Orchestrazione vs Coreografia nel Pattern Saga: Un Progetto E-commerce a Microservizi

Autore – Matteo La Gioia – Dipartimento di Ingegneria dell'Informazione, Università di Roma Tor Vergata

Abstract – Il *pattern* Saga è una strategia per gestire transazioni distribuite in architetture a microservizi, garantendo la consistenza dei dati senza ricorrere a transazioni ACID globali. In questo articolo presento un'analisi approfondita di un progetto dimostrativo (disponibile su GitHub) che implementa un'applicazione e-commerce basata su microservizi per confrontare due varianti del pattern Saga: **orchestrazione** e **coreografia**. Vengono descritte la motivazione e il contesto del progetto nell'ambito dei sistemi distribuiti, i fondamenti teorici del pattern Saga e le differenze tra orchestrazione e coreografia. Si illustra l'architettura realizzata – comprendente microservizi dedicati (ordine, inventario, pagamento, autenticazione), un *API Gateway*, e un broker di messaggi (*RabbitMQ*) – evidenziando i flussi operativi in entrambe le modalità Saga. Si approfondiscono i dettagli implementativi: stack tecnologico (back-end in Go, front-end in React), containerizzazione in Docker, parametri di configurazione esterni, gestione degli eventi e meccanismi di compensazione in caso di fallimenti. Vengono poi analizzate le funzionalità utente offerte (registrazione e login, creazione ordini con controllo scorte e pagamenti, scelta dinamica del tipo di Saga) e le strategie di testing adottate, sia manuali che automatiche, per validare la correttezza e la resilienza del sistema di fronte a guasti simulati. I risultati confermano che l'applicazione soddisfa i requisiti di consistenza e affidabilità: in ogni scenario di fallimento le transazioni di compensazione ripristinano lo stato globale consistente. Viene inoltre presentato un confronto critico tra l'approccio con orchestratore centralizzato e quello con coreografia basata su eventi, discutendone vantaggi, limiti e impatto su modularità e manutenibilità. Infine, si delineano le considerazioni conclusive, evidenziando le limitazioni riscontrate (es. assenza di persistenza duratura, utenze separate per flusso) e possibili sviluppi futuri, come l'estensione delle funzionalità e l'adozione di soluzioni enterprise per il coordinamento Saga.

I. Introduzione

Nei moderni sistemi *software* distribuiti basati su microservizi, garantire la consistenza dei dati attraverso operazioni che coinvolgono servizi multipli è una sfida fondamentale. Transazioni tradizionali con proprietà ACID non sono applicabili in contesti distribuiti con database separati[1]. In tale scenario si inserisce il *pattern* **Saga**, introdotto originariamente in ambito database, che suddivide una transazione globale in una sequenza di transazioni locali autonome coordinate logicamente. Ciascuna sotto-transazione locale esegue aggiornamenti su un singolo servizio e pubblica un evento/messaggio per innescare la successiva; qualora una di esse fallisca, viene eseguita una serie di azioni **compensative** per annullare le modifiche effettuate dai passi precedenti, riportando il sistema in uno stato consistente[2]. Questo approccio incrementa la resilienza e la scalabilità delle transazioni distribuite, evitando blocchi prolungati di risorse e colli di bottiglia[3].

Il progetto analizzato in questo lavoro realizza un'applicazione di e-commerce a microservizi con l'obiettivo di comparare le due principali varianti di Saga: **orchestrazione centralizzata** e **coreografia distribuita**. La motivazione risiede nel comprendere i pro e i contro di ciascun approccio in termini di semplicità di progettazione, consistenza dei dati, tolleranza ai guasti e manutenibilità, in un contesto realistico (gestione di ordini, inventario e pagamenti). In particolare, l'applicazione implementa entrambe le varianti e permette di scegliere dinamicamente quale utilizzare per ogni operazione, fornendo un banco di prova diretto per l'analisi comparativa.

Nel seguito del paper presento dapprima i concetti di base del pattern Saga e le differenze tra orchestrazione e coreografia (Sezione *Background*). Successivamente descrivo nel dettaglio l'architettura e il design della soluzione sviluppata (Sezione *Design della Soluzione*), inclusa la descrizione dei microservizi coinvolti, dei flussi operativi nei due modelli Saga e dei componenti infrastrutturali come RabbitMQ. La Sezione *Dettagli Implementativi* approfondisce le tecnologie e i linguaggi utilizzati (Go, React, Docker), la configurazione del sistema, la gestione degli eventi e delle transazioni compensative. Vengono poi illustrate le principali funzionalità lato utente offerte dall'applicazione (autenticazione, creazione ordini, ecc.) e le modalità di testing adottate (test manuali dei vari scenari di successo/fallimento e test automatici tramite script), insieme ai risultati osservati (Sezione *Risultati*). Nella Sezione *Discussione* si confrontano criticamente i due approcci Saga nell'applicazione, evidenziandone vantaggi, svantaggi e potenziali implicazioni pratiche. Infine, la Sezione *Conclusioni* riassume i punti chiave, discute le limitazioni note del progetto e propone possibili estensioni e sviluppi futuri.

II. BACKGROUND

Il pattern Saga. Una *Saga* in un'architettura a microservizi è definita come una sequenza predefinita di transazioni locali distribuite tra servizi, che insieme realizzano una logica transazionale globale[2]. A differenza delle transazioni tradizionali bloccanti, una Saga non fornisce rollback automatico dell'intera transazione in caso di errore; al contrario, è responsabilità dell'applicazione definire esplicitamente, per ogni step che può fallire, una transazione di **compensazione** che annulli gli effetti dei passi già completati. In questo modo si mantiene la **integrità dei dati** attraverso la *consistenza eventuale*: tutte le sotto-operazioni vengono eseguite con successo oppure, in caso di fallimento parziale, le operazioni completate in precedenza vengono compensate una ad una fino a riportare il sistema a uno stato consistente noto. Il principale beneficio del pattern Saga è la rimozione della necessità di lock distribuiti e coordinamento forte tra servizi durante l'esecuzione della transazione, permettendo a ciascun microservizio di operare in modo autonomo e asincrono. Ciò riduce i colli di bottiglia e migliora la resilienza, al prezzo di una maggiore complessità nella logica applicativa (che deve gestire le compensazioni e la visibilità temporanea di stati intermedi non confermati)[4][5].

Orchestrazione vs Coreografia. Esistono due strategie principali per implementare una Saga nei sistemi distribuiti[6][7]:

***Saga con Orchestrazione**:* è presente un componente **Orchestratore** centrale che coordina esplicitamente l'esecuzione delle transazioni locali presso i vari servizi. L'orchestratore conosce la sequenza delle operazioni da effettuare (ad es. "crea ordine", poi "riserva inventario", poi "processa pagamento") e invia comandi diretti ad ogni microservizio nell'ordine previsto, attendendo la risposta prima di passare allo step successivo. In caso di errore in uno step, l'orchestratore invoca le dovute azioni compensative (ad es. annullare ordini o rilasciare prenotazioni) sempre tramite chiamate dirette. Questo approccio centralizzato offre maggiore **visibilità e controllo** sul flusso della Saga – facilitando monitoraggio e *debug* – e permette di implementare logiche di timeout o retry in un unico punto. Di contro, l'orchestratore introduce un **single point of failure** (può diventare un punto critico se non reso ridondante) e concentra in sé la logica dell'intero processo, aumentando l'accoppiamento e la responsabilità di un singolo servizio[8][9].

