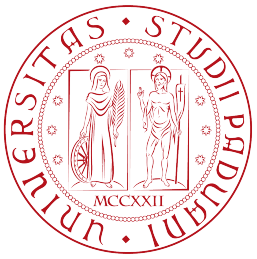

Università degli Studi di Padova

Dipartimento di Matematica

Corso di Laurea in
Informatica

Tesi di Laurea

Sperimentazione di Apache Kafka per l'integrazione funzionale di un'applicazione aziendale

Experimenting with Apache Kafka for the Integration of an Enterprise Application

Relatore:
Prof. Tullio Vardanega

Laureando:
Andrea Dorigo
1170610

Anno Accademico 2020/2021

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Contesto aziendale

1.1 Azienda ospitante

Sync Lab s.r.l.[1] è un'azienda di produzione software, ICT_a[2] e consulenze informatiche nata nel 2002 a Napoli. L'azienda al suo stato attuale presenta un organico aziendale composto da più di 200 risorse, con un fatturato annuo di 12 milioni, una solida base finanziaria e una diffusione sul territorio a livello nazionale. Sync Lab possiede delle significative fette di mercato riguardanti lo sviluppo di prodotti nel settore mobile, videosorveglianza e sicurezza delle strutture informatiche aziendali.

L'azienda ha acquisito numerose certificazioni ISO LL-C per attestare la qualità dei servizi forniti. La certificazione ISO-9001 attesta la gestione della qualità, ISO-14001 la gestione dell'ambiente, ISO-27001 la sicurezza dei sistemi di gestione dati e ISO-45001 la sicurezza nel luogo di lavoro.

Tra i clienti di Sync Lab vi sono ditte a livello nazionale di grandi dimensioni e ampio organico, come Intesa San Paolo, TIM, Vodafone, Enel e Trenitalia che necessitano prodotti di un'elevata sicurezza e adatti al considerevole flusso di dati aziendale.

Sync Lab ha fornito prodotti e consulenze a più di 150 clienti, distribuiti tra clienti diretti e finali, e attualmente possiede cinque sedi (figura 1.1): Napoli, Roma, Milano, Padova e Verona.

L'azienda è suddivisa in molteplici settori dislocati nelle diverse sedi; l'esperienza personale mi ha portato a conoscere il settore dell'*Enterprise Architecture Integration* e del *Technical Professional Services Padova*.

1.2 Servizi offerti dall'azienda

Per comprendere appropriatamente il contesto che ha portato alla nascita del progetto di *stage* è bene conoscere la tipologia di servizi e prodotti che l'azienda offre ai propri clienti. Sync Lab offre per i propri clienti numerosi servizi, tra cui:

- Valutazione e controllo progetti
  - *Planning e project management*; definizione di *Milestone* e *team* di progetto.
  - Valutazione di impatto e *risk analysis*; monitoraggio e *benchmarking*.

[1] *Sync Lab.* URL: https://www.synclab.it/.
[2] *Information and Communication Technologies*

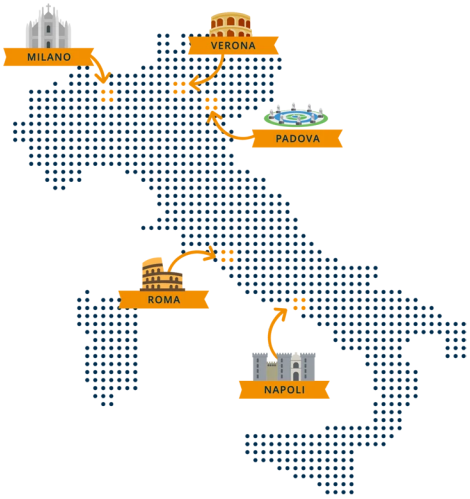

Figura 1.1: Attuali sedi di Sync Lab
Fonte: https://www.synclab.it/

  - Valutazione e controllo di progetti *software* attraverso l'utilizzo di metriche e modelli economici di stima e previsione.

- Sistemi distribuiti di *Enterprise*
  - Progettazione e realizzazione di sistemi distribuiti *Enterprise* in architettura J2EE$_a$[3], EJB$_a$[4], COBRA$_a$[5] e *Web Services*.
  - Progettazione e realizzazione di sistemi basati su MOM$_a$[6] e JMS$_a$[7].
- Tecnologie *Object Oriented*
  - Applicazione delle tecnologie O-O$_a$ all'analisi e progettazione di *software* applicativo e di sistema e nella definizione di architetture distribuite *enterprise*.
  - Utilizzo di metodologie O-O$_a$ per progettazione di applicazioni e processi e UML$_a$, con supporto di strumenti di *modeling*, applicazione e definizione di *Design Pattern*.

1.3 *Way of working* aziendale

1.3.1 Processi interni e fasi progettuali

Durante il percorso di *stage* l'azienda mi ha coinvolto nei processi e fasi progettuali di Formazione, Progettazione architetturale, Codifica, Verifica e Collaudo; le fasi di Manutenzione ed Evoluzione sono state solamente accennate in quanto al di fuori dello scopo del percorso. Questi processi e fasi, nella mia esperienza personale, non sono delineati in modo rigido e rigoroso nei progetti di sperimentazione: ciò ha lo scopo di garantire libertà e flessibilità al *team* di sviluppo e di conseguenza all'intero progetto, attributi necessari in ambito sperimentale.

