a pratiche agili e rilasci frequenti supportati da integrazione continua, a differenza dei tradizionali rilasci lenti e sequenziali. La scalabilità delle applicazioni è ora orizzontale, con più istanze autonome che

permettono di rispondere dinamicamente al carico, mentre i sistemi legacy erano limitati alla scalabilità verticale. Spring Boot è inoltre nativamente cloud-native, facilitando containerizzazione e orchestrazione su piattaforme cloud, e l'ecosistema di librerie disponibili è ampio e costantemente aggiornato, offrendo strumenti moderni e consolidati rispetto alle librerie obsolete dei sistemi legacy [38], [39].

3.3.2 Spring Cloud per architetture distribuite

Spring Cloud estende le funzionalità di Spring Boot per affrontare le complessità delle architetture distribuite, offrendo strumenti per gestire routing intelligente, configurazioni centralizzate, resilienza e tracciamento delle richieste tra più microservizi. Funzionalità come Spring Cloud Gateway e Spring Cloud Config semplificano rispettivamente l'esposizione delle API e la configurazione di tutti i microservizi in diversi ambienti, riducendo il coupling tra codice e infrastruttura. Librerie come Resilience4j, integrate tramite Spring Cloud Circuit Breaker, permettono di implementare pattern di resilienza dichiarativi, evitando che errori o timeout di un servizio si propaghino all'intero sistema. Strumenti di distributed tracing come Spring Cloud Sleuth facilitano il monitoraggio tracciando il percorso di una richiesta attraverso molteplici microservizi. Infine, framework come Spring Cloud Stream forniscono un'astrazione dei message broker come Kafka, permettendo ai microservizi di pubblicare e consumare eventi in modo dichiarativo senza gestire direttamente connessioni o logica di basso livello del broker [38], [39], [40].

Spring Cloud gestisce automaticamente meccanismi di resilienza essenziali per ambienti distribuiti, come l'apertura di circuiti quando un servizio fallisce ripetutamente, in modo da rifiutare le richieste, il retry automatico delle richieste in caso di errori temporanei e l'interruzione delle chiamate che superano un timeout massimo [38], [39], [40]. Queste funzionalità riducono significativamente la complessità della gestione di fallimenti distribuiti, un aspetto critico nel banking dove la propagazione a cascata degli errori tra i servizi deve essere prevenuta [41].

3.3.3 Spring Cloud Stream per l'integrazione event-driven tramite Kafka

Il framework adottato nell'architettura to-be incapsula le funzionalità necessarie a collegare i microservizi con l'infrastruttura di messaggistica Apache Kafka. Tale astrazione è realizzata principalmente attraverso Spring Cloud Stream, che introduce un'astrazione di alto livello chiamata *binder* per collegare il codice applicativo del microservizio al message broker sottostante, senza interagire direttamente con le API native di Kafka. In pratica, Spring Cloud Stream traduce binding dichiarativi in produttori e consumatori Kafka reali e gestisce automaticamente una serie di aspetti infrastrutturali, che altrimenti richiederebbero codice ripetitivo e verboso

per la connessione a Kafka, la serializzazione dei messaggi, la gestione dei consumer group e degli offset [38], [39], [40].

Gran parte della configurazione dell'integrazione event-driven avviene a livello dichiarativo, tramite file di configurazione (`application.yml`). In questo modo, la definizione dei canali di input e output e la loro associazione ai topic Kafka è separata dalla logica di business del microservizio.

```yaml
spring:
  cloud:
    stream:
      bindings:
        transaction-events-out:
          destination: transaction-events
          content-type: application/json
        transaction-events-in:
          destination: transaction-events
          group: notification-service-group
```

In questo esempio, l'applicazione dichiara l'esistenza di un canale di output e di uno di input, associandoli al topic Kafka `transaction-events`. Il binder Kafka traduce automaticamente questa configurazione in producer e consumer concreti.

Un producer invia un evento in modo asincrono, mentre un consumer viene invocato automaticamente dal framework quando un nuovo messaggio è disponibile.

Producer:

```java
@Service
public class TransactionPublisher {
    @Autowired
    private TransactionEventChannel channel; // Binder

    public void publish(TransactionCompletedEvent event) {
        channel.transactionEvents()
               .send(MessageBuilder.withPayload(event).build());
        // Pubblica l'evento sul topic Kafka "transaction-events"
        // (definito nel file di configurazione).
        // La chiamata ritorna immediatamente senza attendere
    risposta
    }
}
```

Consumer:

```java
@Service
public class TransactionEventListener {
    @StreamListener("transaction-events-in")
```

```
4       public void handleTransaction(TransactionCompletedEvent event)
        {
5           // Logica di elaborazione dell'evento.
6           // Il metodo viene invocato automaticamente quando
7           // un nuovo evento arriva su Kafka
8       }
9   }
```

Questo approccio consente di mantenere i microservizi fortemente focalizzati sulla logica di business, delegando al framework la gestione della complessità infrastrutturale. L'astrazione fornita da Spring Cloud Stream è particolarmente preziosa nel contesto bancario, dove la scelta del message broker può cambiare nel tempo poiché non richiede refactoring massiccio, riducendo al contempo il rischio di errori implementativi e rendendo l'adozione di un'architettura event-driven più sostenibile nel tempo [39], [40].

3.4 Confronto tra architettura sincrona legacy e architettura asincrona to-be

L'adozione di un'architettura event-driven basata su microservizi rappresenta un cambiamento significativo rispetto al modello sincrono dei sistemi legacy COBOL/CICS. Mentre l'architettura as-is si fonda su interazioni sincrone e bloccanti, l'architettura to-be introduce un paradigma asincrono e disaccoppiato, in cui i servizi comunicano tramite eventi pubblicati su un message broker. Dal punto di vista delle prestazioni, i sistemi legacy sincroni presentano limiti intrinseci di latenza e throughput, dovuti a chiamate sequenziali tra componenti fortemente dipendenti, che riducono la capacità di gestire picchi di carico. Al contrario, l'architettura to-be asincrona consente ai microservizi di produrre e consumare eventi in modo indipendente, abilitando elaborazioni parallele e un throughput più elevato. L'impiego di broker come Kafka contribuisce inoltre ad assorbire le variazioni di carico grazie alla persistenza degli eventi [10], [25].

Un ulteriore limite dei sistemi legacy riguarda la resilienza. In un modello sincrono, il fallimento o il rallentamento di un singolo componente può propagarsi rapidamente agli altri servizi dipendenti, generando fenomeni di *cascading failure* e timeout a catena [8]. In un'architettura event-driven, invece, il disaccoppiamento temporale tra produttori e consumatori consente di isolare tali malfunzionamenti: un servizio può accettare una richiesta, pubblicare l'evento sul message broker e restituire un acknowledgment al client anche in presenza di indisponibilità temporanee dei servizi a valle. Gli eventi restano persistiti nel broker e vengono elaborati

non appena i consumer tornano operativi, permettendo al sistema di degradare in modo controllato anziché fallire globalmente [34].

Anche in termini di scalabilità emergono differenze sostanziali. I sistemi COBOL/CICS adottano prevalentemente una scalabilità verticale, basata sull'incremento delle risorse hardware, con costi elevati e limiti strutturali. L'architettura to-be, invece, supporta una scalabilità orizzontale nativa, in cui i microservizi possono essere replicati dinamicamente in base al carico, risultando più adatta a contesti cloud-native [39], [40].

Infine, la gestione della consistenza dei dati evolve da un modello fortemente consistente e centralizzato, tipico dei sistemi legacy in cui le transazioni rispettano le proprietà ACID, verso un approccio di *consistenza eventuale* [16]. In un'architettura event-driven, le modifiche ai dati vengono propagate in modo asincrono e indipendente, con un breve intervallo temporale in cui i dati potrebbero non essere ancora uniformemente aggiornati. Questo modello, noto come consistenza eventuale, garantisce che, alla fine dell'elaborazione di tutti gli eventi, lo stato del sistema risulti coerente, pur consentendo una maggiore scalabilità e resilienza. Sebbene questo introduca maggiore complessità concettuale, essa viene gestita attraverso pattern consolidati come saga, idempotenza dei consumer ed eventi compensativi, che consentono di mantenere l'affidabilità funzionale richiesta in ambito bancario [31], [36], [37].

Nel complesso, il passaggio da un'architettura sincrona legacy a un'architettura asincrona event-driven non costituisce solo un'evoluzione tecnologica, ma un cambiamento di paradigma che rende il core banking system più resiliente, scalabile e manutenibile nel lungo periodo.

3.5 Verso l'implementazione

L'architettura to-be descritta nei paragrafi precedenti definisce una visione coerente di come modernizzare il sistema di core banking tramite microservizi, architetture event-driven e tecnologie cloud-native. Tuttavia, la realizzazione concreta di tale architettura richiede una strategia di implementazione che consideri sia gli aspetti tecnici sia i rischi operativi associati a un sistema critico in esercizio, come quello bancario. In questo contesto, dove la discontinuità di servizio rappresenta un rischio inaccettabile, la migrazione dal sistema as-is al to-be non può avvenire tramite una transizione atomica, bensì richiede un approccio graduale e controllato. Una strategia comunemente utilizzata è la coesistenza controllata dei due sistemi durante la fase di transizione: inizialmente, una porzione limitata del traffico viene instradata verso il sistema to-be, mentre il sistema legacy continua a gestire la maggior parte delle operazioni. Con il progressivo consolidamento dell'affidabilità

e delle performance del nuovo sistema, il traffico viene progressivamente migrato fino alla completa dismissione del sistema legacy.

Capitolo 4

Implementazione to-be: strategie di migrazione e meccanismo di coesistenza

La migrazione di un sistema core banking da un'architettura legacy sincrona e monolitica verso un'architettura moderna basata su microservizi rappresenta un'iniziativa di trasformazione strategica di ampia portata. Tale processo non si limita alla mera sostituzione tecnologica, ma incide profondamente sulle modalità operative, sulla capacità di evoluzione del sistema informativo e sull'abilità dell'organizzazione di rispondere in modo tempestivo e affidabile alle esigenze del business.

Nel settore bancario, caratterizzato da requisiti stringenti in termini di affidabilità, disponibilità e conformità normativa, una migrazione architetturale deve essere pianificata e condotta con estrema cautela, adottando strategie che consentano di ridurre il rischio operativo e garantire la continuità del servizio durante l'intero percorso di transizione.

4.1 Obiettivi della migrazione

4.1.1 Continuità operativa

L'obiettivo primario della migrazione è garantire la continuità operativa ininterrotta del sistema core banking durante l'intero processo di transizione. In ambito bancario, una qualsiasi interruzione del servizio, anche di breve durata, può avere conseguenze significative: perdita di fiducia da parte dei clienti, violazione di accordi di servizio (*Service Level Agreements*), sanzioni normative, e danni reputazionali [42]. Di conseguenza, la migrazione non può essere effettuata tramite un approccio di tipo *big-bang*, in cui il sistema legacy viene disattivato e sostituito istantaneamente

dal nuovo sistema, poiché qualsiasi anomalia del nuovo sistema si tradurrebbe in un'interruzione completa del servizio ed esporrebbe l'organizzazione a un elevato rischio di fallimenti catastrofici [43]. La continuità operativa, quindi, impone l'adozione di una transizione graduale e controllata, nella quale il sistema legacy e il nuovo sistema coesistono e operano in parallelo per un periodo transitorio, garantendo al contempo coerenza dei dati.

Un ulteriore aspetto critico della continuità operativa riguarda la gestione dei fallimenti parziali. Qualora un componente del nuovo sistema non sia disponibile o manifesti comportamenti anomali durante la fase di transizione, l'architettura deve consentire il reindirizzamento automatico delle richieste verso il sistema legacy, senza impatto sull'esperienza del cliente [44].

4.1.2 Minimizzazione delle regressioni funzionali e validazione progressiva

Un secondo obiettivo critico è minimizzare il rischio di regressioni funzionali, ovvero il rischio che il nuovo sistema non riproduca correttamente una funzionalità del sistema legacy. Questo aspetto è particolarmente rilevante nei contesti bancari, dove i sistemi legacy COBOL/CICS sono utilizzati da decenni e incorporano moltissime funzionalità che potrebbero non essere esplicitamente documentate [45], [46].

Le regressioni funzionali possono avere conseguenze severe, tra cui errori nel trattamento delle transazioni finanziarie, violazioni delle normative di conformità e inconsistenze nei dati esposti ai clienti. Anche differenze apparentemente minime nel calcolo di un saldo o nell'applicazione di una regola possono generare effetti difficilmente individuabili a posteriori [45], [46].

Per mitigare tali rischi, la strategia di migrazione è progressiva: anziché migrare l'intero sistema contemporaneamente, servizi specifici sono migrati uno alla volta e validati tramite test e monitoraggio in produzione. L'ordine di estrazione dei servizi è strategico: tipicamente si privilegiano inizialmente quelli con basso accoppiamento col resto del sistema, alta coesione interna (confini chiari) e significativo valore di business [47].

La possibilità di confrontare il comportamento del nuovo sistema con quello legacy in condizioni operative reali consente di circoscrivere eventuali errori e ridurre l'impatto complessivo di una regressione, facilitando al contempo l'individuazione delle cause e permettendo di validare ogni servizio in isolamento prima di procedere al successivo, riducendo così la probabilità che errori sistematici si accumulino [45], [46].

In questo contesto, a differenza del testing tradizionale, la validazione progressiva non avviene in ambienti isolati ma in produzione con traffico reale. Questo consente di rilevare problematiche che difficilmente emergerebbero nei test, come colli di

bottiglia prestazionali sotto carichi reali, comportamenti anomali in presenza di casi limite o problemi di integrazione con sistemi esterni [34], [46].

La validazione progressiva si basa sulla raccolta continua di metriche operative e sul confronto dei risultati prodotti dai due sistemi. Questo approccio consente di verificare che il nuovo sistema riproduca correttamente le funzionalità esistenti, controllare i tempi di risposta per assicurare che le prestazioni rimangano accettabili, monitorare eventuali errori o anomalie nel funzionamento e verificare che i dati tra i due sistemi rimangano coerenti. La possibilità di osservare simultaneamente l'output del sistema legacy e del nuovo sistema per lo stesso insieme di operazioni è fondamentale per valutare la correttezza funzionale del nuovo sistema, permettendo di individuare eventuali discrepanze e intervenire prima che possano avere impatti significativi sull'operatività [34], [46].

4.1.3 Preparazione alla dismissione del sistema legacy

Il terzo obiettivo della migrazione consiste nel creare le condizioni ideali per la dismissione sicura del sistema legacy una volta che il nuovo sistema ha dimostrato piena affidabilità e maturità operativa. Tale dismissione risulta fondamentale per ridurre i costi operativi, eliminare il debito tecnico accumulato nel tempo (ossia la complessità e le inefficienze accumulate nel codice legacy dopo numerose manutenzioni) e liberare risorse che possono essere reinvestite in iniziative di innovazione [2].

Tuttavia, anche la dismissione del legacy è un processo graduale che deve essere pianificato con attenzione: inizialmente il sistema rimane operativo senza ricevere traffico nuovo per scopi di diagnostica, quindi una copia completa del sistema legacy (codice, dati, configurazione) è conservata come backup, e infine, una volta confermata la completa migrazione delle funzionalità, il sistema viene spento definitivamente e le risorse associate deallocate [2], [45].