***Saga con Coreografia**:* non vi è alcun coordinatore centrale; la Saga è gestita in maniera **decentralizzata** tramite un meccanismo di scambio di eventi. Ogni microservizio, al completamento di una sotto-transazione locale, pubblica un **evento di dominio** su un canale comune (tipicamente un broker di messaggi). Gli altri servizi sono in ascolto (sottoscritti) a specifici eventi di interesse e, quando ne ricevono uno, reagiscono eseguendo la propria logica locale e a loro volta emettendo un evento successivo. Ad esempio, un servizio Ordini può emettere un evento "OrderCreated" che viene recepito dal servizio Inventario, il quale riserva gli articoli e pubblica "InventoryReserved", captato a sua volta dal servizio Pagamenti, e così via. In caso di fallimento, un servizio emette eventi compensativi (es. "PaymentFailed") che innescano le operazioni di rollback negli altri servizi coinvolti[10]. La coreografia presenta il vantaggio di un **accoppiamento molto debole**: ogni servizio conosce solo gli eventi che deve emettere o ascoltare, senza dipendenze dirette; il sistema può essere esteso aggiungendo nuovi servizi/eventi senza modificare un coordinatore centrale. Inoltre, non avendo un controllore unico, si elimina il rischio di un singolo punto di guasto. Tuttavia, questa libertà comporta **complessità crescente** al crescere del numero di servizi ed eventi: diventa difficile ottenere una visione globale della Saga, il flusso di esecuzione è implicito nei pattern di messaggi e risulta meno **predicibile e tracciabile** (rendendo il *debug* più complesso)[11][12]. Inoltre, implementare strategie comuni di *retry*, timeout o garantire l'ordine assoluto degli eventi può essere complicato in assenza di un coordinamento centralizzato[13]. In sintesi, l'orchestrazione offre più controllo ma introduce un componente centralizzato, mentre la coreografia massimizza la decoupling e la resilienza ma con meno governabilità.

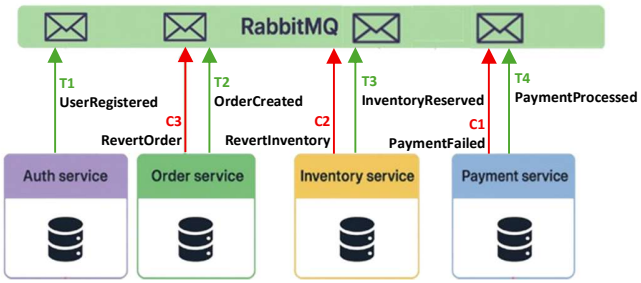

***Figura 1:** Schema semplificato della Saga con coreografia (eventi su RabbitMQ). I servizi Order, Inventory, Payment e Auth eseguono le transazioni locali T1–T4 pubblicando eventi (frecce verdi). In caso di fallimento del pagamento (C1: "Payment failed"), vengono emessi eventi compensativi per ripristinare inventario (C2) e ordine (C3).*

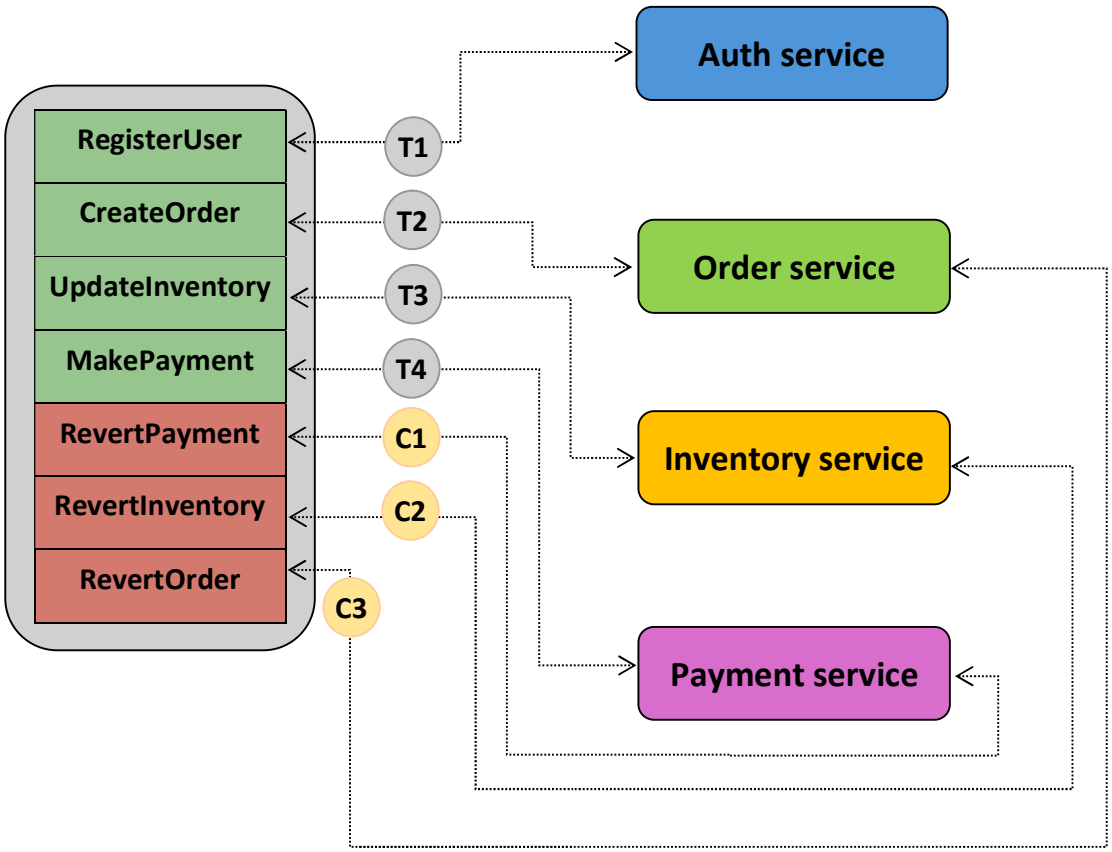

Figura 2: *Schema semplificato di una Saga con orchestrazione. Un servizio orchestratore invoca in sequenza le operazioni sui servizi Order, Inventory, Payment (T1–T4). Se un passo fallisce, l'orchestratore coordina l'esecuzione dei compensi C1–C3.*

III. DESIGN DELLA SOLUZIONE

A. Architettura generale.

L'applicazione sviluppata è un sistema di gestione ordini e-commerce composto da un insieme di microservizi, ciascuno responsabile di un sotto-dominio, che collaborano per elaborare la creazione di un ordine cliente dall'inizio alla fine[18]. Tutti i servizi sono containerizzati tramite Docker e orchestrati con Docker Compose, costituendo un ambiente distribuito eseguibile su singola macchina. La **figura 3** illustra i componenti principali e le interazioni nei due approcci Saga implementati.