[3] *Java 2 Platform Enterprise Edition*
[4] *Enterprise Java Bean*
[5] *Common Object Request Broker Architecture*
[6] *Message Oriented Middleware*
[7] *Java Message Service*

Figura 1.2: Processi e fasi progettuali di cui ho avuto esperienza
Fonte: elaborazione personale

Ogni fase è suddivisa in più moduli, per rendere l'avanzamento efficace e quantificabile (in figura 1.2 sono illustrati i processi e fasi relativi allo *stage* in ordine temporale da sinistra verso destra).

Per il processo di Formazione, Sync Lab fornisce materiale sotto forma di corsi *online* tramite le piattaforme Coursera[8] e Udemy[9], oltre a diapositive aziendali che illustrano i concetti chiave del settore EAI_a.

Per la fase di Analisi, Sync Lab mi ha fornito il materiale relativo al caso d'uso da sviluppare, al fine di consentirmi la conoscenza dei requisiti e funzioni finali richieste dal prodotto.

La fase di Progettazione architetturale è una delle più complesse, che necessita di una considerevole esperienza nell'ambito. Per affrontare questa fase, oltre ad approfondire le mie conoscenze riguardo i diversi *design pattern* e *software architecture style*, l'azienda mi ha accompagnato e supportato nella progettazione stessa, con relative motivazioni. Il tutor aziendale Francesco Giovanni Sanges e il responsabile del settore EAI_a Salvatore Dore sono stati di fondamentale aiuto in questa fase.

La fase di Codifica nella mia esperienza personale è risultata abbastanza libera per quanto riguarda le tecnologie e i *software* utilizzati, purché le scelte fossero adeguatamente motivate e adeguate.

La fase di Verifica è stato eseguita, con il supporto del tutor aziendale e del responsabile del settore EAI_a a scadenza settimanale, tramite colloqui *online* o resoconti sulla *online board* di riferimento.

La fase di Collaudo è avvenuta tramite una presentazione *online* e dimostrazione *live* del prodotto sviluppato all'intera azienda.

1.3.2 Gestione di progetto

Sync Lab utilizza una strategia di gestione di progetto (*project management*) ispirata al metodo *Agile*.
La strategia *Agile* è composta da quattro valori fondamentali:

- gli individui hanno importanza maggiore rispetto ai processi;
- il funzionamento del *software* ha la priorità rispetto alla documentazione esaustiva;
- la collaborazione tra fornitore e cliente è più importante della negoziazione del contratto;
- l'adattamento del progetto ai cambiamenti ha la priorità rispetto alla pianificazione.

Questo metodo di lavoro consente all'azienda un alto livello di adattabilità ed evoluzione del prodotto, in base alle richieste del cliente e alle esigenze del *team* di progetto, al fine di fornire soluzioni *ad-hoc* e soddisfare le loro richieste in modo efficiente ed efficace.

[8] *Coursera | Build Skills with Online Courses.* URL: https://www.coursera.org.
[9] *Online Courses - Learn Anything, On Your Schedule | Udemy.* URL: https://www.udemy.com.

1.3.3 Strumenti organizzativi

Per mantenere un alto livello di organizzazione del progetto l'azienda fa uso di molteplici piattaforme dedicate al *project management*, tra cui l'utilizzo di Kanban *board* di progetto. Questo tipo di *board* sono specialmente adatte alla gestione di un progetto che utilizza la metodologia *Agile* vista sopra.