4.2 Strategia di esecuzione parallela e migrazione incrementale

Una conseguenza diretta della scelta di una migrazione incrementale è l'introduzione di una fase comunemente indicata in ambito bancario come *parallel strategy*, intesa come un periodo durante il quale il sistema legacy e il nuovo sistema target coesistono e operano contemporaneamente sul medesimo dominio funzionale, elaborando in parallelo transazioni di clienti reali [43], [48]. Questo approccio di coesistenza si inserisce in una strategia di modernizzazione ispirata allo *Strangler Pattern*, ampiamente adottato nei processi di trasformazione di sistemi monolitici complessi. Tale pattern prevede l'introduzione graduale di nuovi microservizi che sostituiscono,

uno alla volta, le funzionalità corrispondenti nel sistema legacy, consentendo al legacy di continuare a operare senza interruzioni [30], [49]. Ogni servizio viene identificato, estratto, reimplementato nel nuovo sistema e validato singolarmente prima di procedere al servizio successivo, senza interrompere il funzionamento del sistema esistente. Dal punto di vista operativo, tale strategia è implementata attraverso tecniche *traffic splitting*, che consentono di instradare una percentuale limitata delle transazioni verso il sistema target durante le diverse tappe della migrazione [46], [50].

Nella prima fase, di validazione, tutte le richieste vengono elaborate dal sistema legacy, mentre una copia delle stesse viene inoltrata al nuovo sistema, che riceve quindi il 100% delle richieste ma senza influenzare la risposta restituita al client. Il sistema to-be elabora quindi le richieste in parallelo, producendo un risultato che viene confrontato con quello del legacy. In caso di divergenze, l'anomalia viene tracciata e analizzata, ma la risposta restituita al client rimane quella del sistema legacy, garantendo la continuità operativa e l'assenza di impatti percepibili per l'utente finale. Questa fase si protrae fino a quando il nuovo sistema dimostra di replicare in modo affidabile e consistente il comportamento funzionale del legacy [34], [51].

Una volta raggiunto un livello soddisfacente di equivalenza funzionale, il processo evolve verso una migrazione graduale del traffico. In questa fase, solo una porzione controllata delle richieste, tipicamente un sottoinsieme di client o una percentuale limitata del traffico complessivo, viene instradata direttamente verso il nuovo sistema, mentre il resto continua a essere gestito dal legacy. Il comportamento del nuovo sistema viene monitorato in modo intensivo, consentendo un rapido rollback verso il legacy qualora emergano anomalie funzionali o prestazionali [50]. Nel caso preso in esame, la suddivisione del traffico avviene attraverso il codice ABI del conto, che identifica la specifica divisione bancaria di appartenenza (ad esempio conti retail, conti online, conti per imprese o prodotti finanziari), consentendo una migrazione controllata per segmenti funzionali omogenei. L'aumento progressivo della quota di traffico instradata al sistema target avviene solo a fronte di evidenze operative positive.

Il completamento della migrazione avviene con il cosiddetto *cutover*, momento in cui il nuovo sistema diventa l'unico responsabile dell'elaborazione delle richieste in produzione. Anche in questa fase, il legacy system viene generalmente mantenuto in uno stato di standby per un periodo limitato, al fine di consentire un eventuale rollback di emergenza prima della sua definitiva dismissione. Una volta completata questa fase e verificato che nessun processo o client dipende più dal sistema legacy, quest'ultimo viene dismesso definitivamente, con la conseguente eliminazione dei costi di manutenzione e gestione associati. L'approccio graduale riduce in modo significativo il tasso di incidenti post-migrazione rispetto a strategie di tipo big-bang, abbassando sensibilmente il rischio di interruzioni del servizio [43], [45], [48].

4.2.1 Aspetti critici

Nonostante i benefici in termini di riduzione del rischio, la migrazione incrementale introduce una serie di sfide architetturali e operative rilevanti.

Nello specifico, è necessario garantire la sincronizzazione dei dati tra i due sistemi, in modo che entrambi abbiano costantemente accesso a informazioni aggiornate, evitando disallineamenti [51].

Un ulteriore aspetto critico riguarda l'instradamento delle richieste: deve essere definito un meccanismo di routing deterministico e controllabile, in grado di stabilire per ogni richiesta quale sistema debba gestirla. Allo stesso tempo, è necessario gestire correttamente i fallimenti del nuovo sistema, prevedendo strategie di fallback automatico verso il legacy che consentano di completare l'operazione senza perdita di dati né interruzioni percepibili dal cliente [2], [44].

La migrazione richiede inoltre una validazione continua dell'equivalenza dei risultati prodotti dai due sistemi, al fine di individuare tempestivamente eventuali anomalie o regressioni funzionali. A ciò si aggiunge un inevitabile overhead operativo, dovuto alla necessità di monitorare e mantenere due sistemi in parallelo durante l'intero periodo di coesistenza [45], [46].

4.2.2 Motivazioni della scelta

L'adozione della strategia di esecuzione parallela è motivata dalle esigenze operative e di controllo che caratterizzano fortemente il settore bancario. In primo luogo, la strategia risponde alla necessità di ridurre il rischio operativo associato alla migrazione: rispetto a un approccio big-bang, l'esecuzione parallela permette di contenere l'impatto degli errori e di attivare rollback selettivi senza interrompere mai completamente l'erogazione del servizio [43], [48].

In secondo luogo, la fase di coesistenza abilita una validazione funzionale e prestazionale in condizioni reali, offrendo osservabilità diretta su metriche operative quali tasso di errore, latenza e throughput; il confronto continuo di queste metriche tra legacy e to-be fornisce evidenze oggettive per decidere l'avanzamento, la stabilizzazione o l'eventuale rollback di ciascun servizio [34].

Infine, l'approccio incrementale consente di migrare i servizi uno alla volta, isolando progressivamente le responsabilità dal sistema monolite verso i microservizi indipendenti. Ciò riduce il rischio di regressioni funzionali e permette di validare ogni componente in modo mirato, assicurando una transizione controllata, misurabile e conforme ai requisiti normativi [2], [46]. L'approccio parallelo fornisce inoltre metriche concrete per confrontare le prestazioni e i risultati tra i due sistemi, facilitando il cutover finale e la dismissione del legacy senza interruzioni significative per gli utenti [50].

In sintesi, l'esecuzione parallela non rappresenta soltanto un meccanismo operativo, ma un pilastro strategico per una transizione affidabile verso il sistema target.

4.3 Modello di coesistenza

Nel presente caso di studio, la coesistenza controllata tra i due sistemi è realizzata tramite una modalità di migrazione incrementale in cui i core applicativi implementano un instradamento condizionale, governato da una variabile pilota, che determina per ciascuna richiesta se l'elaborazione debba essere eseguita sul mainframe oppure sul percorso to-be. In questa fase le richieste sono elaborate dal mainframe (percorso attivo) e una copia delle stesse è inoltrata ai core del percorso to-be in modalità shadow, così da supportare la validazione progressiva della migrazione tramite evidenze confrontabili, riducendo il rischio operativo e consentendo un avanzamento controllato del cutover.

La modalità shadow è progettata per essere non-interferente: a seconda della tipologia dell'operazione (lettura, scrittura, o chiamata soggetta a ghosting) la copia shadow viene rieseguita, materializzata tramite eventi, oppure tracciata come evidenza. Questa distinzione è essenziale in un contesto bancario, in cui la riesecuzione indiscriminata di operazioni dispositive introdurrebbe rischi di duplicazione ed effetti collaterali.

4.3.1 Flusso operativo

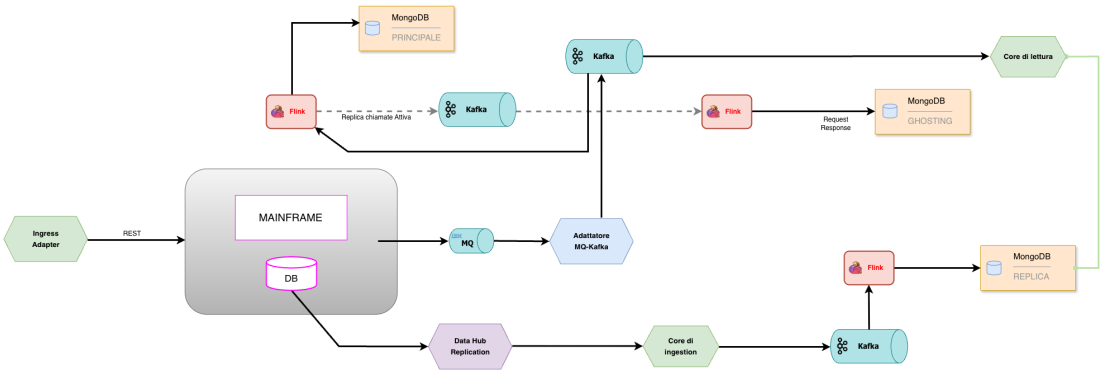

Figura 4.1: Flusso di coesistenza controllata nel caso di studio.

La coesistenza dei due sistemi è rappresentata nel grafico come un flusso a due rami, in cui il ramo attivo interagisce con il mainframe e produce gli esiti

di riferimento, mentre il ramo passivo alimenta viste derivate e risposte shadow interrogando un database replica.

Si possono distinguere tre diversi database, impiegati con responsabilità differenti durante la coesistenza:

- **DB principale**, destinato a diventare il database definitivo del to-be a regime.
- **DB replica**, alimentato da un flusso di replica del database mainframe, usato per letture shadow e controlli di convergenza.
- **DB di ghosting**, usato come archivio di evidenze per richiesta (request/response) per operazioni non duplicabili.

Nel caso di studio coesistono due flussi asincroni con responsabilità distinte. Il primo è un flusso event-driven basato su IBM Message Queue → Kafka → Flink, utilizzato per trasformare e materializzare sul DB principale le informazioni di aggiornamento prodotte dal mondo legacy. Il secondo è un flusso di replica basato sul componente *Data Hub Replication* (DHR), che intercetta le modifiche sul database legacy e abilita l'allineamento near real-time del DB replica tramite una coda Kafka dedicata e job Flink.

Nel ramo attivo, quando una richiesta viene instradata verso il legacy, i core invocano il mainframe e ottengono la response facendo riferimento ai suoi dati (database applicativi del dominio). Questo ramo fornisce l'esito "di riferimento" nella fase di validazione, perché riflette il comportamento consolidato del sistema preesistente.

Per le operazioni che comportano mutazione di stato, il flusso non si limita alla risposta sincrona: le variazioni di stato vengono propagate anche nel dominio di integrazione asincrona. In questo caso entra in gioco IBM MQ, che nel contesto aziendale svolge un ruolo fondamentale di ponte tra mondo legacy e mondo moderno. MQ permette al mainframe di pubblicare informazioni di aggiornamento che possono essere trasformate e inoltrate verso il backbone eventi moderno Kafka, senza richiedere che il mainframe parli nativamente protocolli e formati tipici del mondo cloud-native. Un componente di integrazione legge dalla coda e pubblica degli eventi sul backbone, a valle del quale i job di Flink consumano gli eventi e li materializzano sul database principale. Il database principale è il risultato di questa materializzazione: non è il database nativo del mainframe, ma una copia applicativa del suo stato. Ed è proprio questo DB che, a regime, diventerà la base dati definitiva del nuovo sistema; durante la fase di coesistenza, tuttavia, rimane alimentato dagli eventi e non toccato direttamente dai microservizi, così da mantenere controllo, tracciabilità e coerenza del flusso di migrazione.

Una volta che gli eventi sono disponibili su Kafka, più consumer possono leggere lo stesso stream per scopi diversi (materializzazione, replica e validazione). Le

letture vengono quindi rieseguite in modalità shadow dai microservizi nel ramo to-be interrogando il database replica, in questo modo le response ottenute saranno confrontabili con quelle sincrone del ramo attivo.

Per le scritture, invece, la modalità shadow non prevede la riesecuzione della richiesta nel ramo passivo per evitare duplicazioni ed effetti collaterali. L'allineamento del dato avviene tramite DHR collegato al database legacy (interno al componente Mainframe nella figura), che intercetta le modifiche e notifica ai core di ingestion le variazioni da materializzare, in modo da creare uno stato derivato consultabile dai microservizi to-be, senza accedere al database transazionale né introdurre scritture concorrenti.

Funzionalmente, entrambi i flussi possono essere ricondotti, a livello logico, a una pipeline di ingestion articolata in tre fasi: acquisizione, trasformazione e materializzazione. Nella prima fase le variazioni di stato generate dal legacy vengono raccolte (via MQ o DHR, a seconda del flusso) e pubblicate su Kafka. Nella seconda fase gli eventi vengono elaborati da uno strato di stream processing, implementato nel presente studio tramite Apache Flink, che applica regole di validazione dei dati, normalizza i payload, effettua il mapping verso lo schema target dei microservizi e gestisce condizioni operative come duplicati o messaggi fuori sequenza rispetto alle chiavi di correlazione definite. Nella terza fase l'output della trasformazione viene scritto su uno o più sink, ovvero database che rendono il risultato dell'elaborazione disponibile come stato consultabile.

L'uso di Flink consente di separare in modo netto le responsabilità della pipeline: la logica di trasformazione rimane concentrata nei job di elaborazione, mentre i sink vengono trattati come endpoint indipendenti. Poiché in architetture event-driven la consegna è tipicamente at-least-once e quindi alcuni messaggi possono essere rielaborati, la scrittura sui sink viene progettata per essere idempotente, in modo che l'applicazione ripetuta dello stesso evento non produca effetti duplicati sullo stato salvato.

Infine, per garantire la validazione senza riesecuzione delle operazioni non idempotenti/non duplicabili, request e response di tali chiamate sono salvate in un database di ghosting, così da conservare l'evidenza completa dell'esecuzione senza richiedere che il ramo passivo riesegua la stessa operazione.

Il DB di ghosting non contiene una replica dello stato applicativo, ma un archivio di evidenze per richiesta utilizzato per correlare le operazioni e rendere diagnosticabili eventuali discrepanze durante la fase di migrazione.

La natura asincrona dei flussi introduce inevitabilmente ritardi tra la registrazione di un evento sul sistema legacy e il momento in cui lo stesso cambiamento risulta visibile nel sistema to-be. Questi ritardi possono generare inconsistenze temporanee, che vengono gestite definendo soglie di tolleranza: divergenze entro limiti di tempo predefiniti sono accettabili, mentre oltre le soglie vengono generati alert e attivate procedure di riallineamento dei dati. In questo modo, l'architettura

bilancia coerenza, latenza e throughput, mantenendo la continuità operativa anche in scenari ad alto volume [51], [52].

4.3.2 Modalità di validazione

In sintesi, per le letture la validazione avviene confrontando: la response prodotta dal ramo attivo (ottenuta tramite interrogazione del mainframe e dei suoi dati), con la response prodotta dal ramo passivo, calcolata dai microservizi interrogando il DB replica. Questo confronto permette di individuare regressioni funzionali (mismatch di campi, regole, arrotondamenti, mapping) e di misurare differenze prestazionali (latenza, throughput), mantenendo invariata l'esperienza utente nella fase in cui la risposta al client resta quella del ramo attivo.

Per le scritture, la validazione non può basarsi su riesecuzione nel ramo passivo; si verifica invece la convergenza tra due viste materializzate dello stato legacy, ottenute rispettivamente dal flusso MQ→Kafka→Flink (DB principale) e dal flusso DHR→Kafka→Flink (DB replica). Operativamente, questo si traduce in controlli di allineamento tra DB principale e DB replica (con eventuali tolleranze temporali dovute alla natura asincrona della pipeline), così da garantire che il ramo passivo ricostruisca correttamente le stesse variazioni di stato osservate sul legacy.

4.4 Architettura di migrazione

Nel percorso di migrazione verso un'architettura a microservizi, l'integrazione tra il sistema legacy mainframe e i componenti to-be è un elemento centrale per garantire continuità operativa nella fase di esecuzione parallela.