Nel **flusso coreografico**, non esiste un coordinatore centrale: la comunicazione tra microservizi avviene in modo completamente asincrono mediante un broker di messaggi **RabbitMQ**. I servizi pubblicano eventi sul broker e si sottoscrivono agli eventi di interesse in un meccanismo di reazione a cascata. Il ciclo tipico è: il servizio Ordini pubblica un evento `OrderCreated`; il servizio Inventario (ascoltatore di tale evento) riserva la merce e pubblica un evento `InventoryReserved`; il servizio Pagamenti (ascoltatore) tenta il pagamento e pubblica un evento di esito, ad esempio `PaymentProcessed` se tutto ok oppure `PaymentFailed` in caso di errore. Gli altri servizi reagiscono di conseguenza: ad esempio, su `PaymentFailed`, il servizio Ordini aggiorna lo stato dell'ordine a "reject" e pubblica un evento di compensazione `RevertInventory` per segnalare al servizio Inventario di ripristinare le scorte precedentemente riservate[10][19]. L'intero flusso avviene tramite scambio di messaggi RabbitMQ, garantendo disaccoppiamento e asincronicità.

Nel **flusso orchestrato**, è presente un servizio dedicato denominato **Orchestrator** che gestisce l'intero processo in maniera sincrona e sequenziale[20]. In questo scenario, il frontend (tramite il suo gateway API) invia la richiesta di creazione ordine direttamente all'orchestratore, il quale esegue una serie di chiamate HTTP ai servizi interni in un ordine prestabilito: prima invoca il servizio Ordini per registrare l'ordine (stato iniziale "pending"), poi richiama il servizio Inventario per riservare i prodotti, infine chiama il servizio Pagamenti per addebitare l'importo[21]. Se tutte le chiamate hanno successo, l'orchestratore conferma l'ordine completando la Saga; se una di esse fallisce (es. pagamento rifiutato o errore di rete), l'orchestratore intraprende le

operazioni di **compensazione** necessarie inviando ulteriori chiamate ai servizi interessati. Ad esempio, in caso di fallimento del passo Pagamento, l'orchestratore chiama il servizio Inventario per annullare la prenotazione precedentemente effettuata e lo stesso servizio Ordini per marcare l'ordine come cancellato[17][22]. L'orchestratore mantiene un proprio log in-memory degli step completati e del loro stato, in modo da eseguire le compensazioni in ordine inverso rispetto alle operazioni effettuate. Tutta la logica di coordinamento, dunque, risiede centralmente in questo componente.

B. Microservizi e componenti.

Entrambe le modalità condividono molte componenti comuni, differenziandosi solo per la presenza o meno dell'orchestratore e per il meccanismo di comunicazione. I principali servizi implementati nel progetto sono[23]:

API Gateway – Un gateway di ingresso unico per il frontend. Riceve le richieste HTTP dall'applicazione React e le smista verso i servizi interni appropriati in base al tipo di flusso Saga selezionato (coreografico o orchestrato). Ad esempio, una richiesta di *creazione ordine* viene inoltrata al servizio orchestratore (per Saga orchestrata) oppure direttamente al servizio Ordini (per Saga coreografata). Il gateway incorpora anche una logica di autenticazione centralizzata: intercetta le richieste verificando un header utente e delegando la validazione al microservizio di autenticazione pertinente[24][25].

Service di Autenticazione (uno per il flusso coreografico e uno per l'orchestrato) – Gestisce la registrazione di nuovi utenti e il login. Ogni istanza mantiene un elenco di utenti (con username e *hash* della password) separato[26], simulando due database utenti distinti: ciò riflette l'idea che nei due domini (flusso orchestrato vs coreografico) i dati non siano condivisi. Questa scelta permette di dimostrare la funzionalità di *cross-flow user validation*: se un utente autenticato in un flusso passa all'altro flusso, il sistema verifica se quell'utente esiste anche nell'altro contesto e, in caso negativo, esegue automaticamente il logout per sicurezza[27]. Tale controllo è implementato dal gateway, che su cambio di modalità chiama le API `/validate` del servizio Auth opposto[25][28]. Dal punto di vista implementativo, il servizio Auth espone endpoint REST per `/register`, `/login` (che restituisce un `customer_id` usato come identificatore utente nelle altre richieste) e `/validate`[29]. I dati utente sono mantenuti in memoria (una lista) e inizializzati con alcuni account di esempio[26][30].

Servizio Ordini (uno per il flusso coreografico e uno per l'orchestrato) – Responsabile della creazione e gestione dello stato degli ordini clienti. In entrambi i flussi, all'arrivo di una richiesta di nuovo ordine il servizio Ordini inserisce un record di ordine con stato iniziale `pending` in un archivio locale (una struttura dati in memoria)[31][32]. Nel caso orchestrato, questo inserimento avviene su comando dell'orchestratore (API `/create_order` interna); nel caso coreografato, avviene su chiamata diretta dal gateway al servizio stesso. Successivamente, nel flusso coreografico il servizio pubblica un evento `OrderCreated` su RabbitMQ per notificare gli altri servizi[33], mentre nel flusso orchestrato risponde semplicemente all'orchestratore con un esito di successo. Il servizio Ordini gestisce anche l'aggiornamento finale dello stato dell'ordine: ad esempio, nel modello coreografico è in ascolto di eventi di esito come `PaymentProcessed` (che lo porta a marcare l'ordine come `approved`) o `InventoryReservationFailed`/`PaymentFailed` (che lo portano a marcare l'ordine come `rejected` indicando il motivo)[34][35]. Inoltre, in caso di `PaymentFailed` il servizio Ordini emette l'evento di compensazione `RevertInventory` per attivare il rollback sulle scorte[36]. Nel modello orchestrato invece l'aggiornamento dello stato a `approved`/`rejected` è comandato dall'orchestratore tramite un'apposita API (esposta dal servizio Ordini) `/update_status`[37]. Da notare che il servizio Ordini fornisce anche endpoint per ottenere la lista ordini di un utente o il dettaglio di un ordine, utilizzati dal frontend per aggiornare la UI[38][39].