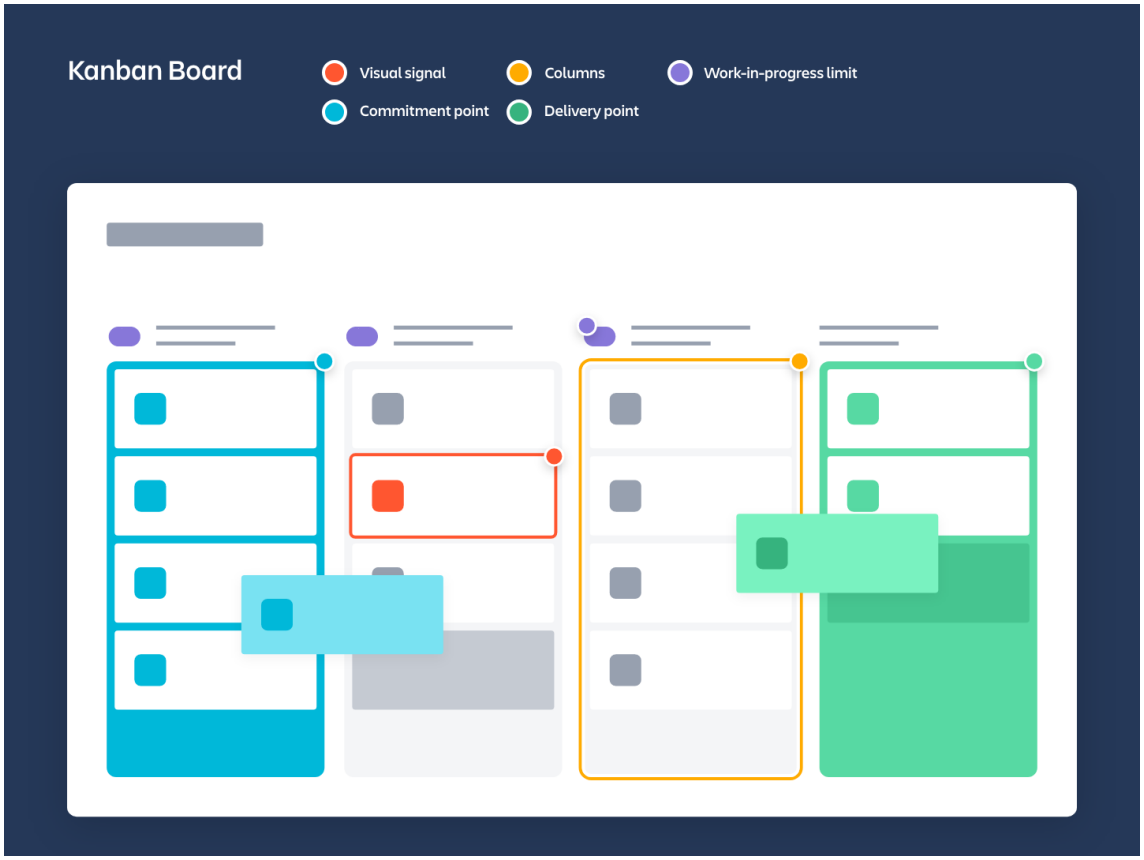

Figura 1.3: Schema riassuntivo degli elementi di una Kanban *board*
Fonte: https://www.atlassian.com/agile/kanban/boards

L'obiettivo di una Kanban *board* è quello di presentare in modo rapido e chiaro una panoramica del progetto. La posizione degli elementi esplicita lo stato di avanzamento delle attività, con un movimento che parte dall'estremo sinistro e si conclude all'estremo destro.
Questo scopo viene raggiunto grazie al delineamento di cinque elementi principali:

- delle *card*, che contengono le diverse attività o *ticket*;
- delle colonne, che rappresentano lo stato d'avanzamento di ogni *card*;
- dei limiti di capacità imposti alle diverse colonne, con lo scopo di evitare impedimenti e rallentamenti nel flusso di lavoro;
- una colonna dedicata al *commitment point* (punto di impegno), in cui vengono raccolte le varie idee o attività che possono essere implementate nel progetto;
- un punto di consegna (*delivery point*), una colonna usualmente posta all'estremo destro della *board*, che indica il completamento del lavoro richiesto per quell'attività.

Il percorso di *stage* ha visto anch'esso l'utilizzo di una Kanban *board* per la gestione del progetto.

1.4 Dominio tecnologico

1.4.1 Dominio aziendale flessibile e indipendente

Il dominio tecnologico aziendale di cui ho avuto esperienza risulta libero e flessibile.

Lo sviluppo del prodotto *software* nell'ambito del EAI_a è fortemente consigliato essere indipendente dal linguaggio di programmazione, dagli strumenti utilizzati per l'esecuzione e sviluppo, e ove possibile, anche dal Sistema Operativo su cui esso esegue. A tal scopo si utilizzano strumenti quali *Virtual Machine* e $container_g$: essi non solo garantiscono l'indipendenza dal Sistema Operativo in uso, ma simulano efficacemente il caso d'uso reale in cui gli eseguibili sono dislocati in più dispositivi come spesso accade per il cliente finale. Il percorso formativo ha visto l'apprendimento di entrambe le tecnologie tramite l'utilizzo dei *software* *Virtual Box*[10] e *Docker*[11], ma infine solo quest'ultima è stata utilizzata durante il progetto.

Nonostante vi sia questa libertà tecnologica riguardo la scelta dei *software* e linguaggi, è necessario tenere in considerazione i concetti principali vincolanti del settore, descritti in seguito.

1.4.2 Sistemi distribuiti

Un concetto fondamentale che caratterizza il dominio tecnologico aziendale, è quello del sistema distribuito, di cui l'azienda ha un forte interesse per soddisfare le necessità di integrazione dei suoi clienti.

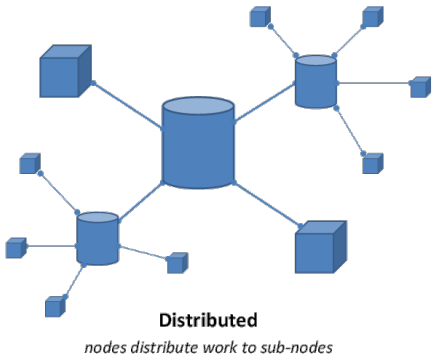

Figura 1.4: Illustrazione di un sistema distribuito
Fonte: https://www.delphitools.info/DWSH/

Un sistema distribuito (figura 1.4) **è una collezione di componenti indipendenti** (spesso collocati in macchine differenti) **che condividono dei messaggi per raggiungere un obiettivo comune.** Un'architettura *software* basata su di un sistema distribuito necessita di una rete che connette tutti i singoli componenti (macchine, *hardware* o *software*), cosicché sia possibile lo scambio dei messaggi.

[10] *Oracle VM Virtual Box*. URL: https://www.virtualbox.org/.
[11] *Empowering App Development for Developers | Docker*. URL: https://www.docker.com/.

Il vantaggio principale di questo tipo di sistemi è la scalabilità orizzontale relativamente economica dei grandi sistemi: per migliorare la *performance* del sistema è sufficiente aggiungere nuove macchine, operazione meno costosa rispetto alla richiesta di *hardware* sempre più potente.

Un secondo vantaggio di fondamentale importanza è la tolleranza ai guasti (*fault tolerance*). In un sistema centralizzato, un guasto nella macchina centrale provoca l'interruzione dell'intero sistema, mentre un sistema distribuito continua a fornire il servizio ove vi sia un guasto in un numero limitato di macchine.

Un ulteriore punto a favore dei sistemi distribuiti è la bassa latenza: il *service client* si connette al nodo più geograficamente vicino, riducendo il tempo di risposta dei sistemi che coprono vasti territori.

Questa soluzione risulta molto favorevole per i clienti di grande dimensione di Sync Lab, per cui i vantaggi elencati superano lo svantaggio del costo iniziale più alto dato dall'installazione del sistema.