In tale contesto, vengono adottati due meccanismi complementari:

- **Integrazione sincrona**, tramite API REST e layer di adattamento, per i flussi request/response che richiedono un esito immediato.
- **Integrazione asincrona**, tramite backbone eventi, per propagare cambiamenti di stato e allineare progressivamente i dati nel sistema target in modo disaccoppiato.

Le chiamate sincrone vengono impiegate nei flussi transazionali in tempo reale, quando la risposta deve essere deterministica e immediata verso i canali. In questi casi, il routing (gestito dal gateway) consente di indirizzare le richieste verso il backend attivo e, per le sole operazioni idempotenti, di inoltrare in parallelo una copia in modalità shadow verso il percorso to-be.

Le integrazioni asincrone sono invece utilizzate per la sincronizzazione e materializzazione dei dati e per l'attivazione di elaborazioni non vincolate al tempo di risposta della richiesta, riducendo l'accoppiamento temporale tra i componenti.

Questi due livelli di integrazione cooperano per supportare i flussi operativi durante la fase di coesistenza, consentendo l'esecuzione parallela delle funzionalità e la sincronizzazione progressiva dei dati tra sistema legacy e sistema target durante l'intero periodo di migrazione [53], [54], [55].

4.4.1 Integrazione sincrona tra legacy e microservizi

L'integrazione tra sistema legacy e microservizi è realizzata attraverso un insieme coordinato di componenti che consentono di mantenere stabile l'interfaccia verso i client mentre la logica applicativa viene progressivamente spostata nel nuovo sistema.

Al centro di questo schema si trova l'API gateway, che rappresenta l'unico punto di accesso per i client. Il gateway centralizza funzionalità trasversali come autenticazione, autorizzazione, rate limiting e logging, e si occupa dell'instradamento delle richieste verso il backend appropriato. Il routing dinamico è deterministico (non casuale), per garantire che richieste successive dello stesso cliente vadano sempre allo stesso backend durante la fase di coesistenza, evitando anomalie di concorrenza. Inoltre, l'API Gateway consente di applicare pattern di resilienza come il circuit breaker, contribuendo a isolare i guasti e a migliorare la stabilità complessiva del sistema [50], [44], [56].

Per gestire le differenze tecnologiche tra i due ambienti, l'architettura introduce specifici layer di adattamento tra legacy e microservizi. Tali componenti possono assumere forme diverse a seconda delle funzionalità esposte dal sistema esistente; nel caso in esame il layer di adattamento è realizzato tramite wrapper REST attorno alle transazioni CICS esistenti, che consentono ai microservizi e all'API gateway di invocare le funzionalità del mainframe attraverso interfacce HTTP standard.

Grazie a questo schema, è possibile introdurre progressivamente i nuovi microservizi senza modificare radicalmente i canali esistenti. Di conseguenza, i sistemi di front-end continuano a interagire con un insieme coerente di API esposte dal gateway, indipendentemente dal fatto che la logica sottostante sia ancora eseguita nel mainframe o già migrata nei microservizi [2], [53].

4.4.2 Integrazione asincrona: backbone eventi

Per supportare flussi di esecuzione asincroni e disaccoppiare i microservizi dal sistema legacy viene adottata un'*Event-Driven Architecture* (EDA). Essa introduce un backbone eventi basato su una piattaforma di event streaming, in questo caso Apache Kafka. Tale infrastruttura consente di gestire comunicazioni asincrone tra i servizi, migliorando scalabilità, resilienza e tracciabilità dei flussi applicativi. In questo modello, Kafka svolge il ruolo di message broker secondo il paradigma *publish and subscribe*. In particolare, componenti legacy pubblicano eventi su topic

specifici, mentre i servizi interessati li consumano in base alle proprie responsabilità applicative, senza dipendenze dirette tra produttori e consumatori [55], [57].

L'introduzione del backbone eventi richiede un allineamento tra il modello dati del sistema legacy e il modello eventi del nuovo sistema to-be. A questo scopo vengono adottati meccanismi di cattura delle variazioni lato legacy: ogni variazione ai dati, come l'aggiornamento di un conto o la registrazione di una transazione, viene intercettata e trasformata in un evento pubblicato sul bus. Oltre alla generazione degli eventi, è necessario progettare con attenzione anche la fase di ingestion sul lato to-be, in cui pipeline di materializzazione applicano le variazioni ricevute al modello dati target. Gli eventi vengono trasformati in operazioni di scrittura sul database principale del to-be attraverso pipeline che includono validazione, arricchimento e mapping verso schemi normalizzati [53], [55].

Durante la fase di coesistenza, il sistema legacy resta la fonte ufficiale dei dati ("system of record"), mentre i microservizi costruiscono progressivamente viste derivate dei dati, come cache o proiezioni ottimizzate per specifici casi d'uso [34], [53]. Questa strategia consente di testare e utilizzare i nuovi servizi senza modificare il funzionamento del legacy, garantendo al contempo la coerenza dei dati tramite meccanismi di idempotenza e controlli di sequenza.

4.5 Implementazione operativa della coesistenza

4.5.1 Requisiti di non interferenza

L'implementazione operativa della coesistenza richiede che i nuovi microservizi non alterino il comportamento del sistema legacy durante la fase di coesistenza. Nelle fasi iniziali di adozione di architetture cloud-native, i componenti to-be operano in modalità osservazionale o limitata, evitando di modificare stati critici finché non è stata acquisita sufficiente confidenza sulla correttezza del loro comportamento [2].

Nel caso di studio in esame, la non interferenza va intesa come assenza di scritture e aggiornamenti concorrenti sul database utilizzato dal mainframe e, più in generale, sulle risorse transazionali che costituiscono la fonte ufficiale dello stato. I microservizi coinvolti nel canale sincrono, durante la coesistenza, operano senza introdurre mutazioni sullo stato legacy e basano le proprie elaborazioni su dati letti da copie derivate mantenute separate dai database legacy e principale. In parallelo, le pipeline asincrone di ingestion possono aggiornare tali viste derivate per rendere disponibile uno stato consultabile dal to-be, ma queste scritture restano confinate allo strato di dati secondario e non interferiscono con l'operatività transazionale del mainframe.

Coerentemente con questo vincolo, la validazione parallela viene applicata alle operazioni idempotenti (tipicamente letture), confrontando l'output del legacy con l'output ricalcolato dal to-be sulle viste replicate, senza richiedere una seconda

esecuzione dispositiva né scritture sul system of record. Per le operazioni che producono mutazioni di stato (o, più in generale, non duplicabili), la non interferenza impone di evitare la riesecuzione della stessa richiesta sul percorso passivo: in tali casi la validazione avviene tramite evidenze e controlli di convergenza sulle viste derivate, e tramite un meccanismo di ghosting: le risposte prodotte dal sistema legacy e le rispettive richieste vengono salvate in un database dedicato, e i microservizi coinvolti nella fase di coesistenza utilizzano tali risposte come riferimento applicativo durante la fase di validazione, evitando di rieseguire la transazione sul proprio percorso. Il database di ghosting è quindi un archivio di evidenze a supporto della validazione durante la coesistenza e non è necessario nello scenario a regime, in cui il to-be diventa l'unico responsabile dell'elaborazione delle scritture.

L'instradamento sincrono delle richieste (*traffic splitting*) è gestito tramite una variabile pilota, che definisce in modo granulare quali operazioni o segmenti di traffico devono essere instradati verso i microservizi. Nel caso in analisi, la variabile pilota si basa sul codice ABI del conto richiedente, determinando se una richiesta venga elaborata dal legacy o dal to-be. Questo meccanismo consente di esporre inizialmente il nuovo sistema a un sottoinsieme controllato di transazioni, monitorarne il comportamento, e aumentare progressivamente il perimetro di responsabilità man mano che vengono raccolte evidenze operative affidabili. In caso di anomalie, la variabile pilota permette di reindirizzare rapidamente il traffico verso il sistema legacy, garantendo continuità operativa e riducendo il rischio complessivo della migrazione. La validazione shadow non dipende dalla variabile pilota, ma dall'attivazione dei flussi asincroni di replica/materializzazione e dei componenti di confronto.

4.5.2 Idempotenza nella fase di coesistenza

La coesistenza di flussi sincroni e asincroni durante la fase di transizione aumenta il rischio di duplicazioni nelle operazioni applicative, dovute a retry automatici, rielaborazioni di eventi o riconsegne di messaggi. Nel contesto dei sistemi di pagamento, l'idempotenza rappresenta un requisito fondamentale per garantire che operazioni ripetute producano sempre lo stesso effetto sullo stato del sistema, evitando duplicazioni di transazioni o aggiornamenti incoerenti [34], [58]. In scenari caratterizzati da elevato parallelismo e da meccanismi di recupero automatico in caso di errore, l'assenza di controlli idempotenti può generare risultati non deterministici e incrementare significativamente il rischio operativo [59].

Per mitigare tali rischi, l'implementazione dei microservizi prevede l'adozione di chiavi idempotenti associate a ciascuna richiesta o evento elaborato. Ogni operazione viene identificata univocamente tramite un identificativo e, prima di

applicare qualsiasi modifica di stato, il servizio verifica se la chiave è già stata processata, scartando eventuali duplicati [58].

L'introduzione di controlli idempotenti comporta implicazioni in termini di latenza e complessità dei servizi. Infatti, la necessità di effettuare controlli aggiuntivi per verificare la presenza di chiavi già elaborate, e di gestire strutture dati per la registrazione dello stato, può aumentare i tempi di accesso e lo spazio di archiviazione richiesto. Tuttavia, questi costi sono giustificati nei sistemi critici come quelli bancari, poiché assicurano l'integrità delle transazioni e riducono il rischio di effetti collaterali indesiderati [58], [59].

4.5.3 Sincronizzazione e coerenza dei dati

Durante la fase di coesistenza, garantire la sincronizzazione dei dati tra sistema legacy e to-be è fondamentale per mantenere coerenza e affidabilità dei processi bancari. Per prima cosa, è necessario un allineamento degli identificativi delle entità principali, come conti, clienti o transazioni. Questo viene realizzato attraverso un mapping tra identificativi legacy e to-be, definendo chiavi di correlazione che consentono di tracciare la stessa entità nei due sistemi e di confrontarne lo stato in maniera puntuale, rilevando eventuali discrepanze [2], [53]. Oltre agli identificativi, è essenziale uniformare i modelli dati dei due sistemi, definendo una mappatura esplicita che preservi l'informazione degli attributi rilevanti ai fini della validazione. Nel passaggio da architetture monolitiche a microservizi, i dati vengono spesso ristrutturati in schemi orientati al dominio, con campi derivati, aggregazioni e strutture ottimizzate [52].

Nel caso di studio, questa ristrutturazione si accompagna anche a un cambiamento tecnologico nello strato di persistenza: il database del sistema as-is è di tipo relazionale (PostgreSQL), mentre il database to-be è di tipo documentale (MongoDB). PostgreSQL, in quanto sistema relazionale conforme agli standard ACID (*Atomicity, Consistency, Isolation, Durability*), garantisce forte consistenza e integrità referenziale nelle operazioni di scrittura, risultando idoneo per la gestione delle transazioni bancarie che richiedono atomicità e isolamento rigorosi. MongoDB, invece, offre un modello documentale a schema flessibile e supporta una consistenza eventuale attraverso meccanismi di replica set, caratteristiche che lo rendono particolarmente efficace per operazioni di lettura ad alta concorrenza e per la gestione di dati semi-strutturati o soggetti a frequenti evoluzioni di schema [60].

Questa scelta implica una trasformazione cruciale del modello dati dalle strutture rigide del linguaggio COBOL (record a lunghezza fissa con campi tipizzati), ai documenti JSON memorizzati in MongoDB, che supportano tipi di dati più ricchi (array annidati, oggetti) e metadati semantici assenti nel legacy (ad esempio timestamp). Tali conversioni, gestite dai Transformer nei microservizi, richiedono denormalizzazione: contrariamente alla normalizzazione relazionale del legacy

che minimizza la ridondanza, i dati to-be incorporano direttamente relazioni per ottimizzare letture ad alta concorrenza [61].

4.5.4 Gestione operativa degli errori

La gestione operativa degli errori durante l'esecuzione parallela ha l'obiettivo primario di garantire che eventuali anomalie nel sistema to-be non compromettano la continuità del servizio, che resta affidata al sistema legacy. Trattandosi di un contesto bancario, caratterizzato da sistemi mission-critical, è importante configurare il nuovo sistema in modo che guasti locali non si propaghino, mantenendo il legacy come percorso di fallback in grado di assicurare la corretta elaborazione delle transazioni. In questo contesto, errori quali time-out, eccezioni applicative o incongruenze nei dati prodotti dai microservizi vengono rilevati e registrati; per la porzione di richieste instradata al to-be l'esito operativo è prodotto dal sistema target, mentre in caso di anomalia è previsto il fallback verso il legacy; per le richieste che restano instradate al legacy durante la coesistenza, il risultato operativo fornito al cliente continua a derivare dal sistema legacy stesso [2], [62].

Gli scenari di errore più frequenti emergono solitamente in corrispondenza dei rilasci o degli aggiornamenti del sistema to-be, necessari per verificare nuove funzionalità o correggere anomalie. In tali casi, modifiche imperfette possono provocare blocchi parziali dei microservizi, comportamenti non deterministici o incongruenze nello stato dei dati. Per prevenire impatti sull'operatività, durante la fase di coesistenza sono previsti meccanismi di rollback logico, che consentono di marcare come non valida una versione difettosa del servizio e riportare temporaneamente il traffico al solo sistema legacy fino alla risoluzione del problema.

Per limitare l'impatto degli errori e facilitare la diagnosi, vengono adottate strategie di isolamento che circoscrivono i problemi a specifici servizi o segmenti pilota. Tali strategie includono la possibilità di disattivare selettivamente singoli microservizi, escludere temporaneamente determinati flussi dal perimetro della validazione in coesistenza o ridurre la quota di traffico instradata al sistema to-be, mantenendo operativo il resto dell'architettura [2], [52].

Questi approcci operativi si combinano con meccanismi di monitoraggio e osservabilità che raccolgono metriche su tassi di errore, latenza e comportamento dei singoli servizi, permettendo di individuare rapidamente condizioni anomale e di attivare piani di mitigazione [34], [63]. In questo modo, la fase di coesistenza non solo riduce il rischio associato alla migrazione, ma fornisce un contesto controllato per testare e affinare i meccanismi di gestione degli errori e di recovery del sistema to-be, prima che esso diventi l'unico responsabile dell'elaborazione in produzione.

4.6 Validazione con quadratore

Durante la fase di coesistenza, la validazione del sistema to-be si basa su un meccanismo di confronto sistematico tra le risposte prodotte dal sistema legacy e quelle generate dai microservizi. Tale meccanismo, detto *quadratore*, esegue confronti record-to-record sui risultati delle elaborazioni, individuando eventuali discrepanze tra i due sistemi.

Nel sistema analizzato, il quadratore rappresenta il punto centrale del processo di validazione: esso raccoglie l'output del legacy e l'output calcolato dal to-be su viste replicate per le operazioni duplicabili (letture/idempotenti) e ne esegue il confronto, registrando automaticamente i casi di non coincidenza. Le differenze rilevate vengono tracciate e rese disponibili per analisi a ritroso, consentendo ai team di sviluppo di individuare errori di logica applicativa, problemi di mapping dei dati o comportamenti non deterministici e di intervenire sul codice dei microservizi. Per le operazioni non duplicabili e per le scritture, la validazione è demandata rispettivamente alle evidenze di ghosting e ai controlli di convergenza tra viste materializzate. In questo modo, la fase di coesistenza non si limita a garantire la continuità operativa, ma diventa anche uno strumento strutturato di verifica e miglioramento progressivo del sistema to-be.