Servizio Inventario (uno per il flusso coreografico e uno per l'orchestrato) – Gestisce il catalogo prodotti e le scorte disponibili. Mantiene in memoria una lista di prodotti con dettagli (nome, descrizione, prezzo, quantità disponibile, URL immagine) predefiniti[40][41]. Nel flusso orchestrato, il servizio espone API come `/get_price` (per ottenere il prezzo di un prodotto, usata dall'orchestratore in fase di calcolo totale ordine) e `/reserve` (per riservare le quantità di prodotti di un ordine, decrementandole dallo stock) e `/cancel_reservation` (per ripristinare le quantità in caso di annullamento)[42][43]. L'orchestratore invoca queste API sincronicamente durante la Saga. Nel flusso coreografico, invece, il servizio Inventario agisce reagendo a eventi: in ascolto di `OrderCreated`, esegue la logica di prenotazione delle scorte; se i prodotti sono disponibili in quantità sufficiente, riduce gli stock e pubblica un evento `InventoryReserved` (includendo nel payload l'importo totale calcolato e altri dettagli), altrimenti pubblica un evento `InventoryReservationFailed` con il motivo del fallimento (es. "stock insufficiente")[10]. Inoltre, ascolta eventi `RevertInventory` per effettuare rollback di prenotazioni (incrementando di nuovo le quantità) qualora un ordine venga cancellato successivamente[19]. In questo progetto le operazioni su inventario sono immediate e atomiche (dato l'uso di un semplice map in memoria protetto da lock), per cui si assume che *T2* e le eventuali *C2* (vedi Fig.1) eseguano in modo affidabile.

Servizio Pagamenti (due istanze analoghe) – Simula l'elaborazione di un pagamento. Nel flusso orchestrato, espone un'API `/process` che l'orchestratore chiama passando importo e cliente; il servizio effettua la transazione e risponde immediatamente con successo o fallimento. Nel flusso coreografico, il servizio Pagamenti è sottoscritto all'evento `InventoryReserved` (che indica che l'ordine ha riservato la merce ed è pronto per addebito): quando lo riceve, procede a *processare* il pagamento[44]. In entrambi i modelli, la logica prevede due possibili cause di fallimento simulate:

- **Soglia importo:** se l'importo totale dell'ordine supera una certa soglia (configurabile), il pagamento viene rifiutato automaticamente per ragioni di limite (es. carta di credito con plafond insufficiente). Questa soglia è impostata via variabile d'ambiente `PAYMENT_AMOUNT_LIMIT` (valore di default 2000.00) e rende riproducibile uno scenario di fallimento business[31][45].
- **Errore casuale:** il servizio Pagamenti utilizza un componente *payment gateway* interno che in alcuni casi può restituire un errore simulando guasti di rete o problemi esterni[46]. Questo introduce una percentuale di fallimenti imprevedibili.

Nel modello coreografico, se il pagamento fallisce il servizio Pagamenti pubblica un evento `PaymentFailed` (contenente l'ID ordine e la ragione dell'errore)[47]; se ha successo, pubblica `PaymentProcessed`. Inoltre, è in ascolto di eventuali eventi di compensazione per il pagamento: nell'implementazione attuale, il servizio Pagamenti si sottoscrive all'evento `RevertInventory` (che in questo caso indica che l'ordine è stato annullato dopo che il pagamento era già andato a buon fine) per eseguire una compensazione dell'addebito[48]. In pratica, su `RevertInventory` effettua una chiamata di `refund` (stornando il pagamento) e segna la transazione come `reverted` nel proprio archivio interno[49]. Nel modello orchestrato, la compensazione del pagamento viene invece richiesta dall'orchestratore tramite l'API `/revert` esposta dallo stesso servizio Pagamenti[50][51].

RabbitMQ – È il broker di messaggi usato unicamente nel flusso coreografico per distribuire gli eventi tra servizi. Viene eseguito come container separato e accessibile via URL amqp interno; le code/topic sono create per gli eventi del dominio Saga (`OrderCreated`, `InventoryReserved`, ecc.). I servizi coreografati si connettono al broker all'avvio e sottoscrivono i rispettivi topic di interesse. Nel flusso orchestrato RabbitMQ rimane inutilizzato (sebbene attivo nel `docker-compose`, i servizi orchestrati non lo usano). È comunque disponibile una console di management su porta 15672 per ispezionare code ed eventuali messaggi[52][53].

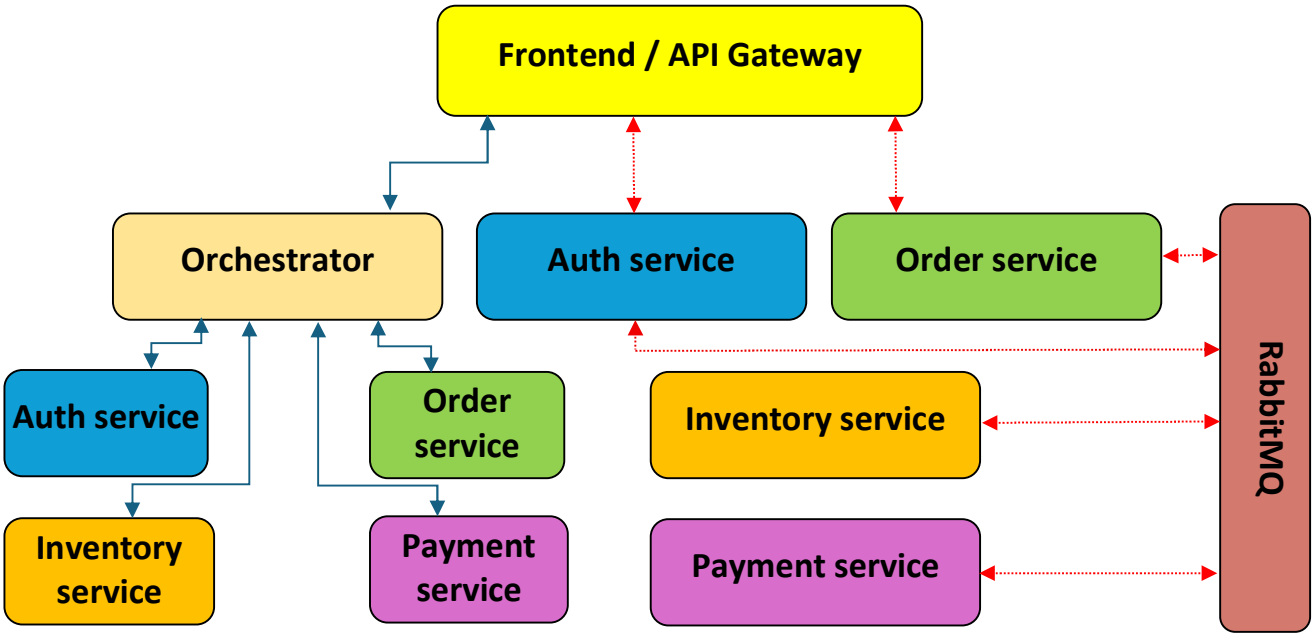

Figura 3: Architettura del sistema Saga Demo. *Entrambi i pattern condividono gli stessi microservizi di dominio (Auth, Order, Inventory, Payment) e l'api gateway/frontend. In modalità coreografica (linee tratteggiate rosse) i servizi comunicano attraverso eventi su RabbitMQ, senza coordinatore centrale. In modalità orchestrata (linee piene blu) le richieste passano attraverso l'Orchestrator, che invoca in sequenza le API REST interne dei servizi. Il frontend può selezionare la modalità Saga desiderata per ciascuna operazione.*

C. *Gestione dello stato e persistenza.*

Per semplicità, il progetto non introduce database esterni: ogni servizio mantiene i propri dati in memoria (strutture dati thread-safe). Esiste un modulo `common` di **data store** condiviso che definisce delle strutture globali in-memory: ad esempio una mappa degli ordini (`DB.Orders`) e dei prodotti (`DB.Products`), più una lista utenti (`DB.Users`), protette da mutex per accesso concorrente[54][55]. All'avvio di ciascun microservizio, viene chiamata una inizializzazione che popola queste strutture con alcuni dati di esempio: tre prodotti (Laptop, Mouse, Tastiera con relative quantità e immagini URL) e due utenti di test[40]. Questo approccio semplificato consente di concentrarsi sulla logica Saga senza dover configurare database; naturalmente, in un sistema reale i servizi avrebbero database separati per ordini, inventario, ecc., incarnando il pattern "database-per-service". La mancanza di persistenza su storage permanente implica che i dati si resetteranno a ogni riavvio dei servizi – una limitazione accettabile per uno scopo demo.