1.4.3 *Service Oriented Architecture*$_g$

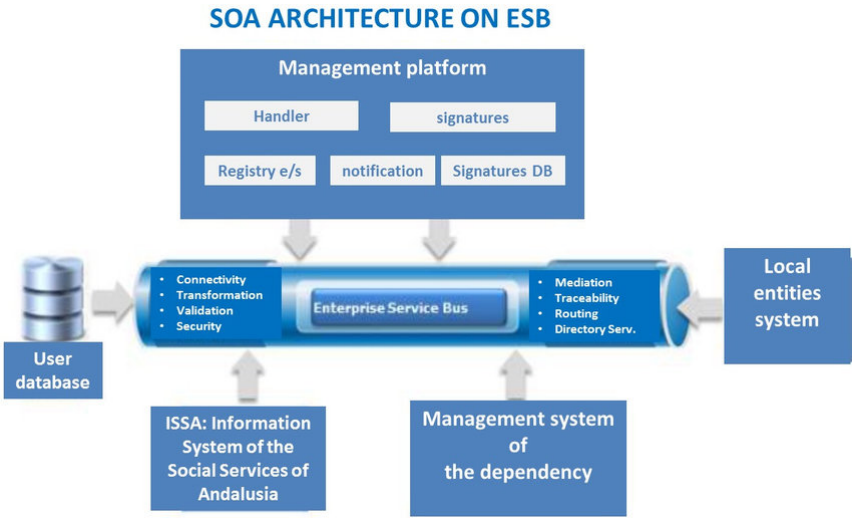

Figura 1.5: SOA$_a$ applicata con un ESB$_a$

Fonte: https://www.researchgate.net/figure/Fig-9-Service-Oriented-Architecture-SOA-Service-Oriented-Architecture-SOA-Service_fig7_330599363

La *Service Oriented Architecture*$_g$ è una tipologia di *software architecture* spesso utilizzata in ambito EAI$_a$ (in figura 1.5, una SOA$_a$ applicata con un ESB$_a$). Essa definisce un modo per rendere i componenti di un architettura *software* riutilizzabili, tramite una decomposizione di un sistema in parti più piccole che comunicano tramite interfacce di servizio che possono essere classificate come sotto-sistemi. **Ogni servizio in una SOA$_a$[12] contiene il codice e le integrazioni dei dati necessari per eseguire una funzione aziendale completa e discreta**. Le interfacce di servizio comportano un *loose coupling*, il che significa che possono essere richiamate con poca o nessuna conoscenza della sottostante modalità di implementazione dell'integrazione. I servizi sono esposti utilizzando protocolli di rete *standard*, come SOAP$_a$[13]/HTTP$_a$ o JSON$_a$[14]/HTTP$_a$, per inviare

[12] *Service Oriented Architecture*$_g$
[13] *Simple Object Access Protocol*
[14] *JavaScript Object Notation*

richieste di lettura o modifica dei dati. I servizi sono pubblicati per consentire agli sviluppatori di trovarli rapidamente e riutilizzarli per assemblare nuove applicazioni in modo modulare.

Questo tipo di architettura consente a Sync Lab di creare sistemi basati sui microservizi$_g$, derivati dalla SOA$_a$, per dare modularità, scalabilità, flessibilità e facilità di manutenzione o aggiornamento ai propri prodotti.

1.4.4 *Messaging pattern*

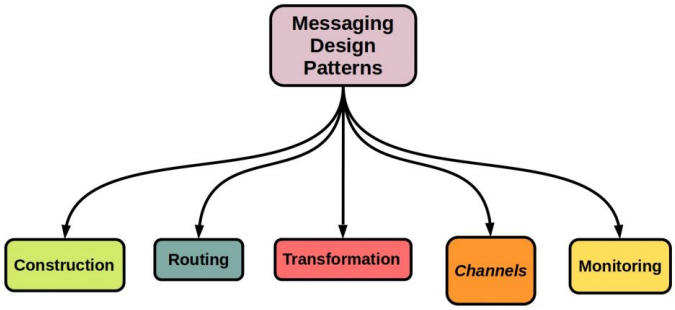

Figura 1.6: Schema di un *messaging design pattern*
Fonte: https://starship-knowledge.com/messaging-patterns

I *messaging pattern* sono alla base di ogni sistema di integrazione. Il *design pattern* si occupa dello scambio di messaggi, come si può intuire dal nome. Un messaggio è un pacchetto di dati atomico che può essere trasmesso attraverso un canale in modalità asincrona. Un canale è un condotto virtuale che connette un servizio che invia i dati ad uno che li riceve.

Nella maggior parte dei sistemi di integrazione, il dato potrebbe necessitare diverse elaborazioni e non conoscere direttamente il destinatario del messaggio; è possibile applicare i concetti relativi alla *pipes and filters architecture* inserendo un ricettore comune, un *message router* che si occupa del *routing* di tutti i messaggi e del passaggio di essi attraverso i vari filtri e trasformazioni. Un *message broker* è un modulo *software* spesso posto all'interno di una MOM$_a$; si occupa, oltre al *routing*, di validazione, memorizzazione ed invio dei messaggi alle destinazioni appropriate

1.4.5 *Enterprise Service Bus*

Un ESB$_a$ è un *pattern* architetturale in cui un *software* centrale consente l'integrazione tra diverse applicazioni (figura 1.7). Esso si occupa della comunicazione, trasformazione e conversione dei dati all'interno di una SOA$_a$; è una tipologia più evoluta di *message broker*. Senza strumenti di questo tipo, la *Service Oriented Architecture*$_g$ porterebbe ad un sistema composto semplicemente da un gruppo di servizi; in questo caso infatti, ogni servizio dovrebbe occuparsi