Accanto al confronto puntuale delle risposte, la validazione si fonda sulla raccolta di metriche operative che guidano le decisioni di avanzamento o rollback della migrazione. L'analisi di tali metriche consente di individuare pattern di degrado prestazionale, instabilità o regressioni funzionali, fornendo evidenze oggettive per decidere se estendere il perimetro della coesistenza, stabilizzare la configurazione corrente o attivare un rollback verso il sistema legacy [34]. Le principali metriche operative utilizzate nel caso di studio vengono analizzate in dettaglio nel capitolo successivo, così da supportare il processo di migrazione con criteri misurabili e verificabili.

Capitolo 5

Valutazione del sistema to-be

L'obiettivo del presente capitolo è valutare le performance e l'affidabilità delle architetture a microservizi implementate durante la migrazione del sistema core banking verso l'architettura moderna. L'analisi si basa su metriche specifiche raccolte durante la fase di coesistenza, con l'intento di confrontare due configurazioni funzionalmente equivalenti: una sincrona (sistema legacy as-is) e una asincrona (sistema target to-be). Lo scopo è fornire evidenze quantitative sul comportamento dei sistemi in condizioni operative reali. La valutazione è stata condotta secondo il paradigma Goal-Question-Metrics (GQM), al fine di garantire un approccio strutturato e misurabile.

5.1 Goal-Question-Metric (GQM)

5.1.1 Struttura e funzionamento del framework

Nel contesto dell'ingegneria del software, la misurazione rappresenta un elemento fondamentale per il feedback sui processi, la valutazione dei prodotti e la costruzione di una memoria aziendale utile alla pianificazione e al miglioramento continuo. L'approccio *Goal Question Metric* costituisce un framework strutturato per la definizione di sistemi di misurazione considerati significativi, poiché collegati esplicitamente a obiettivi organizzativi o tecnici. Il modello adotta una struttura gerarchica articolata su tre livelli principali:

Goal (livello concettuale): obiettivo di misurazione riferito a un prodotto, processo o risorsa, che ne specifica lo scopo, l'attributo di qualità da analizzare e il punto di vista da cui la valutazione viene effettuata. Un goal GQM è quindi

caratterizzato da quattro dimensioni: *purpose*/scopo, *issue*/aspetto critico, *object*/oggetto, e *viewpoint*/punto di vista.

Question (livello operativo): l'obiettivo viene raffinato in un insieme di domande che consentono di caratterizzare l'oggetto di misurazione rispetto all'attributo di qualità selezionato. Le domande hanno lo scopo di rendere operativo l'obiettivo, individuando le informazioni necessarie per valutarlo.

Metric (livello quantitativo): a ciascuna domanda è associato un insieme di metriche che permettono di fornire risposte misurabili. Le metriche possono essere oggettive, quando dipendono esclusivamente dall'oggetto osservato, oppure soggettive, quando incorporano il punto di vista dell'osservatore, come nel caso di valutazioni di soddisfazione o percezione della qualità.

Il processo segue un approccio top-down nella fase di definizione, dagli obiettivi alle metriche, e un'interpretazione bottom-up nella fase di analisi, in cui i dati raccolti vengono letti alla luce degli obiettivi prefissati [64].

L'adozione del GQM presenta diversi vantaggi rispetto ad approcci di misurazione non guidati da obiettivi espliciti. In primo luogo, garantisce una forte focalizzazione: ogni metrica raccolta è direttamente collegata a un obiettivo, evitando la raccolta di dati non rilevanti. In secondo luogo, favorisce la contestualizzazione delle misurazioni, poiché l'interpretazione dei dati avviene in relazione al contesto organizzativo e agli attributi di qualità considerati [64].

Il framework risulta inoltre adattabile a differenti oggetti di analisi, quali prodotti, processi o risorse, e può essere applicato in contesti industriali complessi ed eterogenei. In questo senso, l'approccio GQM non si limita a definire indicatori tecnici isolati, ma consente di costruire nel tempo una memoria aziendale basata su dati empirici, utile per confrontare progetti, processi e soluzioni architetturali. Tali informazioni possono essere utilizzate per supportare la pianificazione dei progetti, attraverso stime più informate su costi, risorse e prestazioni attese. Infine, il GQM supporta il miglioramento continuo, permettendo di individuare punti di forza e debolezza dei processi e dei prodotti esistenti, e di valutare in modo quantitativo l'impatto delle scelte progettuali o delle tecniche adottate [64].

Nell'ambito di questa tesi, il metodo GQM è stato scelto poiché consente di derivare le metriche direttamente dagli obiettivi di valutazione del sistema, evitando misurazioni arbitrarie o non allineate con le esigenze del dominio applicativo. In particolare, il confronto tra le architetture sincrona e asincrona è stato definito in termini di performance e affidabilità, individuate come obiettivi principali dell'analisi, e le metriche sono state selezionate di conseguenza.

Questo approccio consente quindi di strutturare l'analisi in modo sistematico e tracciabile ed è particolarmente adatto al contesto core banking, dove le decisioni architetturali richiedono evidenze quantitative affidabili e confrontabili. Inoltre,

consente di dimostrare oggettivamente l'impatto delle scelte tecnologiche sui sistemi in condizioni operative reali, supportando le decisioni progettuali durante le diverse fasi della migrazione.

5.1.2 Design GQM

Nel caso analizzato, il goal di valutazione è stato formalizzato secondo le dimensioni del template GQM in questo modo:

- **Purpose:** confrontare
- **Issue:** performance e affidabilità
- **Object:** due architetture a microservizi (sincrona e asincrona) funzionalmente equivalenti
- **Viewpoint:** dal punto di vista dei responsabili della qualità

GOAL: confrontare performance e affidabilità di due architetture a microservizi equivalenti dal punto di vista dei responsabili della qualità, nel contesto di un sistema core banking ad alta frequenza transazionale.

Le domande guida della valutazione formulate a partire da esso sono le seguenti:

1. Quali sono le performance dei microservizi?
2. Qual è l'affidabilità dei microservizi?

Seguendo le linee guida del metodo GQM, ciascuna domanda è stata quindi associata a un insieme di metriche quantitative, misurabili in modo consistente sugli stessi endpoint applicativi per entrambe le architetture. La valutazione si focalizza quindi su due dimensioni principali:

- **Performance**, misurata attraverso processing time medio, CPU time medio, latenza media e throughput.
- **Affidabilità**, misurata tramite failure rate ed exception count.

5.1.3 Raccolta dei dati

La raccolta dei dati è stata condotta su quattro core rappresentativi del dominio funzionale del sistema core banking, considerati sia nella configurazione legacy (as-is) sia nella nuova architettura (to-be). La valutazione si basa su un approccio osservazionale in condizioni reali di produzione. Gli oggetti di misura sono quindi quattro core logici, ciascuno realizzato sia nell'architettura sincrona as-is sia in quella asincrona to-be ed esposto come endpoint funzionalmente equivalente:

- **Core A** – servizio di consultazione integrata di saldo e movimenti
- **Core B** – servizio specializzato nella consultazione delle transazioni
- **Core C** – servizio di consultazione dei conti
- **Core D** – servizio di consultazione del saldo

Le metriche sono state raccolte mediante la piattaforma di *Application Performance Monitoring* Dynatrace, adottata dall'organizzazione per l'osservabilità end-to-end di applicazioni, microservizi e infrastrutture IT. La piattaforma consente di mappare automaticamente le dipendenze tra i servizi, tracciare le singole transazioni e monitorare in tempo reale i microservizi, identificando le origini delle anomalie prestazionali, oltre a registrare indicatori di performance e affidabilità. Per ciascun core, i valori sono stati estratti con granularità a livello di singolo controller applicativo, assicurando che il confronto tra sistema as-is e to-be fosse effettuato su componenti funzionalmente equivalenti.

5.1.4 Finestra temporale e volume di dati

È importante sottolineare che i dati sono stati raccolti nell'arco di sei giornate distinte di esercizio. La distribuzione delle rilevazioni su più giorni consente di attenuare la dipendenza da condizioni operative puntuali, catturando una variabilità del carico più rappresentativa del comportamento tipico del sistema in produzione. Il periodo di osservazione disponibile risulta tuttavia limitato in ragione delle vicende operative del sistema: in una fase precedente della migrazione, il meccanismo di esecuzione parallela era stato temporaneamente disabilitato, circostanza che non ha consentito di raccogliere misurazioni comparative su un intervallo temporale più esteso. Solo in seguito alla sua riattivazione è stato possibile acquisire i dati qui analizzati, sebbene in una finestra temporale ancora contenuta. L'analisi su un arco temporale ristretto costituisce quindi una limitazione dal punto di vista quantitativo, poiché riduce la possibilità di osservare la variabilità delle metriche su periodi più ampi e ne limita la rappresentatività statistica in scenari di carico prolungato. Tuttavia, tali condizioni consentono comunque di effettuare un confronto preliminare tra le due architetture, evidenziandone le principali tendenze prestazionali e comportamentali. Per questo motivo, i risultati presentati vanno interpretati come evidenze preliminari riferite a una specifica fase della migrazione, ma da consolidare su un arco temporale più ampio e a regime completo.

5.1.5 Rappresentazione grafica

I risultati delle misurazioni sono stati rappresentati tramite grafici a barre raggruppate comparativi, realizzati con la libreria Python Matplotlib, nei quali per ciascuna

metrica analizzata, ad eccezione del throughput, vengono posti a confronto, per ognuno dei quattro core applicativi, i valori medi pesati delle metriche misurate sul sistema sincrono (as-is) e asincrono (to-be). Tale scelta consente di effettuare un confronto visivo diretto tra le due architetture per ciascun core.

Poiché le misurazioni sono associate a volumi di richieste differenti, il valore rappresentato nelle barre non corrisponde alla media aritmetica semplice, bensì alla media pesata sul numero di richieste processate in ciascuna finestra temporale. Formalmente, per una metrica x misurata nelle finestre $i = 1, \ldots, 6$ con peso n_i, pari al numero di richieste nella finestra i, la media pesata è definita come:

$$\bar{x}_w = \frac{\sum_{i=1}^{6} n_i \cdot x_i}{\sum_{i=1}^{6} n_i} \tag{5.1}$$

Questo approccio è stato adottato in modo da rispettare la natura dei dati: finestre con volumi di traffico più elevati forniscono stime più stabili e rappresentative della metrica e devono pertanto contribuire in misura maggiore al valore aggregato, mentre la media aritmetica semplice tratterebbe ogni finestra allo stesso modo, introducendo una distorsione sistematica verso valori atipici. I pesi n_i differiscono tra as-is e to-be poiché, durante la fase di migrazione osservata, solo una quota controllata del traffico produttivo viene instradata verso il nuovo sistema, mentre parte della validazione avviene in shadow mode o tramite meccanismi indiretti di confronto, rendendo fisiologicamente diversi i volumi di richieste processate dai due sistemi in ciascuna finestra di osservazione.

5.1.6 Metriche

Metriche di performance:

Processing time medio: tempo medio di elaborazione di ciascuna richiesta all'interno del servizio. Riflette l'efficienza interna della logica applicativa e del modello di comunicazione sincrono o asincrono lungo la catena di chiamate.

CPU time medio: tempo medio di utilizzo della CPU per richiesta. Consente di correlare eventuali miglioramenti prestazionali con l'impiego delle risorse hardware.

Latenza media: tempo medio percepito tra l'invio della richiesta e la ricezione della risposta, comprensivo dell'elaborazione applicativa e dei tempi di comunicazione interna.

Throughput: numero di richieste elaborate nell'unità di tempo (richieste al minuto). Misura la capacità del servizio di sostenere carichi elevati.

Metriche di affidabilità:

Failure rate: percentuale di richieste non andate a buon fine rispetto al totale delle richieste elaborate nella finestra temporale considerata. Rappresenta il tasso di fallimento delle richieste.

Exception count: numero di eccezioni applicative registrate nel periodo di osservazione, normalizzato rispetto alla finestra temporale analizzata. Utilizzato come indicatore di anomalie ed errori applicativi.

5.2 Analisi dei dati

5.2.1 Analisi delle performance

5.2.1.1 Processing time e latenza

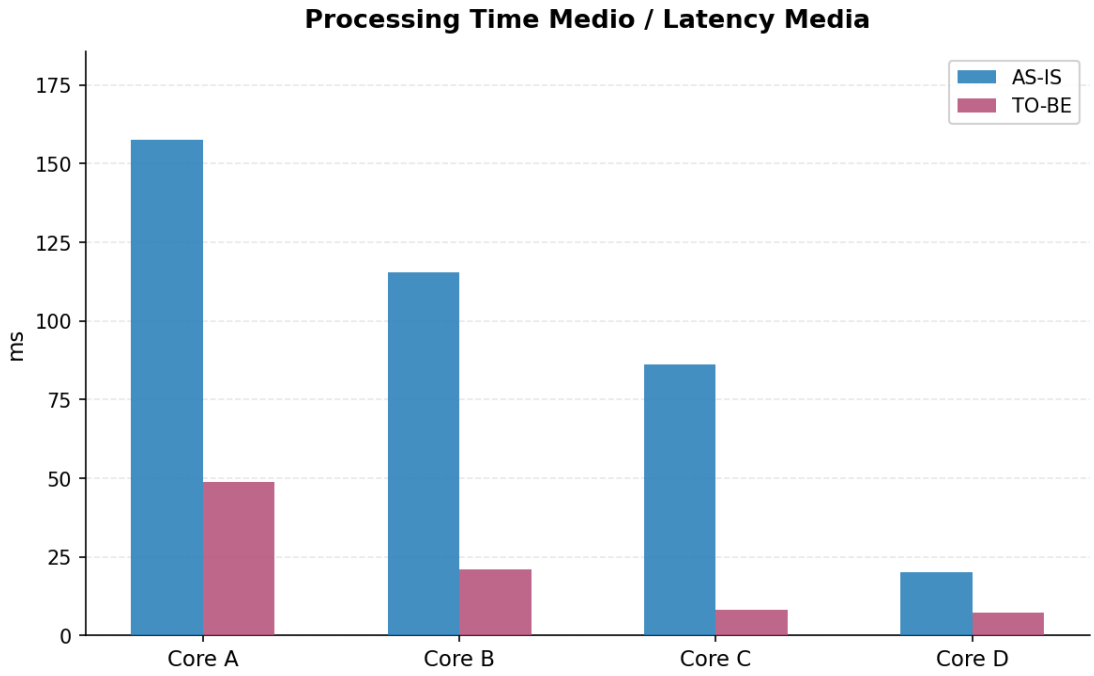

Figura 5.1: Confronto di processing time/latency tra configurazioni as-is e to-be per i quattro core bancari.

I valori medi di processing time e latenza risultano coincidenti in tutti i core analizzati, poiché le metriche sono state rilevate a livello di singolo controller applicativo, dove il tempo di elaborazione interno costituisce la componente dominante

del tempo di risposta complessivo, rendendo le due misure sostanzialmente equivalenti. Per tale motivo, le due metriche sono state rappresentate congiuntamente in un unico grafico.