D. *Configurabilità e Parametri.*

Tutti i parametri di configurazione critici sono esterni al codice e gestiti tramite variabili d'ambiente definite in `docker-compose.yml`, come richiesto da specifiche. Ad esempio, le porte di ascolto dei servizi (`ORDER_SERVICE_PORT`, etc.), gli URL interni dei servizi (usati dal gateway e orchestrator), l'URL RabbitMQ (`RABBITMQ_URL`) e la soglia `PAYMENT_AMOUNT_LIMIT` sono tutti impostabili esternamente[53][56]. Docker Compose permette inoltre di far scalare orizzontalmente i servizi (ad esempio, avviando repliche multiple di un microservizio per gestire più load) e, tramite i meccanismi di healthcheck e restart, assicura una certa tolleranza ai guasti di container isolati. L'applicazione in sé è pensata per essere **fault-tolerant** a livello logico: grazie al pattern Saga, un fallimento in uno dei servizi durante una transazione viene compensato dalle altre componenti, mantenendo la consistenza dei dati globali[58].

IV. Dettagli Implementativi

E. *Linguaggio.*

Il progetto è realizzato con un mix di linguaggi: il **back-end** è interamente scritto in linguaggio Go. Il **front-end** è una Single Page Application in React (JavaScript/JSX) che offre un'interfaccia utente web interattiva. Il sistema di *build* e runtime impiega Docker: ogni microservizio Go ha il suo `Dockerfile` e viene eseguito come container separato, così come il front-end (che usa un'immagine `Node.js` per servire l'applicazione React) e il broker RabbitMQ. Il coordinamento dell'intera applicazione è affidato a Docker Compose, che con un singolo comando può costruire le immagini e lanciare l'insieme di container coinvolti, collegandoli in una rete virtuale comune. Ho quindi una **architettura distribuita**, seppur in miniatura: in totale, considerando i servizi duplicati per i due approcci e i componenti comuni, il sistema consiste di oltre 10 container isolati, comunicanti via HTTP REST (per orchestrazione) o via RabbitMQ (per eventi coreografici)[59].

F. *Interfaccia utente e flussi utente.*

L'applicazione React fornisce le seguenti funzionalità all'utente finale (cliente e-commerce) attraverso un'unica interfaccia web:

Registrazione e Login – L'utente può registrarsi fornendo username e password, oppure autenticarsi se possiede già un account. Poiché il sistema mantiene separati gli utenti per i due ambienti Saga, l'interfaccia distingue le registrazioni per flusso orchestrato e coreografato. Una volta *loggato*, l'app memorizza l'identificativo cliente (`customer_id`) restituito e lo include nelle richieste future (tramite header personalizzato `X-Customer-ID`)[60]. Il gateway utilizza questo ID per autenticare ogni chiamata (delegando al servizio `Auth` interno come descritto sopra). Importante: se l'utente passa da un flusso Saga all'altro attraverso l'apposito toggle UI, il sistema controlla se quell'utente esiste nell'altro contesto; in caso negativo informa l'utente che dovrà registrarsi anche in quell'ambiente, e lo disconnette.

Visualizzazione Catalogo Prodotti – Dopo il login, l'utente può vedere la lista dei prodotti disponibili a catalogo, comprensiva di nome, descrizione breve, prezzo e quantità attualmente disponibili in stock. Questi dati provengono dal servizio Inventario (endpoint `/catalog`), il quale li fornisce direttamente leggendo la sua mappa di prodotti. Il gateway instrada la richiesta di catalogo al servizio Inventory appropriato in base al flusso attivo (es. `http://choreographer-inventory-service:8082/catalog` se in modalità coreografica)[61]. Il front-end mostra anche un'immagine per ogni prodotto (le URL delle immagini sono codificate nei dati iniziali e puntano a risorse online statiche)[40]. La disponibilità è aggiornata in tempo reale.

Creazione di un Ordine – L'utente può selezionare uno o più prodotti dal catalogo, scegliere le quantità e sottomettere un ordine. L'applicazione consente di **scegliere dinamicamente il tipo di Saga** da usare per processare l'ordine. Questa scelta viene trasmessa al backend come parametro (come query string `?flow=orchestrated` oppure `?flow=choreographed` nelle richieste HTTP)[62][63]. Il gateway, leggendo questo parametro, deciderà come inoltrare la richiesta di creazione ordine:

- in modalità orchestrazione, effettua un POST verso l'endpoint dell'orchestratore (`/create_order` sull'URL dell'Orchestrator service)[64];

- in modalità coreografia, effettua un POST direttamente al servizio Ordini coreografico (`/create_order` su Order Service)[64].

In entrambi i casi, il corpo della richiesta include i dettagli dell'ordine (lista di item con ID prodotto e quantità). Nel caso orchestrato, l'orchestratore esegue la Saga sincronicamente e restituisce direttamente la risposta finale: se tutto ok, uno `status 200` con l'ordine completo (stato `approved`); se qualche passo fallisce, uno status di errore (`409 Conflict`) contenente l'ordine con stato `rejected` e motivazione[65][66]. Nel caso coreografico, invece, il servizio Ordini crea subito l'ordine in stato `pending` e avvia la Saga asincrona pubblicando l'evento, quindi risponde immediatamente con uno `status 202 Accepted`, indicando che l'ordine è "in elaborazione"[67]. In questo caso il front-end non conosce subito l'esito finale e deve ottenerlo successivamente.

Per fornire **feedback utente tempestivo anche nel flusso asincrono**, il front-end implementa un meccanismo di polling: dopo aver ricevuto l'ID ordine in risposta alla creazione (nel body della `202` viene ritornato l'`order_id` creato[68]), l'app periodicamente interroga il backend (api gateway) per chiedere lo stato aggiornato di quell'ordine (`GET /orders/{order_id}`) finché non vede uno stato finale (`approved`/`rejected`). Il gateway smista tali richieste al servizio Ordini appropriato. Grazie a questo polling, l'utente viene informato quasi in tempo reale sull'esito dell'ordine anche usando la coreografia. Nel caso orchestrato questo non è necessario poiché la risposta della chiamata iniziale è già definitiva.