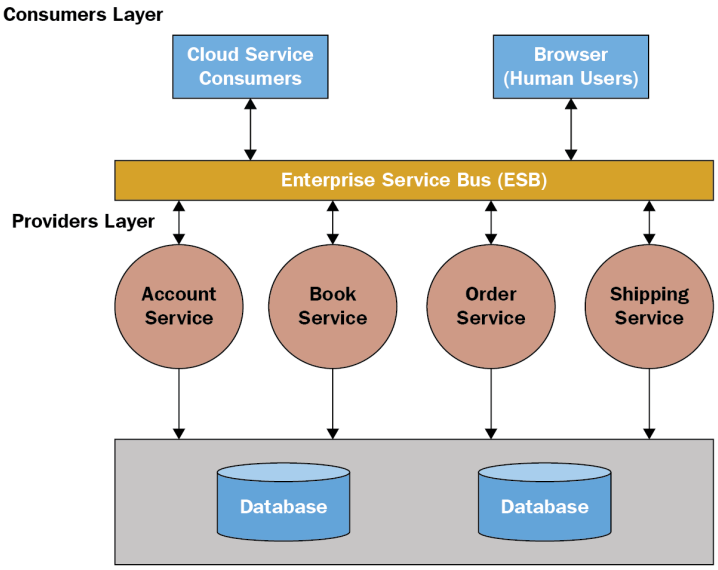

Figura 1.7: Illustrazione esemplificativa di un ESB_a

Fonte: https://subscription.packtpub.com/book/application_development/9781789133608/1/ch01lvl1sec12/service-oriented-architecture-soa

dello scambio di messaggi con tutti gli altri ($P2P_a$[15]) creando, nei sistemi più grandi, problemi per quanto riguarda l'estensione e manutenibilità dei servizi.

1.4.6 EAI_a e *Middleware*

Il percorso di *stage* intrapreso è associato al settore del *Enterprise Architecture Integration*, che si occupa principalmente del EAI_a (*Enterprise Application Integration*$_g$, figura 1.8) ovvero dell'integrazione funzionale di applicazioni aziendali per una clientela di grandi dimensioni (come un'azienda di telecomunicazioni), tramite sistemi di integrazione e *Middleware*.

I *Middleware* e sistemi di integrazione prodotti comprendono l'utilizzo di molteplici linguaggi e tecnologie in continua evoluzione.
Dal sito di Red Hat[16]:

> *Il middleware è un software che fornisce alle applicazioni servizi e capacità frequentemente utilizzati, tra cui gestione dei dati e delle API, servizi per le applicazioni, messaggistica e autenticazione.*
> *Aiuta gli sviluppatori a creare le applicazioni in modo più efficiente e agisce come un tessuto connettivo tra applicazioni, dati e utenti.*
> *Può rendere conveniente lo sviluppo, l'esecuzione e la scalabilità di applicazioni alle organizzazioni con ambienti multi cloud e containerizzati.*

I *Middleware* pertanto vedono due importanti utilizzi nel settore EAI_a:

- **Integrazione su più livelli:** i *Middleware* connettono i principali sistemi aziendali interni ed esterni. Capacità di integrazione quali trasformazione, connettività, componibilità e mes-

[15] *Point To Point*
[16] *Cos'è il Middleware?* URL: https://www.redhat.com/it/topics/middleware/what-is-middleware.

Figura 1.8: Concetti principali del EAI$_a$

Fonte: https://commons.wikimedia.org/wiki/File:KrisangelChap2-EAI.png

saggistica *enterprise*, abbinate all'autenticazione SSO$_a$[17], aiutano gli sviluppatori a estendere tali capacità su diverse applicazioni.

- **Flussi di dati:** le API$_a$[18] rappresentano una modalità per condividere i dati tra le applicazioni. Un altro approccio è quello del flusso di dati asincrono, che consiste nella replica di un *set* di dati in un livello intermedio, da cui i dati possono essere condivisi con più applicazioni.

- **Ottimizzazione di applicazioni esistenti:** con l'adozione del *Middleware*, gli sviluppatori possono trasformare le applicazioni monolitiche esistenti in applicazioni *cloud native* o a microservizi$_g$, mantenendo i validi strumenti già in uso ma migliorandone prestazioni e portabilità (figura 1.8).

1.4.7 *Container* e *Virtual Machine*

Come anticipato nella sezione precedente, tra le tecnologie più utilizzate in questo settore aziendale vi sono molte piattaforme che permettono di simulare ambienti distribuiti su più macchine fisiche, ove possibile anche indipendenti dal Sistema Operativo su cui viene eseguito il prodotto *software*.

La simulazione di questi *distributed environment* avviene grazie a sistemi basati sul concetto di *container*$_g$ (come Docker e Kubernetes) oppure interi Sistemi Operativi che vengono eseguiti all'interno di una *Virtual Machine*. Il vantaggio principale di queste tecnologie è che rendono l'esecuzione del *software* al loro interno completamente indipendente dall'ambiente circostante, eliminando problemi di OS$_a$[19] differenti tra i componenti del *team* o tra l'azienda e i clienti o divergenze nelle dipendenze con relative versioni. Un *container*$_g$ o una *Virtual Machine* contengono tutto il necessario affinché sia possibile eseguire il *software* al suo interno su diverse macchine fisiche (o anch'esse virtuali).