Il grafico mostra un miglioramento complessivo delle prestazioni nella configurazione asincrona per tutti i quattro core analizzati. L'ottimizzazione più significativa si osserva nel Core C, che gestisce il maggiore volume di richieste, dove il processing time e la latenza registrano una riduzione superiore al 90%. Anche nel Core B il miglioramento risulta particolarmente marcato, con una riduzione di oltre l'80%, mentre nel Core A si osserva una diminuzione superiore al 60%. Il Core D presenta a sua volta un miglioramento significativo, sebbene più contenuto in valore relativo rispetto agli altri core, in quanto partiva già da tempi di risposta sensibilmente inferiori nella configurazione as-is. I valori sono stati calcolati come media pesata sul numero di richieste per finestra temporale, per tenere conto dell'eterogeneità del carico. Nelle finestre osservate, i valori misurati sul sistema to-be mostrano una variabilità inferiore rispetto all'as-is. Tale andamento è coerente con la natura disaccoppiata dell'architettura asincrona, sebbene debba essere interpretato con cautela, poiché il sistema to-be è stato esercitato con un volume di traffico inferiore durante la fase di coesistenza: solo una quota controllata del traffico viene instradata direttamente al nuovo sistema, mentre parte della validazione avviene in shadow mode o tramite meccanismi indiretti di confronto.

Nel complesso, i risultati evidenziano una riduzione netta dei tempi di risposta nella configurazione to-be, suggerendo l'efficacia dell'architettura asincrona, con benefici particolarmente evidenti nei core caratterizzati da operazioni di consultazione più semplici.

5.2.1.2 CPU time

L'analisi del CPU time medio evidenzia un andamento differenziato tra i core.

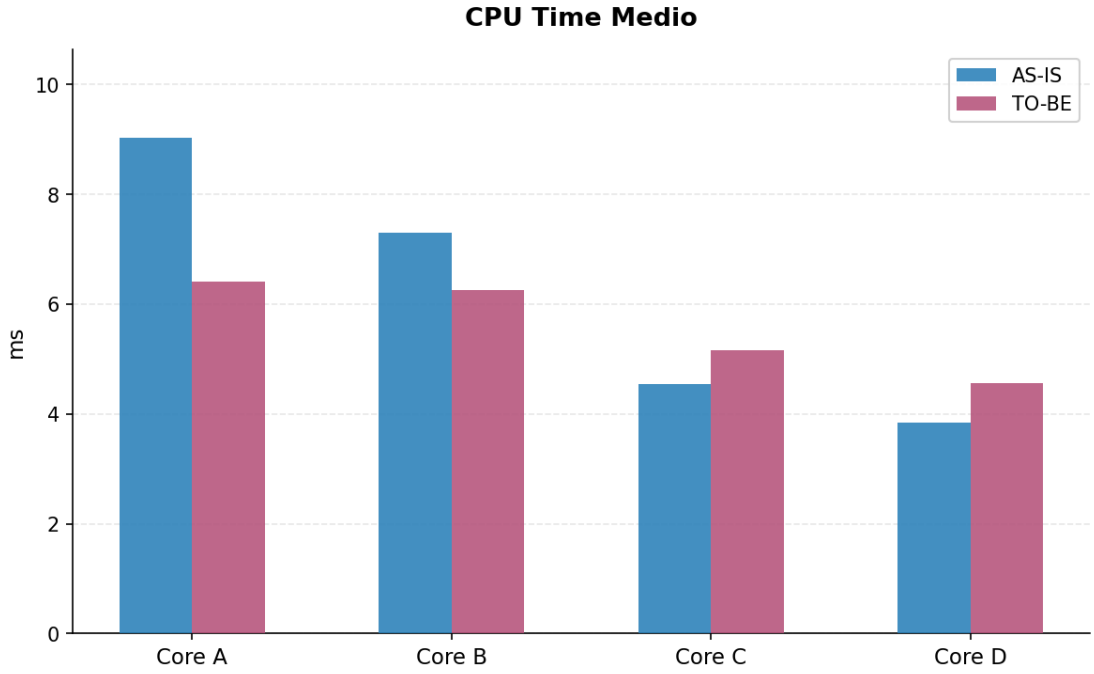

Figura 5.2: Confronto del CPU time tra configurazioni as-is e to-be per i quattro core bancari.

Nei Core C e D si osserva un aumento del tempo medio di utilizzo della CPU (+13,9% e +18,9%), suggerendo un maggiore impiego delle risorse computazionali nella configurazione asincrona, plausibilmente riconducibile all'overhead introdotto dal layer di messaging asincrono, in particolare alle attività di produzione e consumo degli eventi e alla maggiore complessità della pipeline di elaborazione.

Nei Core A e B, invece, il CPU time si riduce, rispettivamente del 29,0% e del 14,3%, indicando un miglioramento nell'efficienza dell'elaborazione, coerente con la natura asincrona delle operazioni, che riduce il tempo in cui i thread restano vincolati ad attese bloccanti.

Nel complesso, l'architettura asincrona non comporta un aumento sistematico del consumo di CPU: in alcuni casi si osserva una riduzione, mentre in altri il miglioramento dei tempi di risposta è accompagnato da un maggiore utilizzo delle risorse computazionali.

5.2.1.3 Throughput

Per quanto riguarda il throughput, i dati raccolti non consentono di formulare un confronto diretto tra as-is e to-be in termini di miglioramento prestazionale, poiché i due sistemi hanno operato su volumi di traffico significativamente differenti

nelle finestre temporali analizzate. In particolare, il sistema to-be ha gestito una quota di richieste compresa tra l'1,335% e il 2,319% rispetto al sistema as-is nei quattro core analizzati, condizione che non consente di interpretare i valori assoluti di throughput come indicatore direttamente comparabile della capacità elaborativa delle due architetture. Le medie pesate calcolate sulle sei finestre di osservazione mostrano:

	as-is	**to-be**
Core A	4 174 k/min	67,8/min
Core B	1 816 k/min	39,9/min
Core C	25 355 k/min	526,1/min
Core D	10 554 k/min	128,1/min

Tabella 5.1: Throughput medio pesato per core (richieste/minuto).

Il confronto tra i valori medi pesati mostra che il throughput osservato del sistema to-be risulta sistematicamente inferiore a quello dell'as-is in tutti i core analizzati. Tale andamento è coerente con il fatto che, durante la fase di coesistenza, il sistema to-be ha ricevuto soltanto una quota limitata del traffico complessivo, e riflette pertanto principalmente la differente numerosità delle richieste elaborate nei due ambienti più che una diversa capacità prestazionale intrinseca. In questo senso, i valori osservati descrivono pertanto il carico effettivamente gestito dai due ambienti nella specifica configurazione di migrazione, ma non permettono di inferire in modo conclusivo un incremento o una riduzione della capacità elaborativa del sistema to-be rispetto all'as-is. Eventuali benefici del nuovo assetto architetturale devono quindi essere ricondotti principalmente alla riduzione dei tempi di risposta e non al throughput osservato nelle presenti misurazioni.

Da un punto di vista teorico, l'architettura asincrona presenta caratteristiche strutturalmente favorevoli alla scalabilità. Nel modello sincrono, infatti, ogni richiesta tende a occupare un thread per l'intera durata dell'operazione: il thread non esegue soltanto la logica applicativa, ma rimane associato alla richiesta anche durante le fasi di attesa di I/O, come accessi a database, rete o servizi esterni. In tali condizioni, il numero di richieste concorrenti gestibili risulta fortemente vincolato dalla dimensione del thread pool e dal tempo medio di permanenza di ciascun thread nello stato di attesa. Nei modelli asincroni con I/O non bloccante, al contrario, l'attesa dell'operazione non impegna il thread nello stesso modo: una volta avviata l'operazione di I/O, il controllo può essere restituito al runtime, consentendo al thread di essere riutilizzato per altre attività fino alla disponibilità del risultato. Questo meccanismo permette, a parità di risorse disponibili, un utilizzo più efficiente dei thread e una migliore capacità di sostenere elevati livelli di concorrenza. Nel caso del layer di messaging, inoltre, la scalabilità può essere

ulteriormente incrementata in senso orizzontale mediante l'aumento del numero di partizioni Kafka e delle istanze consumer [65].

5.2.2 Analisi dell'affidabilità

5.2.2.1 Failure rate

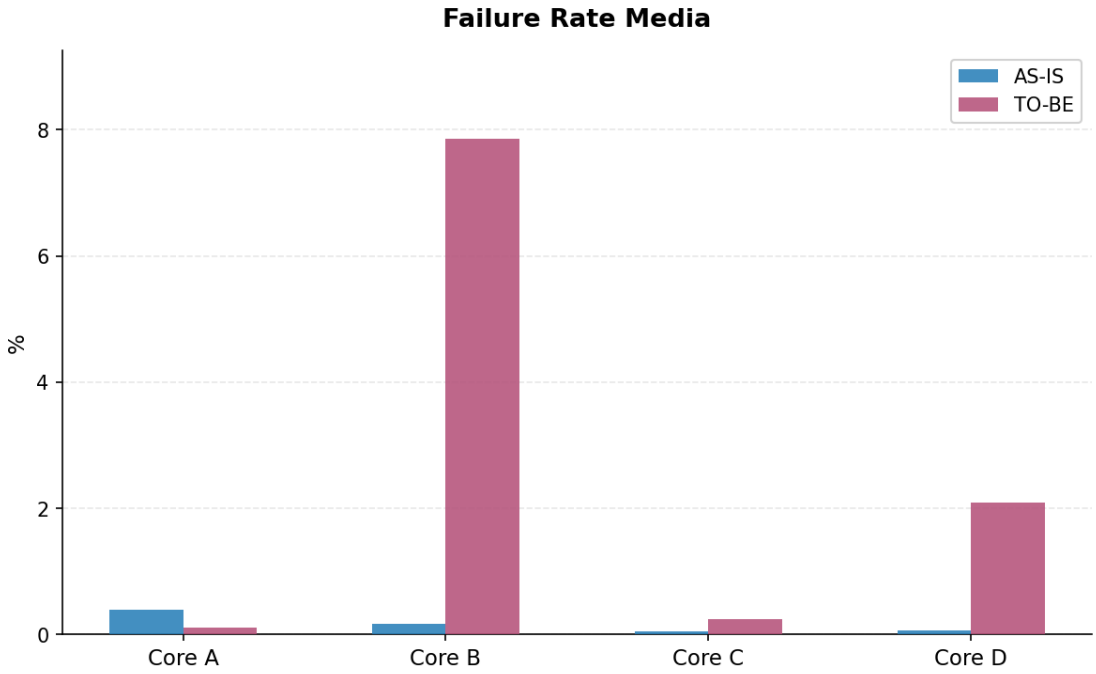

Figura 5.3: Confronto del failure rate tra configurazioni as-is e to-be per i quattro core bancari.

L'analisi del failure rate evidenzia comportamenti differenti tra i quattro core. Il Core A è l'unico che mostra una riduzione netta (−73,3%), indicando un miglioramento dell'affidabilità nella configurazione asincrona. Per i restanti core, il sistema to-be registra valori significativamente più elevati: il Core B passa da 0,170% a 7,852%, il Core C da 0,047% a 0,240% e il Core D da 0,071% a 2,086%.

Tali incrementi non devono tuttavia essere interpretati automaticamente come una regressione funzionale dell'architettura. In un sistema event-driven basato su Kafka, il failure rate può includere non soltanto errori applicativi, ma anche eventi legati al layer di messaging, quali errori di deserializzazione, problemi di validazione, timeout, fallimenti del consumer, retry esauriti e inoltro dei messaggi verso *Dead Letter Queue* (DLQ). La DLQ rappresenta infatti un topic dedicato in cui vengono instradati i messaggi che non possono essere elaborati con successo dopo i tentativi

previsti, così da evitare il blocco della pipeline e consentirne l'analisi o il successivo reprocessing [66]. A differenza del sistema sincrono, dove un timeout era spesso silente, il sistema asincrono rende visibili condizioni di errore precedentemente non tracciate. Inoltre, al momento della raccolta dei dati il sistema asincrono non operava ancora a regime, l'aumento potrebbe quindi essere riconducibile a configurazioni non totalmente stabilizzate o a errori temporanei, piuttosto che a un limite strutturale dell'architettura.

5.2.2.2 Exception count

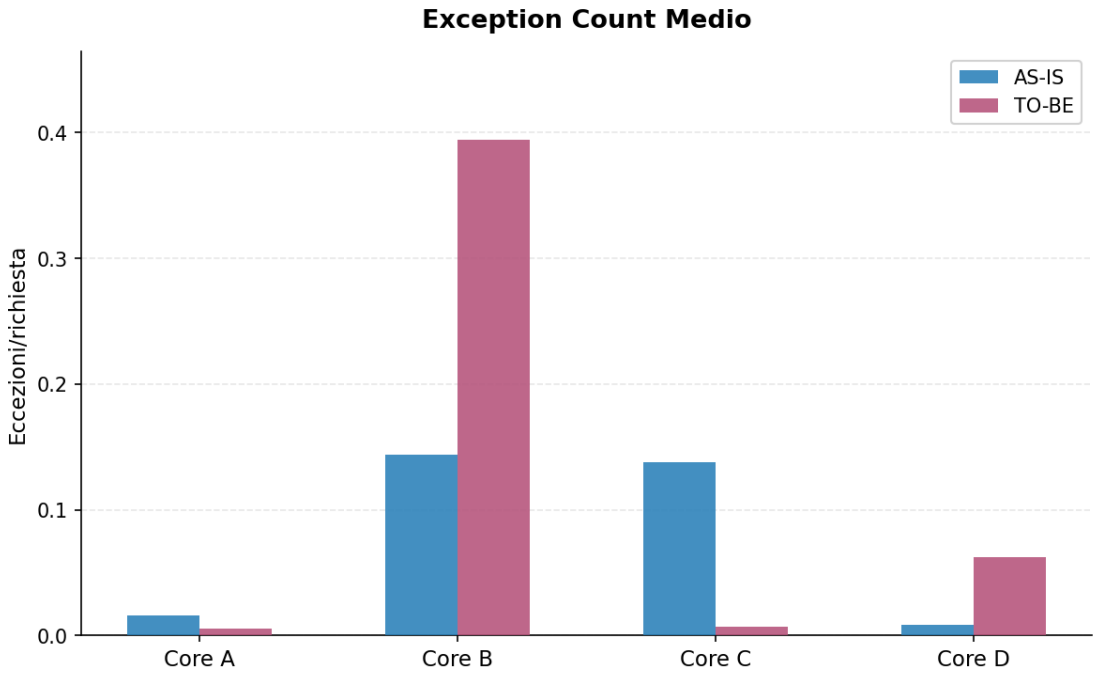

Figura 5.4: Confronto dell'exception count tra configurazioni as-is e to-be per i quattro core bancari.

L'analisi dell'exception count evidenzia comportamenti eterogenei tra i core. Nei Core B e D esso cresce in modo significativo, coerentemente con l'elevato tasso di fallimento delle richieste osservato. Al contrario, il Core C mostra il miglioramento più marcato, con una riduzione del 94,8%, mentre il Core A registra anch'esso una diminuzione significativa (−66,6%).

Per il Core C, il risultato appare in contrasto con l'aumento del failure rate osservato nello stesso core. Tuttavia, le due metriche misurano layer distinti del sistema: l'exception count riflette le eccezioni generate nel codice applicativo durante l'elaborazione, mentre il failure rate può includere eventi di fallimento a

livello di messaging, indipendentemente dall'esecuzione del codice [34]. Nel caso del Core C, la riduzione delle eccezioni applicative suggerisce che il consumer asincrono gestisce i casi limite in modo più controllato rispetto al sistema sincrono, mentre l'aumento del failure rate riflette una maggiore visibilità degli eventi transitori nel layer Kafka. I due fenomeni sono quindi complementari, non contraddittori.

Nel complesso, il confronto tra failure rate ed exception count evidenzia come la migrazione a un'architettura asincrona possa produrre effetti differenziati sui due layer del sistema, rendendo necessaria un'analisi congiunta delle metriche per una valutazione accurata dell'affidabilità complessiva.