Visualizzazione Storico Ordini: L'interfaccia offre la possibilità di vedere tutti gli ordini precedentemente effettuati dall'utente loggato, con relativi stati. Questa funzionalità interroga un endpoint GET `/orders?customer_id=<id>` sul gateway, il quale lo inoltra al servizio Ordini di competenza (orchestrato o coreografato)[69]. Il servizio Ordini filtra tutti gli ordini in memoria di quell'utente e li restituisce come lista JSON[70]. Ciò è utile per permettere all'utente di verificare, ad esempio, quali ordini sono stati approvati, quali eventualmente rifiutati (e perché).

G. Gestione degli errori e compensazioni.

Un aspetto critico implementato nel progetto è la corretta gestione dei casi di errore, per testare la robustezza del pattern Saga. Come accennato, i possibili punti di fallimento intenzionali sono:

- **Pagamento negato per importo eccessivo:** se l'ammontare supera la soglia (di default 2000). In tal caso, nel flusso orchestrato il servizio Pagamenti risponde all'orchestratore con un errore applicativo (codice `400` o `409`) includendo un messaggio "`amount exceeds limit`"; l'orchestratore intercetta l'errore e attiva la compensazione chiamando Inventario e Ordini per annullare[17][71]. Nel flusso coreografico, il servizio Pagamenti pubblica l'evento `PaymentFailed` con reason "`amount <X> exceeds limit <Y>`"[45]; il servizio Ordini reagisce marcando l'ordine `rejected` e innescando `RevertInventory`[72].
- **Errore casuale gateway pagamento:** simulato dalla funzione `ProcessPayment` che a volte restituisce errore (es. potrebbe lanciare eccezione per "`network timeout`"). La gestione è identica al caso precedente: orchestratore attiva compensazioni, coreografia propaga evento di fallimento. In aggiunta, se il pagamento era andato a buon fine ma successivamente un altro passo fallisce (es. un ipotetico step successivo come spedizione), il sistema prevede la compensazione del pagamento: orchestratore chiamando `/revert` su Payment Service, coreografia emettendo un evento apposito. Nel nostro scenario, *Payment* è ultimo step quindi questa situazione non si presenta (se pagamento è ok, la Saga termina con successo, non ci sono step dopo che possano fallire).
- **Fallimento inventario insufficiente:** Altro scenario gestito (più raro nei test data in quanto gli stock iniziali sono ampi) è il caso in cui l'inventario non possa riservare la quantità richiesta di un prodotto (es. ordini 5 pezzi ma ne è disponibili 2). Nel flusso orchestrato, l'API `/reserve` di Inventario restituirebbe un errore che fa interrompere l'orchestratore, il quale compensa eventuale ordine creato marcandolo come fallito[73]. Nel flusso coreografico, il servizio Inventario pubblicherebbe l'evento `InventoryReservationFailed`; il servizio Ordini recepisce tale evento e marca l'ordine come `rejected` con motivo appropriato[35], senza necessità di ulteriori compensazioni poiché nulla era ancora stato addebitato.

Le **transazioni compensative** sono esplicitamente implementate in entrambi i modelli. Nel caso orchestrato, il codice dell'orchestratore include una funzione `compensateSaga(orderID, order, reason)` che registra un evento di inizio compensazione nel log Saga e itera sugli step completati in ordine inverso, effettuando le seguenti azioni per ogni step da annullare[74][75]:

1. Se era già stato effettuato il pagamento (step T3 completato), chiama l'operazione di **revert payment** sul servizio Pagamenti (`POST /revert`) per stornare l'addebito[50].
2. Se l'inventario era stato riservato con successo (step T2 completato), invoca l'API di **cancel inventory reservation** (`POST /cancel_reservation`) sul

servizio Inventario, passando l'ID ordine e le quantità da ripristinare[76].

3. Infine, se l'ordine era stato creato (step T1 completato), ordina al servizio Ordini di aggiornare lo stato a "`rejected`" (tramite `POST /update_status`) per chiudere l'ordine come fallito[43][77].

Nel caso coreografico, le compensazioni avvengono implicitamente tramite i messaggi di evento: ad esempio, come descritto, `PaymentFailed` porta Order service a emettere `RevertInventory`[36], che a sua volta fa sì che il servizio Inventario ripristini le scorte. Inoltre il servizio Pagamenti, ascoltando `RevertInventory`, effettua un eventuale storno pagamento se necessario[48]. Si noti che in questa implementazione, dopo aver pubblicato `RevertInventory`, non si attende conferma dell'effettivo ripristino prima di completare la Saga – ciò riflette la natura asincrona del processo, dove si *presume* che le compensazioni vengano eseguite a breve. In un sistema reale si potrebbe prevedere un evento di conferma (es. `InventoryReverted`) se servisse garantire tracking completo, ma qui non è stato ritenuto necessario.

V. RISULTATI

L'applicazione è stata sottoposta a un insieme di test sia manuali sia automatici per verificare il corretto funzionamento di tutti i flussi (sia in condizioni ideali che in presenza di guasti simulati) e valutare gli obiettivi di correttezza e robustezza.

H. Testing manuale dei casi d'uso

Sono stati esplorati manualmente, tramite interazione con il front-end web, i seguenti scenari principali[78]:

Flusso di successo (no errori): creazione di un ordine valido (es. importo sotto la soglia) in entrambe le modalità Saga. *Atteso:* l'ordine viene approvato e compaiono ridotte le quantità in inventario.

Fallimento per limite importo: creazione di un ordine con totale deliberatamente sopra 2000 (es. aggiungendo molti pezzi), in entrambe le modalità. *Atteso:* l'ordine viene rifiutato; motivo segnalato "`amount exceeds limit…`"; nessuna modifica permanente a inventario (le scorte riservate vengono restituite).

Fallimento per errore casuale: forzato ripetendo più volte ordini di medio importo finché il servizio Pagamenti (che ha una componente random) simula un errore. *Atteso:* ordine rifiutato con motivo di errore tecnico; inventario ripristinato.

Validazione utente cross-flow: loggati come utente presente solo in un flusso (es. registrato in orchestrato ma non in coreografato), si prova a passare all'altro flusso. *Atteso:* il sistema rileva che l'utente non esiste nell'altro database e fa *logout* automatico, costringendo a registrarsi anche lì[79].

Tutti questi test manuali hanno dato esito positivo: in particolare, si è osservato che nelle condizioni di errore il sistema reagisce correttamente eseguendo le operazioni di compensazione previste e non lasciando **stati inconsistenti** (es. dopo un fallimento di pagamento, l'ordine risulta `rejected` e le quantità tornano disponibili come prima, confermando la corretta applicazione del pattern Saga).