[17] *Single Sign On*
[18] *Application Programming Interface*
[19] *Operating System*

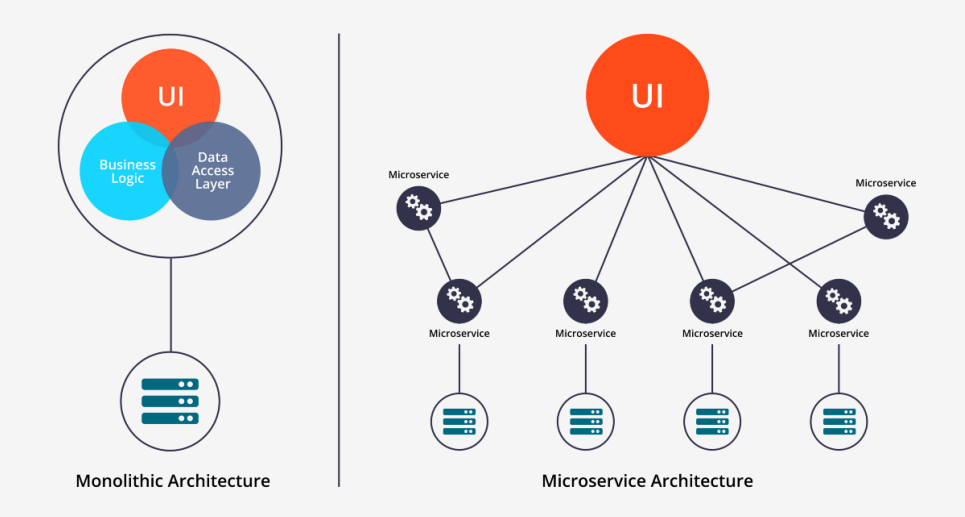

Figura 1.9: Dalle classiche soluzioni monolitiche ai moderni sistemi a microservizi$_g$

Fonte: https://aymax.fr/en/why-a-microservices-architecture/

Queste piattaforme non solo rendono agevole l'esecuzione del *software* prodotto, ma anche lo sviluppo: la condivisione, *debugging* e manutenzione risultano più agevoli grazie alla condivisione dell'intero *container*$_g$ o VM$_a$ con gli altri membri del team.

Inoltre, si adattano particolarmente bene a simulare l'ambiente distribuito, un concetto fondamentale nel settore del EAI$_a$; infatti è sufficiente generare molteplici *container*$_g$ o VM$_a$ sulla stessa macchina fisica per simulare un sistema composto da più macchine fisiche distinte, minimizzando l'utilizzo di risorse senza compromettere il risultato del prodotto finale. È così possibile per l'azienda riprodurre un sistema complesso che si avvicina alle risorse ed esigenze effettive del cliente, che usualmente possiede molti computer e server dislocati.

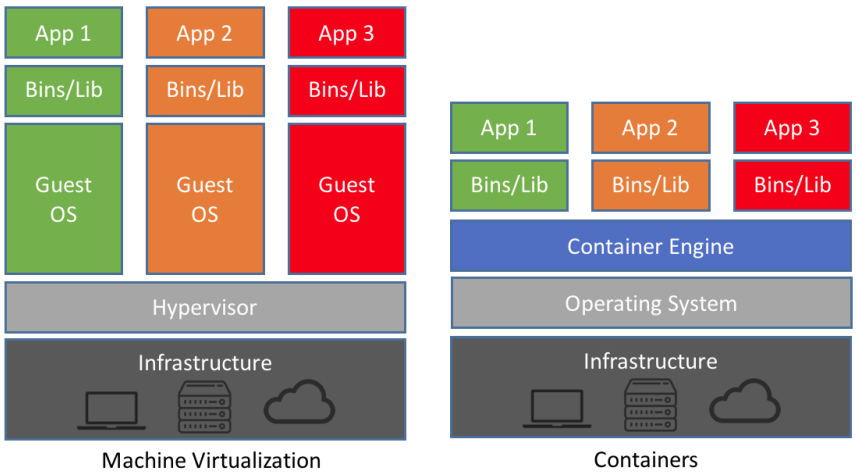

Figura 1.10: Differenti implementazioni legate alle VM$_a$ e *container*

Fonte: https://pawseysc.github.io/container-workflows/01-docker-intro/index.html

La figura 1.10 rappresenta graficamente le due diverse implementazioni delle due tecnologie. Vi

sono dunque delle notevoli differenze, vantaggi e svantaggi tra l'utilizzo dell'una e dell'altra, tra cui:

1. i *container* sono più rapidi delle VM_a nell'esecuzione;
2. i *container* sono più leggeri delle VM_a in termini di memoria;
3. i *container* sono più adatti a simulare un'architettura a $microservizi_g$, dato la relativa semplicità ed efficienza rispetto ad una VM_a;
4. le VM_a sono considerate tendenzialmente più sicure dei *container*
5. le infrastrutture e strumenti di gestione di grandi quantità di VM_a sono più consolidate dei corrispettivi strumenti associati ai *container*.

Durante il percorso di stage ho approfondito le mie conoscenze riguardo entrambe le tecnologie, optando per l'utilizzo di *container* all'interno del mio progetto di *stage*, poiché più efficiente considerate le risorse a mia disposizione. La scelta della piattaforma di *container* è stata quella di Docker. Più precisamente, ho utilizzato l'estensione docker-compose[20] per gestire in modo elegante generazione e collaudo di più servizi indipendenti: non solo questo *software* consente di creare velocemente una rete di $container_g$ comunicanti su di una rete isolata, ma rende anche rapido ed efficiente lo sviluppo, grazie alla possibilità di modificare e riavviare rapidamente un singolo servizio.

```yaml
version: "3.9"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
```

Figura 1.11: Frammento di codice che espone un esempio di ambiente *multi-container* con docker-compose
Fonte: elaborazione personale

Come si può vedere in figura 1.11, è relativamente semplice e veloce creare un ambiente composto da due container entrambi connessi alla stessa rete locale di *default* di Docker (il servizio intitolato `web`, composto a partire dalla cartella corrente e accessibile alla porta 5000, e il servizio `redis`, generato a partire da un'immagine predefinita).