5.2.3 Sintesi comparativa per core

Nel **Core A**, l'architettura asincrona mostra il quadro complessivamente più equilibrato. Si osserva una riduzione significativa dei tempi di risposta, accompagnata anche da una diminuzione del CPU time. Dal punto di vista dell'affidabilità, il core registra inoltre un miglioramento netto, con riduzione sia del failure rate sia dell'exception count. Nel complesso, il Core A rappresenta il caso in cui il passaggio all'architettura asincrona produce benefici simultanei in termini di performance, efficienza computazionale e affidabilità.

Nel **Core B**, il sistema to-be evidenzia un miglioramento prestazionale molto marcato, con una riduzione del processing time e della latenza, accompagnate da una diminuzione del CPU time. Tuttavia, tali benefici sono accompagnati da un netto peggioramento delle metriche di affidabilità, con incremento significativo sia del failure rate sia dell'exception count. Il caso del Core B evidenzia quindi un trade-off rilevante: l'architettura asincrona migliora sensibilmente i tempi di risposta, ma presenta ancora criticità applicative o di configurazione che ne compromettono la robustezza operativa nella fase osservata.

Nel **Core C**, invece, si osserva il miglioramento prestazionale più significativo tra tutti i core analizzati. Il processing time e la latenza si riducono di oltre il 90%, mentre il throughput osservato, pur non essendo direttamente comparabile con l'as-is, conferma che il core continua a rappresentare quello a maggiore volume di traffico. Il CPU time cresce moderatamente, suggerendo un maggiore impiego delle risorse computazionali nella configurazione asincrona, plausibilmente associato all'overhead introdotto dal layer di messaging e alla maggiore articolazione della pipeline di elaborazione [8]. Dal punto di vista dell'affidabilità, il comportamento appare articolato: da un lato il failure rate aumenta, dall'altro l'exception count si riduce drasticamente, suggerendo una maggiore stabilità del layer applicativo nelle operazioni ad alto volume. Tale divergenza è coerente con la diversa semantica delle due metriche: il failure rate può includere eventi di mancata consegna e timeout registrati a livello di broker Kafka, mentre l'exception count riflette esclusivamente le eccezioni lanciate durante l'elaborazione [34]. Nel complesso, il Core C rappresenta

il caso in cui l'architettura asincrona produce i benefici prestazionali più evidenti, pur richiedendo un ulteriore affinamento del layer infrastrutturale.

Nel **Core D**, infine, il sistema to-be mostra una riduzione significativa dei tempi di risposta, ma tale miglioramento è accompagnato da un aumento del CPU time e da un peggioramento marcato delle metriche di affidabilità. Il Core D evidenzia pertanto un comportamento meno favorevole rispetto agli altri casi: pur beneficiando della riduzione dei tempi di risposta, il nuovo assetto architetturale presenta ancora un livello di stabilità insufficiente, che suggerisce la necessità di un ulteriore affinamento della logica applicativa e della configurazione infrastrutturale.

5.3 Discussione dei risultati

Nel complesso, l'analisi mostra benefici prestazionali significativi dell'architettura to-be asincrona rispetto alla configurazione as-is sincrona. In tutti e quattro i core analizzati si osserva infatti una riduzione netta del processing time e della latenza, con miglioramenti compresi tra il 64,5% e il 90,5%. Tali risultati indicano una maggiore reattività del sistema to-be e sono coerenti con i benefici attesi da un'architettura disaccoppiata basata su comunicazione asincrona. L'andamento del CPU time risulta invece differenziato tra i core. Nei Core A e B si osserva una riduzione del tempo medio di utilizzo della CPU, mentre nei Core C e D emerge un incremento moderato, plausibilmente associato all'overhead introdotto dal layer di messaging e dalla maggiore articolazione della pipeline di elaborazione, ma complessivamente compatibile con le aspettative per un'architettura asincrona. Ne consegue che l'architettura asincrona non determina un effetto univoco sul consumo di CPU: in alcuni casi migliora l'efficienza dell'elaborazione, in altri trasferisce parte del costo computazionale su componenti differenti del flusso.

Dal punto di vista dell'affidabilità, i risultati sono eterogenei. Il Core A mostra un miglioramento netto delle metriche, configurandosi come il caso più maturo e bilanciato; i Core B e D presentano invece un peggioramento marcato, suggerendo la presenza di criticità ancora aperte nella configurazione asincrona; il Core C mostra infine un comportamento misto. Tuttavia, tali valori devono essere interpretati alla luce del contesto operativo in cui sono stati raccolti i dati, caratterizzato da una migrazione non ancora a regime. In queste condizioni, eventuali squadrature tra i due rami architetturali, configurazioni non allineate o errori temporanei di orchestrazione possono aver influenzato negativamente alcune metriche, senza rappresentare necessariamente un limite strutturale dell'architettura asincrona.

In relazione alla domanda Q1 (performance), le evidenze risultano coerenti con la letteratura che associa l'adozione di architetture event-driven a una riduzione dei tempi di risposta medi e a un miglior sfruttamento delle risorse in scenari ad alto volume transazionale [9]. Per quanto riguarda la domanda Q2 (affidabilità), la

presenza di pattern differenziati tra i core è in linea con gli studi che evidenziano come la migrazione a microservizi richieda un progressivo affinamento delle politiche di gestione degli errori, soprattutto in presenza di configurazioni di coesistenza [34].

È tuttavia necessario considerare le limitazioni del dataset analizzato. La raccolta dei dati su una finestra temporale limitata a sei giornate di osservazione, non ancora pienamente rappresentativa di una condizione di esercizio a regime, limita la rappresentatività statistica delle metriche osservate. I risultati devono quindi essere interpretati come indicativi delle tendenze emergenti piuttosto che come stime robuste del comportamento del sistema nel lungo periodo.

Nel complesso, i dati suggeriscono quindi un trend positivo del sistema to-be: la migrazione verso un'architettura asincrona introduce miglioramenti prestazionali evidenti, pur in presenza di criticità di affidabilità ancora da consolidare in alcuni core. Saranno tuttavia necessarie ulteriori campagne di raccolta dati in condizioni operative stabili e a pieno carico per validare in modo definitivo i benefici osservati e confermare la stabilità del sistema nel lungo periodo.

5.3.1 Possibili estensioni

Dal punto di vista metodologico, una prima estensione naturale della valutazione consisterebbe nell'ampliare la raccolta delle stesse metriche su un orizzonte temporale più lungo, includendo un numero maggiore di giornate di osservazione e periodi operativi differenziati. Ciò permetterebbe di stimare in modo più robusto la variabilità temporale delle metriche, ridurre l'influenza di condizioni contingenti e consolidare le evidenze empiriche emerse in questa prima analisi.

Una seconda estensione riguarderebbe la possibilità di condurre prove in condizioni di carico maggiormente controllate e comparabili tra as-is e to-be. In particolare, test di carico a parità di volume di richieste consentirebbero di valutare in modo più rigoroso la capacità elaborativa e il throughput delle due architetture, superando il principale limite della fase di coesistenza, nel quale il traffico instradato verso il sistema to-be rappresenta soltanto una frazione di quello gestito dal sistema legacy. Sarebbe inoltre opportuno approfondire l'analisi dell'affidabilità mediante una classificazione più granulare delle cause di failure, distinguendo ad esempio tra errori applicativi, timeout, eventi di retry esauriti, fallimenti del consumer e inoltro verso DLQ. Studi sperimentali su architetture Spring Boot a microservizi mostrano infatti che, sebbene a basso carico modelli sincroni e asincroni possano risultare comparabili, con l'aumentare del carico le architetture event-driven basate su Kafka ottengono latenza inferiore, throughput più alto, scalabilità quasi lineare e maggiore resilienza grazie allo storage duraturo dei messaggi e ai meccanismi di consumer group [67].

Capitolo 6

Limitazioni e possibili miglioramenti futuri

Il presente capitolo ha l'obiettivo di esplicitare i principali limiti emersi nel percorso di modernizzazione descritto e, a partire da essi, delineare un insieme di possibili miglioramenti futuri di natura architetturale, operativa e metodologica.

L'approccio adottato presenta infatti limiti tecnici intrinseci e vincoli contestuali che ne circoscrivono l'estendibilità, nonostante risultati empirici solidi. L'obiettivo della discussione non è sminuire i risultati ottenuti, bensì chiarire in quali condizioni le evidenze osservate possano essere considerate generalizzabili e in quali, invece, permanga un rischio residuo significativo. In un dominio ad alta criticità come il core banking, tale analisi consente di individuare gli interventi prioritari per rendere l'approccio più robusto, scalabile e sostenibile nel lungo periodo.

6.1 Limitazioni

6.1.1 Disallineamenti tra basi dati

Un limite strutturale dell'impostazione adottata risiede nella possibile mancanza di allineamento perfetto, in ogni istante, tra le diverse basi dati introdotte per supportare coesistenza e validazione.

Nel caso specifico, la coesistenza di un DB legacy contrapposto a un DB principale e un DB replica può produrre divergenze temporanee dovute a latenza di propagazione, trasformazioni applicate lungo la catena di eventi e differenze di modello dati (normalizzato vs denormalizzato) che impattano la ricostruzione dell'informazione. A ciò si aggiunge la presenza di un DB di ghosting per le operazioni non duplicabili o non riproducibili in shadow mode, che crea per definizione un

perimetro informativo asimmetrico e può generare differenze "legittime" tra dataset utilizzati da legacy e to-be.

In aggiunta, le discrepanze osservabili non derivano necessariamente da errori logici, ma possono dipendere dal fatto che la replica non dispone dello stesso livello di storico o della stessa profondità informativa del sistema sorgente. Ciò può dipendere da scelte di perimetro (ossia quali entità/attributi vengono replicati e con quale granularità), da politiche di retention differenti o da modalità di caricamento iniziale non completamente allineate; a questi fattori si somma la latenza di propagazione tra sorgente e replica. Un'ulteriore causa è l'asimmetria tra modelli dati: a fronte di record legacy rigidi e fortemente tipizzati, il sistema to-be utilizza documenti denormalizzati, con conseguenti differenze nella rappresentazione e nella ricostruzione dell'informazione. Questo aspetto è particolarmente rilevante per funzionalità di consultazione che richiedono informazioni retrospettive (ricerche su finestre temporali ampie, ricostruzioni di movimenti, ricalcoli su base storica) e può introdurre differenze tra output legacy e output to-be anche a parità di richiesta.

Dal punto di vista della validazione in fase di coesistenza, tali fenomeni complicano il confronto di equivalenza funzionale: diventa necessario distinguere in modo sistematico tra divergenze attese perché dovute a dataset differenti o incompleti e divergenze anomale perché dovute a regressioni o bug del nuovo sistema. Ne consegue che, senza meccanismi espliciti di riconciliazione e senza un contratto dati chiaro tra le viste, il rischio è di accumulare segnalazioni difficili da interpretare, aumentando costi di analisi e tempi di stabilizzazione.

6.1.2 Garanzie di correttezza nel paradigma asincrono

La migrazione da un modello sincrono a un modello asincrono modifica il punto in cui vengono "posizionate" alcune garanzie di correttezza: nel legacy molte proprietà sono supportate nativamente dall'ambiente transazionale (UOW, syncpoint, ACID, locking), mentre nell'architettura event-driven una parte rilevante della correttezza viene spostata dal middleware transazionale a meccanismi applicativi (idempotenza, gestione dei retry, deduplicazione, ordinamento, gestione degli errori).

Questo comporta un limite intrinseco: la correttezza globale non è più garantita soltanto dal commit su una singola risorsa, ma dipende dall'interazione tra più componenti, consumer e stream di eventi, con potenziali effetti collaterali in caso di riconsegne, rielaborazioni o errori parziali. In particolare, durante una fase di coesistenza, in cui due implementazioni coesistono e l'instradamento del traffico può essere progressivo, ogni incoerenza transitoria o duplicazione può riflettersi in mismatch tra output, anche quando l'esperienza utente resta protetta perché la risposta viene ancora servita dal legacy (fase di validazione) [68].

Nel caso in esame, la validazione separa implicitamente due piani: per le letture, il confronto avviene tra risposta legacy e risposta ricostruita interrogando il DB replica, mentre per le scritture la correttezza viene osservata tramite convergenza tra viste materializzate (DB principale vs DB replica). Questa impostazione evidenzia come la correttezza del to-be sia misurata in modo indiretto e a posteriori, e risente quindi delle latenze e delle asimmetrie informative tra basi dati, oltre che della complessità delle semantiche di consegna e dei retry tipici di un sistema asincrono. Ne deriva che, per mantenere affidabile la validazione, è necessario formalizzare i criteri di equivalenza, altrimenti si rischia di confondere differenze architetturali fisiologiche con regressioni.

Un'ulteriore fonte di complessità è l'evoluzione degli eventi e dei relativi schemi, perché nel paradigma event-driven l'evento costituisce un contratto tra producer e consumer che tipicamente evolvono e vengono rilasciati in momenti diversi. Poiché gli eventi restano persistiti su Kafka e possono essere riprocessati, è possibile che consumer vecchi continuino a leggere eventi prodotti con payload aggiornati (o viceversa), con rischi di incompatibilità di deserializzazione e, soprattutto, di interpretazione semantica (stesso campo, significato diverso). Di conseguenza, oltre a idempotenza e gestione dei retry, diventa cruciale garantire compatibilità evolutiva del payload (versioning e regole di backward/forward compatibility), così che producer e consumer rilasciati in tempi diversi possano coesistere senza interrompere elaborazioni e viste materializzate [69].

6.1.3 Complessità operativa e coordinamento

La coesistenza è un pilastro della strategia incrementale ma introduce un carico operativo non trascurabile, poiché impone di gestire due sistemi in parallelo e di coordinare routing deterministico, criteri di rollback e osservazione continua di anomalie. In pratica, si devono monitorare simultaneamente due sistemi, interpretare metriche di entrambi e intervenire rapidamente quando emergono divergenze, con un aumento della complessità di troubleshooting e del rischio di errori procedurali.

In parallelo, la comunicazione con gli stakeholder è una componente critica: dirigenti e responsabili dei processi devono comprendere la timeline, i rischi e le contingenze, perché le decisioni su stabilizzazione, incremento del traffico o cutover hanno impatti diretti sulla continuità del servizio. Se questa comunicazione non è strutturata, si rischia di rallentare la migrazione per eccesso di cautela o, al contrario, di sottovalutare segnali deboli di regressione, con conseguenze potenzialmente gravi.

6.1.4 Limiti metodologici della valutazione

La valutazione empirico-sperimentale basata su GQM e su metriche raccolte in fase di coesistenza rappresenta un punto di forza, ma la sua interpretazione dipende

dalla rappresentatività del campione: numero di servizi analizzati, varietà dei casi d'uso, copertura di condizioni operative e presenza di carichi anomali o scenari limite. Nel caso di questa sperimentazione, il campione è limitato e concentrato su tipologie funzionali omogenee; di conseguenza, i risultati non sono automaticamente generalizzabili all'estensione del sistema verso altri domini o servizi con caratteristiche differenti.

Inoltre, la raccolta delle metriche è avvenuta su una finestra temporale ristretta e in un contesto operativo non completamente a regime, riducendo la possibilità di osservare variabilità e stabilità delle misure su periodi più ampi e a pieno carico. Infine, la fase di coesistenza tende a privilegiare criteri di sicurezza operativa e non interferenza, limitando la sperimentazione di pratiche come failure injection, test di resilienza aggressivi o cutover ripetuti, che sarebbero utili per stimare la robustezza del sistema a regime.