I. Script di test automatici

Oltre ai test interattivi, il progetto fornisce uno script Bash (`scripts/test_saga.sh`) per automatizzare alcune verifiche sui flussi principali[80]. Questo script permette di lanciare dal terminale una sequenza di chiamate API per simulare diversi scenari.

```
./scripts/test_saga.sh orchestrated success
./scripts/test_saga.sh choreographed fail_limit
```

Il primo comando esegue un test di un ordine `orchestrated` che dovrebbe concludersi con successo, il secondo simula un ordine `choreographed` destinato a fallire per superamento soglia importo[81]. Lo script effettua richieste HTTP dirette agli endpoint appropriati (gateway o orchestratore) e verifica le risposte (codici HTTP attesi e campi nel JSON di risposta). I test automatici confermano che in ciascuno degli scenari di fallimento configurati, il sistema esegue la Saga di compensazione corretta, riportando lo stato all'origine (inventario ripristinato, ordine contrassegnato annullato) e garantendo consistenza dei dati[82]. Ad esempio, nel caso `fail_limit` coreografato, lo script verifica che dopo la creazione ordine (`202 Accepted`) e un breve wait, un `GET` sullo stato dell'ordine restituisca `status: rejected` con `reason` indicante il superamento della soglia, e che la quantità di prodotto sia rimasta invariata rispetto all'inizio. Analogamente, scenari di successo mostrano ordini in stato `approved`.

J. Raggiungimento dei requisiti e prestazioni

Dal punto di vista **funzionale**, il progetto soddisfa i requisiti prefissati di implementare entrambi i pattern Saga e di gestire tutte le principali casistiche (ordine valido, pagamento respinto, problemi di inventario, utente non valido) mantenendo la coerenza. Ogni operazione dell'utente riceve un esito corretto e motivato. **Correttezza**: in ogni test svolto, i dati finali riflettono esattamente il risultato atteso della Saga (nessun ordine rimasto in stato intermedio, nessuna quantità "persa" in caso di rollback, etc.), comprovando l'efficacia del coordinamento Saga.

Dal punto di vista **di resilienza**, il sistema dimostra tolleranza a guasti a livello applicativo; inoltre, grazie al disaccoppiamento, un temporaneo down di RabbitMQ blocca solo il flusso coreografico mentre quello orchestrato continua e viceversa. In caso di riavvio di un servizio, i dati persi vengono ricostituiti da zero: questo è accettabile per la demo ma evidenzia la necessità di database persistenti.

Quanto a **performance**, non sono stati condotti test di carico quantitativi, ma qualitativamente l'esperienza utente è risultata fluida. La scelta di implementare in linguaggio Go e di far girare tutto in locale contribuisce a buone prestazioni.

Infine, la **modularità** del sistema ha facilitato le prove: ad esempio, è stato semplice modificare configurazioni (come la soglia importo) via variabili senza toccare il codice, nonché scalare i servizi.

VI. Discussione

L'implementazione parallela dei due approcci Saga nello stesso dominio applicativo ci permette di confrontarne direttamente le caratteristiche sul campo. Di seguito discuto i punti salienti emersi dal confronto **orchestrazione vs coreografia** nell'ambito di questo progetto.

Chiarezza del flusso e Complessità di sviluppo: L'orchestrazione si è rivelata relativamente più **semplice da implementare** in codice in quanto la sequenza di operazioni è codificata in un unico posto (nel servizio Orchestrator). La logica Saga è esplicita e facile da seguire leggendo la funzione `startSaga(order)`[65][83]: questo ha agevolato anche il *debug* durante lo sviluppo, poiché era possibile tracciare il flusso con log centralizzati. Di contro, l'approccio coreografico ha richiesto di suddividere la logica globale in parti più piccole in ciascun microservizio e di affidarsi alla corretta emissione e ricezione di eventi: inizialmente, coordinare i naming degli eventi e assicurarsi che ogni servizio reagisse appropriatamente ha aggiunto complessità. Inoltre, per capire l'intero flusso è necessario considerare diversi punti del sistema, il che rende più ardua la comprensione rispetto al codice sequenziale dell'orchestratore. Questo riflette la differenza teorica di "visibilità" tra i due pattern: l'orchestrazione centralizza la conoscenza del workflow, la coreografia lo disperde tra i servizi.

Accoppiamento e indipendenza dei servizi: Dal punto di vista dell'**accoppiamento**, la coreografia ha mostrato i suoi vantaggi: i microservizi coreografici non conoscono l'esistenza l'uno dell'altro se non tramite gli eventi. Ad esempio, il servizio Pagamenti nella Saga coreografata ignora chi abbia originato l'evento `InventoryReserved` – potrebbe anche provenire da un sistema diverso (finché il formato coincide). Ciò conferisce una maggiore flessibilità architetturale: sarebbe possibile riutilizzare o estendere il sistema aggiungendo un nuovo servizio che si sottoscrive agli eventi esistenti, senza modificare quelli attuali. Nell'orchestrazione, invece, l'orchestratore contiene riferimenti diretti ai servizi esistenti e alle loro API (URL, formati payload): aggiungere un nuovo step (servizio) richiederebbe di modificarlo e ridistribuirlo. Si tratta quindi di un approccio più **rigido** e accoppiato. Tuttavia, vale la pena notare che nel progetto l'orchestratore è comunque configurato tramite variabili d'ambiente per gli URL dei servizi, e potrebbe quindi puntare a implementazioni diverse senza ricompilazione, ma sempre conoscendo a priori quali servizi chiamare[84]. In coreografia, i servizi aggiuntivi potrebbero integrarsi semplicemente ascoltando/emettendo eventi, il che è un forte punto a favore in scenari evolutivi.

Tolleranza ai guasti e single point of failure: La coreografia intrinsecamente elimina il single point of failure: non avendo un controllore centrale, il fallimento di un singolo servizio impatta solo le Saga che lo coinvolgono direttamente. Ad esempio, se l'orchestratore si guasta o diventa irraggiungibile, nel progetto **tutte** le operazioni orchestrated si bloccano, rendendo l'intero sistema inutilizzabile in quella modalità (il gateway non può procedere). Questo è un limite noto dell'orchestrazione – mitigabile con istanze ridondanti dell'orchestratore e meccanismi di fail-over, ma comunque un aspetto da gestire[85][8]. Nel modello coreografico, se un servizio (es. Payment) è giù, gli ordini rimangono in pending e RabbitMQ accumula messaggi o va in timeout: l'indisponibilità di Payment incide sugli ordini futuri ma non paralizza il resto. In sintesi, la coreografia offre un'architettura più **distribuita e resiliente** in termini di singoli punti di guasto. D'altro canto, è emerso un rischio inverso: la coreografia può soffrire di **cascata di errori** difficili da gestire globalmente – ad esempio se un messaggio critico viene perso o un servizio consumer non lo elabora correttamente, la Saga potrebbe restare incompleta senza un'entità centrale che se ne accorga. Nel nostro progetto RabbitMQ garantisce *at-least-once delivery* e i servizi sono stati implementati in modo idempotente, quindi i messaggi vengono ritentati se fallisce la consegna, ma in un contesto reale andrebbe curata molto l'osservabilità (log/tracing end-to-end) per individuare Saga bloccate[86]. Al contrario, con orchestrazione il tracking è più semplice: l'orchestratore "sa" se una Saga non termina e potrebbe implementare meccanismi di timeout e retry centralizzati.