Il caso d'uso realizzato nel mio percorso ha modellato grazie ai *container* un sistema a $microservizi_g$ che simula le risorse di un grande cliente gestore di telecomunicazioni, secondo una visione coerente con il tipo di clientela reale dell'azienda.

1.4.8 Introduzione di Apache Kafka

Attualmente Sync Lab sta intraprendendo dei percorsi di *stage* per implementare nuove tecnologie nei prodotti *Middleware*. Uno di questi prodotti è Apache Kafka.

[20] *Overview of Docker Compose | Docker Documentation*. URL: https://docs.docker.com/compose/.

Kafka è una piattaforma di *event streaming*, un sistema distribuito e moderno basato sugli eventi anziché su di una soluzione più classica come può essere quella del *request/response* e P2P$_a$.

L'*event streaming* è una pratica focalizzata sul catturare dati in *real-time* da diverse fonti come *database*, sensori e dispositivi mobili sotto la forma di un flusso di eventi; garantisce uno scambio continuo di informazioni e la loro interpretazione.

Apache Kafka è un *software* fondato sul *design pattern* del *publish/subscribe*. Contiene infatti delle API$_a$ intitolate *Producer* e *Consumer*, che un utente della piattaforma può utilizzare rispettivamente per pubblicare degli eventi o riceverli istantaneamente. Permette inoltre di memorizzare questo flusso di dati in modo affidabile, *fault tolerant*, e duraturo, oppure di processare il flusso per trasformarlo.

Un evento è definito come un'occorrenza significativa o un cambiamento di stato del sistema, e nell'ottica del *messaging design pattern* occuperebbe il ruolo di un messaggio atomico.

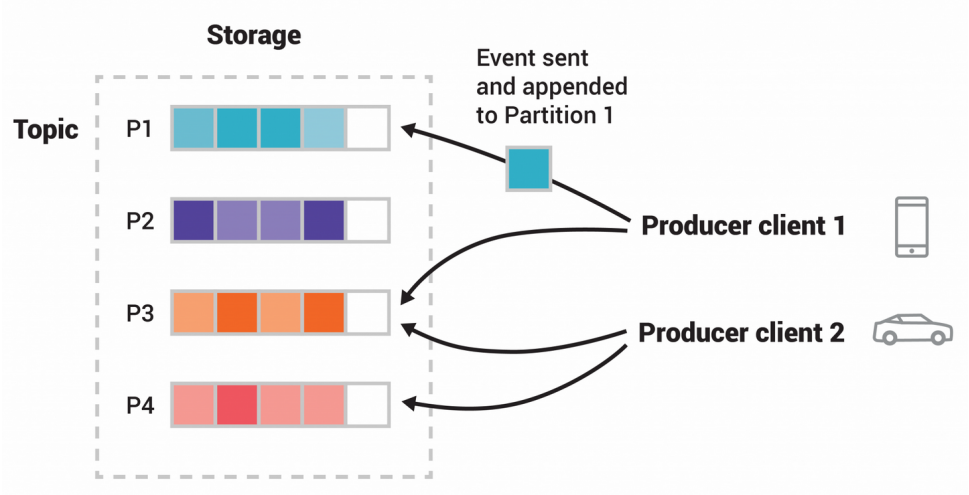

Figura 1.12: Schema di un topic contenente diversi eventi, diviso in partizioni (P), con molteplici *producer*
Fonte: https://kafka.apache.org/intro

Essi sono organizzati all'interno di *topics*, delle liste ordinate di eventi in cui molteplici applicazioni possono scrivere o leggere tali eventi simultaneamente. In figura si può vedere un esempio di *topic*, diviso in diverse partizioni, in cui due *producer* inseriscono dati.

Kafka viene eseguito come un gruppo (*cluster*) di *server* distribuiti. Alcuni di questi *server* sono chiamati *broker*, e insieme compongono il livello di memorizzazione (*storage layer*). Altri *server* sono dedicati all'esecuzione di Kafka Connect, un componente essenziale per l'importazione dei dati da altre fonti; esso è molto rilevante nel campo dei *Middleware*, poiché permette di integrare Kafka all'interno di sistemi pre-esistenti in modo graduale, con un investimento iniziale parziale.

I *client* invece, permettono di sviluppare applicazioni distribuite a microservizi$_g$ che leggono, scrivono e processano il flusso di dati; questi *client* mettono a disposizione delle interfacce per molti linguaggi e *software* differenti, tra cui Java, Scala, Python, C++/C oltre a delle REST$_a$ API$_a$.

Il sistema è molto adattabile, dato che può essere installato anche su *Virtual Machine*, *container*$_g$, ambienti *cloud* e addirittura *bare-metal hardware*. La piattaforma è costituita da un sistema distribuito di *server* e *client* che comunicano attraverso un protocollo *network* TCP$_a$[21] ad alta *performance*.

[21] *Transmission Control Protocol*

Il *software* ha dimostrato negli anni recenti un notevole successo in diversi campi[22], come quello del flusso di *Big Data*, del monitoraggio e dell'elaborazione dati in tempo reale. L'adozione del *software* nell'ambito del EAI_a è in crescita dato le dimostrate qualità nel gestire grandi moli di dati: le sue performance, sicurezza e scalabilità sono i punti che hanno portato il *software* al suo attuale successo.