6.2 Miglioramenti futuri

6.2.1 Consolidamento della piattaforma e qualità del codice

Un'evoluzione naturale del lavoro consiste nel consolidare la piattaforma to-be non solo come insieme di microservizi, ma come prodotto interno riutilizzabile, riducendo la variabilità implementativa tra servizi e rendendo più prevedibili tempi e rischi di evoluzione. In questa direzione, un miglioramento rilevante è la definizione di un percorso standard di sviluppo e rilascio (template, librerie e convenzioni) che standardizzi aspetti trasversali quali logging strutturato, metriche, tracing distribuito, gestione degli errori e integrazione Kafka, così da rendere l'architettura più omogenea e ridurre i costi di manutenzione e di formazione iniziale [46].

Dal punto di vista della manutenibilità, tale standardizzazione consente inoltre di rendere sistematiche pratiche che, se lasciate alla discrezione dei singoli team, tendono a degradare nel tempo. Questo approccio semplifica la revisione del codice e rende più controllabile l'evoluzione degli schemi evento, che in architetture event-driven costituiscono un contratto persistente e riprocessabile [46].

Oltre alla standardizzazione, un'ulteriore linea di evoluzione riguarda l'adozione di pratiche sistematiche di miglioramento continuo del codice (refactoring guidato da evidenze), poiché l'evoluzione incrementale del to-be e l'estensione progressiva del perimetro tendono a evidenziare aree di debito tecnico. Nel contesto di coesistenza analizzato, tali interventi possono essere prioritizzati usando segnali oggettivi raccolti in esercizio (pattern ricorrenti di eccezioni, mismatch del quadratore) e focalizzandosi su componenti trasversali ad alta riusabilità, così da ottenere benefici sistemici su più microservizi.

6.2.2 Progressive delivery

Dal punto di vista del rilascio in produzione, l'approccio incrementale adottato può essere rafforzato introducendo pratiche di *progressive delivery*, in cui l'aumento di traffico verso il to-be avviene per step controllati ed è governato da policy misurabili.

In tale impostazione, SLI (*Service Level Indicator*) indica un indicatore osservabile che misura il comportamento del servizio, mentre SLO (*Service Level Objective*) definisce l'obiettivo/limite accettabile per tale indicatore, espresso come soglia quantitativa. Gli SLO possono quindi essere usati come soglie operative per automatizzare le decisioni di rollout: se gli SLI restano entro gli obiettivi si procede con l'aumento graduale dell'esposizione, altrimenti si interrompe la progressione o si attiva un rollback [70].

A supporto, l'uso sistematico di *feature flag* consente di separare il deploy del codice dall'attivazione delle funzionalità: il codice può essere distribuito in produzione spento e abilitato/disabilitato in modo controllato senza ridistribuire, riducendo il rischio e migliorando la reversibilità operativa [70].

Questo tipo di evoluzione aumenta la sicurezza del cutover e rende più sostenibile la gestione prolungata della coesistenza, che richiede coordinamento e monitoraggio continuo.

6.2.3 Riallineamento dati

Poiché una parte dei mismatch può dipendere da latenza di propagazione, differenze di perimetro informativo e incompleta storicizzazione, una direttrice prioritaria è rendere esplicito il contratto tra DB principale e DB replica: quali entità e attributi sono replicati, quali campi sono "source of truth", quale finestra storica è garantita, e come vengono gestite correzioni tardive o ricalcoli.

In particolare, laddove la replica sia utilizzata per consultazioni retrospettive, occorre prevedere un meccanismo di storicizzazione (anche selettivo) o un processo di backfill controllato, così che le query storiche non dipendano da dataset incompleti.

Questa misura riduce squadrature "strutturali" e rende più interpretabile la distinzione tra divergenze attese (dataset/timing) e divergenze anomale (regressioni) [71].

Sul piano tecnico-operativo, è utile introdurre pipeline di riconciliazione e riallineamento: controlli periodici di consistenza tra viste (legacy vs to-be, DB principale vs DB replica) e procedure di ripristino (ad esempio replay controllato di eventi o processi di riallineamento batch).

L'obiettivo è trasformare la "squadratura" da anomalia occasionale analizzata caso per caso a un fenomeno misurabile. In questo contesto, anche la comparazione automatica può essere evoluta classificando sistematicamente le divergenze (attese

per differenze di dataset, attese per timing, anomalie) e tracciandone l'evoluzione nel tempo come indicatore di maturità del percorso di migrazione.

6.2.4 Strumenti AI a supporto

In prospettiva, l'impiego di tecniche basate su intelligenza artificiale può contribuire a gestire l'elevato volume di informazioni prodotto dai sistemi moderni, supportando attività di osservabilità e gestione operativa quali l'identificazione di anomalie su log e metriche, la classificazione dei mismatch tra as-is e to-be e l'assistenza alle fasi di raggruppamento e analisi incident, attraverso l'individuazione di pattern ricorrenti.

In particolare, durante l'esecuzione parallela la molteplicità di segnali disponibili rende utile affiancare alle soglie statiche approcci di *anomaly detection*, in grado di evidenziare regressioni prestazionali e degradazioni progressive non immediatamente riconoscibili con regole statistiche [54].

Tuttavia, in un dominio core banking l'adozione di tali strumenti deve essere governata tramite requisiti espliciti di protezione dei dati, auditabilità, controllo degli accessi e validazione del comportamento, così da ridurre i rischi legati all'utilizzo di dati sensibili e all'introduzione di componenti non pienamente deterministiche nei processi operativi. Di conseguenza, una linea di evoluzione realistica consiste nell'impiegare l'AI come supporto all'osservabilità e all'efficienza operativa, ma non come componente decisionale sul piano transazionale.

6.2.5 Estensione della valutazione

Poiché la tesi adotta un impianto GQM per collegare obiettivi a domande e metriche osservabili, un miglioramento metodologico consiste nell'estendere tale struttura includendo obiettivi specifici per le aree più critiche emerse durante la migrazione e valutandoli su un insieme di dati più ampio, considerando una finestra temporale più estesa e una maggiore variabilità di carico.

A titolo esemplificativo, possono essere introdotti obiettivi relativi a: riduzione dello scostamento progressivo tra viste dati, sostenibilità operativa della fase di coesistenza (ad esempio tasso di falsi positivi nelle squadrature), robustezza dei flussi asincroni (retry, gestione delle eccezioni) e integrazione tra processi batch e aggiornamenti real-time (latenza e accuratezza di convergenza delle viste).

Questa estensione consente di valutare in modo più completo l'evoluzione verso condizioni a regime post migrazione, mantenendo il rigore empirico già impostato e trasformando le limitazioni osservate in target misurabili per iterazioni successive.

Capitolo 7

Conclusioni

La modernizzazione di un sistema core banking non può essere interpretata come un esercizio tecnologico fine a sé stesso, né come una semplice sostituzione di piattaforme e linguaggi. Essa rappresenta, piuttosto, la risposta necessaria a un divario crescente tra le capacità effettive di un'organizzazione, costruite su sistemi affidabili ma rigidi, e le aspettative del mercato digitale, caratterizzato da rapidità di evoluzione, integrazione continua con ecosistemi esterni e domanda di servizi fruibili in tempo reale. In tale contesto, la modernizzazione diventa una scelta strategica che mira a preservare il valore accumulato nei decenni trasferendolo su un paradigma architetturale in grado di sostenere il cambiamento nel lungo periodo.

Il punto di partenza di questo lavoro è stato un sistema legacy funzionante, altamente affidabile e consolidato, ma intrinsecamente vincolato da un modello architetturale e operativo che limita la capacità di evoluzione. Il punto di arrivo è un'architettura che mira a mantenere gli stessi requisiti non funzionali di affidabilità e continuità, riconquistandoli su fondamenta diverse: modularità, disaccoppiamento, scalabilità orizzontale, e un modello di integrazione capace di ridurre dipendenze temporali e fragilità legate a catene sincrone.

Uno dei risultati più rilevanti che emerge dall'esperienza condotta è che la difficoltà principale non consiste nel progettare un'architettura to-be coerente, ma nel governare in modo controllato la coesistenza di due sistemi funzionalmente equivalenti e strutturalmente opposti, in produzione su traffico reale, senza degradare l'esperienza del cliente finale e senza introdurre rischi inaccettabili. La strategia di coesistenza adottata, basata su quadratura sistematica degli output e meccanismi di rollback, definisce un framework operativo replicabile per modernizzazioni di sistemi mission-critical.

Inoltre, la valutazione empirica impostata tramite GQM ha permesso di legare le scelte architetturali a evidenze misurabili: l'analisi è stata ancorata a dati raccolti in condizioni operative reali durante l'esecuzione parallela, consentendo un confronto diretto tra configurazione as-is e configurazione to-be. Questo approccio permette

di rilevare precocemente fenomeni che difficilmente emergono in test tradizionali: colli di bottiglia sotto carico reale, anomalie di integrazione, divergenze dovute a differenze nei dataset. I risultati ottenuti confermano che il modello asincrono può produrre benefici prestazionali concreti già in una fase di coesistenza parziale, prima del cutover definitivo.

Adottare microservizi significa accettare una maggiore complessità distribuita e trasferire parte delle garanzie che nel mondo legacy erano incorporate nel middleware transazionale verso discipline architetturali quali la gestione dell'idempotenza, della coerenza eventuale e della resilienza. Ne consegue che i vantaggi tecnici analizzati si realizzano pienamente solo se accompagnati da un'evoluzione organizzativa, senza la quale i microservizi rischiano di diventare monoliti distribuiti, con tutti i costi aggiunti e nessuno dei benefici attesi.

Nonostante i limiti evidenziati nel corso della trattazione, legati principalmente all'ampiezza campionaria dei servizi analizzati e alla complessità di gestione dei riallineamenti dati, l'approccio validato costituisce una base solida e generalizzabile per le iterazioni successive. I risultati risultano robusti nel perimetro analizzato e, proprio perché derivati da osservazioni in produzione, offrono indicazioni affidabili sul comportamento del sistema nelle condizioni effettive considerate. Tuttavia, l'estensione di tali evidenze a scenari più ampi richiede ulteriori campagne di valutazione, includendo una raccolta dati su periodi più lunghi e una maggiore variabilità di carico, così da aumentare la rappresentatività del campione e osservare la stabilità delle metriche nel tempo. Ciò non indebolisce le conclusioni, ma ne delimita l'ambito di validità: in un contesto empirico su sistemi complessi, la solidità delle evidenze si consolida progressivamente attraverso misurazioni ripetute, condizioni operative diversificate e strumenti di comparazione via via più maturi.

L'architettura to-be descritta in questo elaborato non è semplicemente un sistema più moderno e scalabile del precedente: è, a tutti gli effetti, una piattaforma concepita per accogliere il cambiamento. Mentre il monolite legacy limitava l'integrazione di nuovi modelli di business a causa della sua intrinseca rigidità, la nuova infrastruttura distribuita pone le basi per affrontare proattivamente le sfide dell'ecosistema bancario contemporaneo.

I risultati suggeriscono inoltre che l'evoluzione verso architetture event-driven asincrone è tecnicamente fattibile e produce benefici misurabili in termini di riduzione della latenza e dei tempi di elaborazione.

In conclusione, l'approccio proposto in questa tesi fornisce un modello metodologico e ingegneristico che mostra come sia possibile condurre transizioni architetturali profonde mantenendo la continuità operativa e riducendo il rischio di regressioni funzionali, a condizione di adottare strategie incrementali e di investire in osservabilità e validazione sistematica.

Bibliografia

[1] H. Kaplan. «A Migration Management Framework Proposal for COBOL/-CICS Based Mainframes». Tesi di laurea mag. Ankara, Turkey: Middle East Technical University, 2010. URL: https://open.metu.edu.tr/handle/11511/19139 (cit. alle pp. 1, 4).

[2] J. Fritzsch, J. Bogner, S. Wagner e A. Zimmermann. «Microservices Migration in Industry: Intentions, Strategies, and Challenges». In: *Proc. IEEE Int. Conf. Software Maintenance and Evolution (ICSME)*. 2019, pp. 481–490. DOI: 10.1109/ICSME.2019.00081 (cit. alle pp. 1, 39, 41, 46, 47, 49, 50).

[3] H. M. Sneed e K. Erdoes. «Migrating AS400-COBOL to Java: A Report from the Field». In: *Proc. 17th European Conf. Software Maintenance and Reengineering (CSMR)*. 2013, pp. 143–152. DOI: 10.1109/CSMR.2013.32 (cit. a p. 5).

[4] R. L. Glass. «COBOL — A Contradiction and an Enigma». In: *Communications of the ACM* 40.9 (1997), pp. 11–13. DOI: 10.1145/260750.260752 (cit. a p. 5).

[5] N. Ensmenger. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. Cambridge, MA: MIT Press, 2010 (cit. a p. 5).

[6] A. Upadhaya. «Understanding Legacy Software: The Current Relevance of COBOL». Tesi di laurea mag. Amsterdam, The Netherlands: VU University Amsterdam, 2023. URL: https://ictinstitute.nl/wp-content/uploads/2023/12/COBOL_Thesis_Dec4_Ashish.pdf (cit. a p. 5).

[7] C. M. Yalamanchili. «IBM Mainframe & Z/OS: Advanced Insights from a Programmer's Perspective». In: *International Journal for Multidisciplinary Research* 2.3 (2020). DOI: 10.36948/ijfmr.2020.v02i03.22604 (cit. a p. 5).

[8] A. S. Tanenbaum e M. Van Steen. *Distributed Systems: Principles and Paradigms*. Second. Upper Saddle River, NJ: Pearson Education, 2007 (cit. alle pp. 5, 6, 11, 34, 63).

[9] C. Song, M. Xu, K. Ye, H. Wu, S. S. Gill, R. Buyya e C. Xu. «ChainsFormer: A Chain Latency-Aware Resource Provisioning Approach for Microservices Cluster». In: *Proc. Int. Conf. Service-Oriented Computing (ICSOC)*. Springer, 2023, pp. 197–211. DOI: 10.48550/arXiv.2309.12592 (cit. alle pp. 6, 11, 24, 64).

[10] V. Kanvar, S. Tamilselvam e K. N. Raghunath. *Enabling Communication via APIs for Mainframe Applications*. 2024. URL: https://arxiv.org/abs/2408.04230 (cit. alle pp. 6–8, 10–12, 24, 34).

[11] IBM. *CICS and z/OS Connect*. 2025. URL: https://www.ibm.com/docs/en/cics-ts/5.5.0?topic=services-cics-zos-connect (cit. alle pp. 7, 12).

[12] C. M. Yalamanchili. «Managing Memory in CICS: Techniques and Best Practices». In: *International Journal of Multidisciplinary Research and Growth Evaluation* 3.3 (2022), pp. 658–662. DOI: 10.54660/.IJMRGE.2022.3.3.658-662 (cit. alle pp. 7, 11).

[13] IBM. *IBM z/OS Connect*. 2024. URL: https://www.ibm.com/products/zos-connect (cit. a p. 7).

[14] K. P. Eswaran, J. N. Gray, R. A. Lorie e I. L. Traiger. «The Notions of Consistency and Predicate Locks in a Database System». In: *Communications of the ACM* 19.11 (1976), pp. 624–633. DOI: 10.1145/360363.360369 (cit. alle pp. 7–9, 11).

[15] P. A. Bernstein. *Eight Transaction Papers by Jim Gray*. 2023. DOI: 10.48550/arXiv.2310.04601. URL: https://arxiv.org/abs/2310.04601 (cit. alle pp. 7, 9).

[16] J. Gray e A. Reuter. *Transaction Processing: Concepts and Techniques*. San Francisco, CA: Morgan Kaufmann, 1993 (cit. alle pp. 8, 9, 35).

[17] H. Lu, C. Li, J. Chen, Y. Chen e J. Zhou. *Towards Transaction as a Service*. 2023. URL: https://arxiv.org/abs/2311.07874 (cit. a p. 8).