Consistenza ed *endpoint* utente: Dal punto di vista dell'utente finale, la differenza più tangibile riscontrata è stata nella modalità di ottenere il risultato dell'operazione. Con orchestrazione, l'utente riceve l'esito in modo **sincrono**: l'API di creazione ordine risponde definitivamente con successo o fallimento; questo è semplice da gestire anche per

client esterni. Con coreografia, invece, l'API risponde subito prima che la Saga sia conclusa, per cui l'utente deve effettuare chiamate aggiuntive (polling) o attendere notifiche per conoscere l'esito finale. Nel nostro front-end ciò è stato nascosto all'utente con un *spinner* e aggiornamento automatico. Questo evidenzia come l'orchestrazione fornisca un'esperienza più vicina alle transazioni sincrone tradizionali, mentre la coreografia espone maggiormente la natura asincrona. Tuttavia, anche nella Saga orchestrata del progetto si è optato per restituire immediatamente l'ordine e poi l'esito, invece di tenere bloccata la connessione fino a completamento di tutti i passi.

Scalabilità e throughput: Entrambi gli approcci nel progetto potrebbero scalare aumentando istanze dei servizi. Nel pattern coreografico, RabbitMQ facilita un bilanciamento naturale dei consumatori (se ho N istanze di Inventory Service in ascolto sullo stesso queue, i messaggi OrderCreated saranno distribuiti tra di esse). Nell'orchestrazione, invece, l'orchestratore stesso potrebbe diventare un collo di bottiglia sotto carico intenso, dovendo processare sequenzialmente molte Saghe – sebbene replicabile, mantenere l'ordine corretto delle Saga tra orchestratori multipli può essere complesso. Nel nostro caso non si è notato alcun problema di prestazioni poiché il carico era modesto, ma concettualmente la coreografia tende a distribuire il lavoro in parallelo (ogni servizio prosegue indipendentemente dopo aver emesso l'evento) mentre l'orchestrazione serializza almeno parte delle operazioni.

Monitoraggio e logging: Durante l'analisi, un aspetto emerso è la differenza nella facilità di **monitorare** lo stato delle transazioni. L'orchestratore è stato dotato di un log interno degli eventi Saga[87][88] che ha reso agevole stampare o esaminare cosa fosse avvenuto per ogni ordine. Nella coreografia, ogni servizio produce i propri log locali ma non esiste una visione unificata: per ricostruire la sequenza di eventi di una Saga bisogna correlare log di servizi diversi usando magari l'OrderID come chiave. In breve, l'orchestrazione offre **migliore osservabilità** out-of-the-box, mentre la coreografia richiede più sforzo su questo fronte[86].

In sintesi, i risultati confermano le considerazioni teoriche sui due pattern. L'approccio con orchestratore risulta più **coeso e facile da gestire** a livello di logica, adatto probabilmente a flussi transazionali complessi dove si preferisce avere controllo centrale. L'approccio coreografico brilla per **elasticità e disaccoppiamento**, rivelandosi appropriato in contesti di microservizi evolutivi e altamente distribuiti, dove la *business logic* può essere suddivisa in eventi autonomi e si vuole evitare di creare dipendenze forti. Dal punto di vista delle **garanzie di consistenza** entrambe le varianti, se implementate correttamente, assicurano l'assenza di effetti collaterali permanenti in caso di fallimenti parziali. Tuttavia, la coreografia tende verso una consistenza **eventuale** più marcata, mentre l'orchestrazione può talvolta nascondere questi stati all'esterno restituendo solo il risultato finale.

VII. CONCLUSIONI

Il progetto ha permesso di esplorare concretamente l'applicazione del pattern Saga in un caso d'uso realistico di e-commerce a microservizi, fornendo un confronto diretto tra due tipi di coordinamento distribuito. Ho visto come, grazie al pattern Saga, sia possibile garantire consistenza delle operazioni distribuite senza bloccare risorse, tramite transazioni locali con eventi/compensazioni, realizzando una forma di transazione globale più flessibile. L'implementazione ha dimostrato che **entrambe le varianti funzionano** e possono coesistere nel medesimo sistema, ognuna con i propri punti di forza.

In particolare, l'**orchestrazione** si è rivelata vantaggiosa in termini di semplicità logica e monitoraggio, a fronte di una minore flessibilità e potenziale SPOF. La **coreografia** ha mostrato un disaccoppiamento eccellente e nessun ***S**ingle **P**oint **O**f **F**ailure*, al prezzo di una maggiore complessità nel tracciamento dei flussi e nella gestione di comportamenti emergenti man mano che cresce il numero di servizi. La scelta tra i due approcci, dunque, dipende dal contesto: se si privilegia controllo centralizzato e chiarezza, un orchestratore può essere preferibile; se invece si punta a scalabilità estrema, la coreografia è un modello potente.

Limitazioni: Il progetto presentato ha alcune limitazioni intrinseche. Primo, è pur sempre una simulazione semplificata: non sono stati utilizzati database reali, quindi non si sono affrontate questioni come problematiche di isolamenti transazionali tra Saga concorrenti. Secondo, l'applicazione non integra servizi esterni reali (il pagamento è simulato, etc.), quindi lo scenario è limitato. Anche la gestione utenti separata per flusso, seppur utile per dimostrazione, non è comune in sistemi reali (dove si preferirebbe un unico sistema di identity); ciò comporta un piccolo disagio utente nel prototipo, ma era necessario per testare la funzionalità di validazione incrociata. Inoltre, non sono stati testati scenari di concorrenza elevata.

Sviluppi futuri: Vi sono diverse possibili estensioni e migliorie. Un naturale passo successivo sarebbe introdurre una **persistenza con database** per ciascun servizio. Si potrebbe poi aggiungere complessità al processo Saga includendo un microservizio di spedizione/consegna.

In conclusione, il pattern Saga è una soluzione valida per implementare transazioni distribuite consistenti in sistemi distribuiti, e che sia l'approccio con orchestratore sia quello coreografico hanno validità a seconda dei requisiti specifici.

RIFERIMENTI PRINCIPALI

Documentazione progetto Saga Demo (GitHub) – *"SAGA Pattern Project: Orchestration versus Choreography – README"*, StitchMl, 2024. (Descrizione architettura, funzionalità e istruzioni)[18][23]

Microservices.io – Saga Pattern – C. Richardson, *"Pattern: Saga"*, Microservices.io, 2019. (Definizione del pattern Saga, varianti coreografia vs orchestrazione, esempi e discussione)[2][6]

AWS Prescriptive Guidance – Saga – Amazon Web Services, *"Saga choreography vs orchestration patterns"*, 2022. (Linee guida su applicabilità dei pattern Saga, considerazioni su complessità, consistenza ed esempi architetturali)[11][85]

Blogs4Devs – Saga pattern – *"Saga pattern: Choreography and Orchestration"*, Medium.com, 2024. (Introduzione divulgativa al pattern Saga e confronto orchestrazione/coreografia con esempi)[3][7]

Codice sorgente Saga Demo – StitchMl, *Repository GitHub saga-demo*, 2024. (Implementazione in Go dei servizi orchestratore, coreografati e script di test)[17][36]