[18] J. Gray. «The Transaction Concept: Virtues and Limitations». In: *Proc. 7th Int. Conf. Very Large Data Bases (VLDB)*. 1981, pp. 144–154. URL: https://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf (cit. alle pp. 8, 9).

[19] J. Gray. «A Transaction Model». In: *Automata, Languages and Programming*. Berlin, Germany: Springer, 1980, pp. 282–298. DOI: 10.1007/3-540-10003-2_78 (cit. a p. 9).

[20] A. Gill. «Building Scalable Batch Processing Systems for Financial Transactions Using Mainframes». In: *Turkish Journal of Computer and Mathematics Education* 10.12 (2019), pp. 92–109. DOI: 10.61841/turcomat.v10i12.14954 (cit. a p. 10).

[21] S. K. Das. «Modernizing QA Frameworks for Omni-Based Retirement Systems: A Shift from Manual Mainframe Testing to Intelligent Automation». In: *International Interdisciplinary Business Economics Advancement Journal* 6.5 (2025), pp. 68–89. DOI: 10.55640/business/volume06issue05-04 (cit. a p. 10).

[22] A. Ciborowska, A. Chakarov e R. Pandita. *Contemporary COBOL: Developers' Perspectives on Defects and Defect Location*. 2021. URL: https://arxiv.org/abs/2105.01830 (cit. a p. 11).

[23] K. Gite, A. Gunjal e V. Shinde. «Monolithic vs. Microservices Architecture: A Comparative Study of Software Development Paradigms». In: *International Scientific Journal of Engineering and Management* 4.6 (2025). DOI: 10.55041/ISJEM04468 (cit. alle pp. 12, 14, 24).

[24] S. K. Munugoti. «A Framework for Modernizing Legacy Core Banking Systems into Cloud-Native Microservices». In: *Journal of Information Systems Engineering and Management* 10.63s (2025). DOI: 10.52783/jisem.v10i63s.13858 (cit. a p. 12).

[25] N. Narkhede, G. Shapira e T. Palino. *Kafka: The Definitive Guide. Real-Time Data and Stream Processing at Scale*. Sebastopol, CA: O'Reilly Media, 2017 (cit. alle pp. 14, 25–28, 30, 34).

[26] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA: O'Reilly Media, 2015 (cit. a p. 14).

[27] C. M. Yalamanchili. «Java Interoperability with COBOL and Assembler in IBM CICS: Bridging Legacy and Modernization». In: *International Journal for Multidisciplinary Research* 6.2 (2024). URL: https://www.ijfmr.com/research-paper.php?id=45864 (cit. a p. 14).

[28] S. Brown. *The C4 Model for Software Architecture*. 2020. URL: https://c4model.com (cit. a p. 15).

[29] S. Habibullah, X. Liu, Z. Tan, Y. Zhang e Q. Liu. «Reviving Legacy Enterprise Systems with Microservice-Based Architecture Within Cloud Environments». In: *Proc. 7th Int. Conf. Software and Computer Applications (ICSCA)*. 2019, pp. 59–65. URL: http://aircconline.com/csit/papers/vol9/csit90713.pdf (cit. a p. 24).

[30] M. Fowler e J. Lewis. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html (cit. alle pp. 24, 40).

[31] P. Desai. «Ensuring Exactly-Once Semantics in Kafka Streaming Systems». In: *Journal of Computer Science and Technology Studies* (2025). URL: https://al-kindipublisher.com/index.php/jcsts/article/view/10849/9630 (cit. alle pp. 27, 28, 35).

[32] G. Hohpe e B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA: Addison-Wesley, 2004 (cit. a p. 28).

[33] C. M. Yalamanchili. «CICS Communication Frameworks: Deep Dive into Intercommunications and External Interfaces». In: *International Journal for Multidisciplinary Research* 3.4 (2021). DOI: 10.36948/ijfmr.2021.v03i04.41032 (cit. a p. 28).

[34] M. Mohammad. *Resilient Microservices: A Systematic Review of Recovery Patterns, Strategies, and Evaluation Frameworks*. 2025. URL: https://arxiv.org/abs/2512.16959 (cit. alle pp. 28, 35, 39–41, 47, 48, 50, 51, 63, 65).

[35] D. Kazemarks e J. Decouchant. *SoK: Microservice Architectures from a Dependability Perspective*. 2025. URL: https://arxiv.org/abs/2503.03392 (cit. alle pp. 28, 29).

[36] K. S. Hebbar. «Optimizing Distributed Transactions in Banking APIs: Saga Pattern vs. Two-Phase Commit (2PC)». In: *The American Journal of Engineering and Technology* 6 (2025), pp. 1–25. URL: https://theamericanjournals.com/index.php/tajet/article/view/6297 (cit. alle pp. 29, 35).

[37] S. Aydin e C. B. Cebi. «Comparison of Choreography vs Orchestration Based Saga Patterns in Microservices». In: *Proc. 3rd Int. Informatics and Software Engineering Conf. (IISEC)*. 2022, pp. 1–6. URL: https://ieeexplore.ieee.org/document/9872665 (cit. alle pp. 29, 30, 35).

[38] S. Singh, V. Shrivastava e A. Pandey. «Spring Framework / Spring Boot». In: *International Journal of Scientific Research in Engineering and Management* (2025). URL: https://ijsrem.com/download/spring-framework-spring-boot/ (cit. alle pp. 31–33).

[39] S. Pasuparthi. «Building Scalable Microservices with Spring Boot: A Technical Deep Dive». In: *Global Journal of Engineering and Technology Advances* 23.01 (2025), pp. 226–231. URL: https://gjeta.com/sites/default/files/GJETA-2025-0113.pdf (cit. alle pp. 31–35).

[40] M. K. Pillutla. «Enterprise-Scale Microservices Architecture: Domain-Driven Design and Cloud-Native Patterns Using the Spring Ecosystem». In: *Enterprise Architecture Journal* (2025), pp. 1–35. URL: https://eajournals.org/ejcsit/wp-content/uploads/sites/21/2025/06/Enterprise-Scale-Microservices.pdf (cit. alle pp. 31–35).

[41] R. A. Deshpande. «Application of Spring Boot Microservice Architecture for Scaling Banking Applications». In: *The American Journal of Engineering and Technology* 6 (2025), pp. 1–40. URL: https://theamericanjournals.com/index.php/tajet/article/view/6673/6108 (cit. alle pp. 31, 32).

[42] D. B. Jayaprakash. «Best Practices for Legacy System Modernization in Enterprise IT». In: *European Modern Studies Journal* 1.5 (2025). URL: https://lorojournals.com/index.php/emsj/article/view/1529/1482 (cit. a p. 37).

[43] C. A. De Souza e R. Zwicker. «Big-Bang, Small-Bangs ou Fases: Estudo dos Aspectos Relacionados ao Modo de Início de Operação de Sistemas ERP». In: *Revista de Administração Contemporânea* 7.4 (2003), pp. 59–82. URL: https://www.scielo.br/j/rac/a/b4Z6qsDJZzsKj3FFyYk5q3F/ (cit. alle pp. 38–41).

[44] S. R. Gudi. «Deconstructing Monoliths: A Fault-Aware Transition to Microservices with Gateway Optimization Using Spring Cloud». In: *IEEE Transactions on Services Computing* 18.3 (2025), pp. 1567–1582. URL: https://ieeexplore.ieee.org/document/11212326/ (cit. alle pp. 38, 41, 46).

[45] L. Kirkila, M. Klotina, J. Sproge, Z. Vilcina-Vectirane e A. Romanovs. «Case Study Review: IT Legacy System Migration Success Factors». In: *Proc. 66th Int. Scientific Conf. Information Technology and Management Science of Riga Technical University (ITMS)*. 2025. URL: https://ieeexplore.ieee.org/document/11236637/ (cit. alle pp. 38–41).

[46] J. Bogner, S. Wagner, J. Fritzsch, M. Haug e A. Zimmermann. «Towards an Architecture-Centric Methodology for Migrating to Microservices». In: *Software Architecture (ECSA)*. Springer, 2022, pp. 152–167. URL: https://arxiv.org/abs/2207.00507 (cit. alle pp. 38–41, 69).

[47] H. M. Ayas, P. Leitner e R. Hebig. «Facing the Giant: A Grounded Theory Study of Decision-Making in Microservices Migrations». In: *Proc. 15th Int. Conf. Mining Software Repositories (MSR)*. 2022. URL: https://arxiv.org/abs/2104.00390 (cit. a p. 38).

[48] T. O. Sanyaolu, A. G. Adeleke, C. P. Efunniyi, L. A. Akwawa e C. F. Azubuko. «Data Migration Strategies in Mergers and Acquisitions: A Case Study of Banking Sector». In: *Computer Science & IT Research Journal* 4.3 (2023), pp. 546–561. URL: https://fepbl.com/index.php/csitrj/article/view/1503 (cit. alle pp. 39–41).

[49] A. C. Suroju. «Legacy Application Modernization: A Strategic Framework for Enterprise Transformation». In: *World Journal of Advanced Research and Reviews* 26.2 (2025), pp. 4202–4207. DOI: 10.30574/wjarr.2025.26.2.2042 (cit. a p. 40).

[50] J. J. Carroll, P. Anand e D. Guo. *Preproduction Deploys: Cloud-Native Integration Testing*. 2021. URL: https://arxiv.org/abs/2110.08588 (cit. alle pp. 40, 41, 46).

[51] N. M. Pageler, M. J. Grazier G'Sell, W. Chandler, E. Mailes, C. Yang e C. A. Longhurst. «A Rational Approach to Legacy Data Validation When Transitioning Between Electronic Health Record Systems». In: *Journal of the American Medical Informatics Association* 23.5 (2016), pp. 991–994. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC11741007/ (cit. alle pp. 40, 41, 45).

[52] N. Shenisetty. «Architecting Cloud-Native Financial Systems: Key Principles and Patterns». In: *World Journal of Advanced Research and Reviews* 26.1 (2025), pp. 3672–3677. URL: https://journalwjarr.com/sites/default/files/fulltext_pdf/WJARR-2025-1518.pdf (cit. alle pp. 45, 49, 50).

[53] V. Jayakumar. «Enterprise System Integration Patterns: Lessons from Financial Services Transformation Projects». In: *Enterprise Journal of Computer Systems and Information Technology* (2025). URL: https://eajournals.org/ejcsit/wp-content/uploads/sites/21/2025/06/Enterprise-System-Integration-1.pdf (cit. alle pp. 46, 47, 49).

[54] L. Akmeemana, H. Faiz, C. Attanayake e S. Wickramanayake. *GAL-MAD: Towards Explainable Anomaly Detection in Microservice Applications Using Graph Attention Networks*. 2025. URL: https://arxiv.org/abs/2504.00058 (cit. alle pp. 46, 71).

[55] O. T. Odofin, A. A. Abayomi, S. Owoade, O. A. Agboola, B. I. Adekunle e A. C. Uzoka. «Designing Event-Driven Architecture for Financial Systems Using Kafka, Camunda BPM, and Process Engines». In: *International Journal of Scientific Research in Science, Engineering and Technology* (2024). URL: https://ijsrset.com/index.php/home/article/view/IJSRSET25121178/IJSRSET25121178 (cit. alle pp. 46, 47).

[56] F. Montesi e J. Weber. *Circuit Breakers, Discovery, and API Gateways in Microservices*. 2016. URL: https://arxiv.org/abs/1609.05830 (cit. a p. 46).

[57] S. K. Biradhara Nanagowda. «Microservices Architecture for High-Volume Finance Compliance Applications». In: *American Journal of Finance and Business Management* (2025). URL: https://gprjournals.org/journals/index.php/ajfbm/article/view/436/440 (cit. a p. 47).

[58] S. Chandnani e A. Nagrale. «Idempotency in Payment Systems: A Critical Analysis». In: *World Journal of Advanced Research and Reviews* 26.1 (2025), pp. 3996–4002. URL: https://journalwjarr.com/sites/default/files/fulltext_pdf/WJARR-2025-1569.pdf (cit. alle pp. 48, 49).

[59] S. S. V. Chekuri. «A Review of Idempotent and Concurrency-Safe Data Processing Patterns for Large-Scale Financial Systems». In: *Universal Library of Innovative Research and Studies* 2.4 (2025). URL: https://ulopenaccess.com/papers/ULIRS_V02I04/ULIRS20250204_018.pdf (cit. alle pp. 48, 49).

[60] M. Fotache e D. Cogean. «NoSQL and SQL Databases for Mobile Applications: Case Study: MongoDB Versus PostgreSQL». In: *Informatica Economică* 17.2 (2013), pp. 41–58. DOI: 10.12948/issn14531305/17.2.2013.04 (cit. a p. 49).

[61] R. T. Mason. «NoSQL Databases and Data Modeling Techniques for a Document-Oriented Database». In: *Proc. Informing Science and Information Technology Education Conf. (InSITE)*. 2015, pp. 259–268. URL: https://proceedings.informingscience.org/InSITE2015/InSITE15p259-268Mason1569.pdf (cit. a p. 50).

[62] N. Dragoni, S. Dustdar, S. T. Larsen e M. Mazzara. *Microservices: Migration of a Mission Critical System*. 2017. URL: https://arxiv.org/abs/1704.04173 (cit. a p. 50).

[63] M. Singh. «Resilient Microservices Architecture with Embedded AI Observability for Financial Systems». In: *Journal of Engineering Studies* 16.4 (2024). URL: https://journal.esrgroups.org/jes/article/view/8596/5766 (cit. a p. 50).

[64] V. R. Basili, G. Caldiera e H. D. Rombach. «The Goal Question Metric Approach». In: *Encyclopedia of Software Engineering*. Wiley, 1994. URL: https://www.cs.umd.edu/~mvz/handouts/gqm.pdf (cit. a p. 53).

[65] R. von Behren, J. Condit, F. Zhou, G. C. Necula e E. Brewer. «Capriccio: Scalable Threads for Internet Services». In: *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. 2003. DOI: 10.1145/945445.945471 (cit. a p. 61).

[66] Y. Fu e C. Soman. «Real-Time Data Infrastructure at Uber». In: *Proc. 2021 Int. Conf. Management of Data (SIGMOD '21)*. 2021. URL: https://arxiv.org/abs/2104.00087 (cit. a p. 62).

[67] R. T. Yarlagadda. «Performance Optimization of Spring Boot Microservices Using Kafka-Based Asynchronous Communication». In: *International Journal of Trend in Research and Development* 11.6 (2024). URL: https://www.ijtrd.com/ViewFullText.aspx?Id=29119 (cit. a p. 65).

[68] M. Waseem, P. Liang, M. A. Babar e J. Schneider. *An Empirical Study on Challenges of Event Management in Microservice Architectures*. 2024. URL: https://arxiv.org/abs/2408.00440 (cit. a p. 67).

[69] Y. Jia, H. Hu, S. He, F. Xu, F. Zhang, S. Wang e M. Zhang. *Microservice API Evolution in Practice: A Study on Strategies and Challenges*. 2023. URL: https://arxiv.org/abs/2311.08175 (cit. a p. 68).

[70] D. Taibi, V. Lenarduzzi e C. Pahl. *Continuous Architecting with Microservices and DevOps: A Systematic Mapping Study*. 2020. URL: https://arxiv.org/abs/1908.10337 (cit. a p. 70).

[71] D. Taibi, V. Lenarduzzi, M. Felderer e F. Auer. «From Monolithic Systems to Microservices: An Assessment Framework». In: *Information and Software Technology* 137 (2021). DOI: 10.1016/j.infsof.2021.106600. URL: https://www.sciencedirect.com/science/article/pii/S0950584921000793 (cit. a p. 70).