1.5 Innovazione all'interno dell'azienda

Questo contesto dell'integrazione aziendale porta dunque l'impresa ad avere un'importante propensione all'innovazione, talvolta esplicitamente richiesta dai clienti.

Una direzione dell'evoluzione attuale nel settore EAI_a riguarda la migrazione verso sistemi sempre più distribuiti, in accordo con l'ambiente di lavoro descritto nelle sezioni precedenti, in grado di gestire efficacemente ed in tempo reale flussi di dati in continua crescita e appartenenti al mondo del *Big Data*.

L'avanguardia tecnologica è pertanto uno dei principali temi dell'azienda, che garantisce che essa rimanga sempre competitiva sul mercato dei sistemi di integrazione e nel settore dell'EAI_a. L'implementazione di nuove tecnologie, come può essere una *Event Driven Architecture*$_g$ basata su Kafka, permetterebbe a Sync Lab di offrire soluzioni sempre più moderne, scalabili e affidabili ai propri clienti.

[22]Fonte: https://kafka.apache.org/powered-by

Capitolo 2

Kafka nell'integrazione aziendale

2.1 Obiettivi aziendali

2.1.1 Migrazione verso un *Event Driven Architecture*

Per soddisfare le richieste dei clienti ed essere sempre competitiva e all'avanguardia, una priorità di Sync Lab sono le esplorazioni tecnologiche e di prodotto anche tramite l'utilizzo di percorsi di *stage* insieme ai laureandi, come accaduto nella mia esperienza. Questi percorsi consentono all'azienda non solo di testare l'utilizzo di nuovi *software* ma anche di conoscere e mettere alla prova le capacità del laureando in vista di una potenziale assunzione al termine dello *stage*.

Nel settore dell'*Enterprise Application Integration*$_g$, l'evoluzione tecnologica è diretta verso soluzioni sempre più distribuite e con un flusso di dati in continuo aumento. Uno degli obiettivi specifici nell'area EAI$_a$ di Sync Lab è pertanto quello di individuare un *software* o tecnologia in grado di soddisfare i bisogni dei clienti di gestire un flusso di dati di dimensioni molto maggiori a quelle attuali, tramite architetture a messaggio che utilizzano servizi distribuiti.

Figura 2.1: Illustrazione di un sistema basato sul P2P$_a$

Fonte: https://news.pwc.be/messaging-architecture-with-salesforce/

Nei *Middleware* per i sistemi di integrazione, l'aumento del flusso di dati e lo spostamento verso strutture distribuite provoca una difficoltà nella trasmissione dei dati tra i diversi servizi.

Nell'ambito dell'integrazione per un cliente di piccole dimensioni, in cui i dati circolano tra un numero di componenti limitato, può essere sufficiente un'architettura di tipo $P2P_a$[1].

Nel caso di un cliente di maggiori dimensioni tuttavia, questo approccio rende la manutenzione e gestione del flusso di dati molto difficoltoso e costoso in termini di risorse (figura 2.1), dato il grande numero di collegamenti tra i vari punti.

Una delle soluzioni che viene maggiormente implementata per risolvere questo problema è la migrazione verso una EDA_a (*Event Driven Architecture*$_g$), un'architettura basata sugli eventi in grado di scambiare dati tra punti multipli.

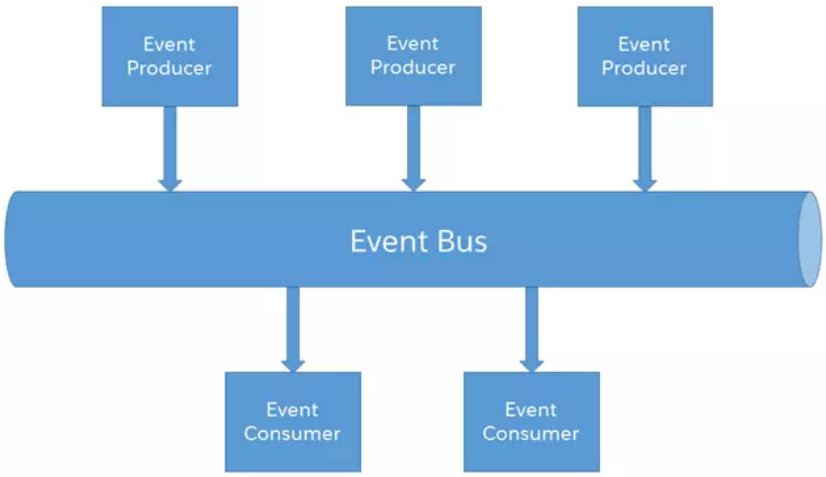

Figura 2.2: Illustrazione di un sistema basato sulla EDA_a
Fonte: https://news.pwc.be/messaging-architecture-with-salesforce/

Questo tipo di architettura è pertanto definita per gestire la produzione, il rilevamento e la reazione agli eventi (figura 2.2), grazie ad un *Design Pattern* di tipo *Publish/Subscribe*, eliminando i problemi delineati precedentemente causati dal sistema $P2P_a$. Questa architettura prevede l'utilizzo di servizi chiamati *Producer*, il cui scopo è fornire dati al *event bus* centrale. Una volta che i dati sono inseriti all'interno del *bus* centrale, ogni servizio in ascolto (*Subscriber*) li riceverà idealmente in tempo reale.

2.1.2 Kafka come *Middleware*

Per soddisfare le esigenze di innovazione, l'azienda ha avviato un percorso per indagare le capacità del *software* Apache Kafka nell'ambito dell'integrazione aziendale.

Apache Kafka si integra in modo ottimale in molti sistemi basati sul *messaging pattern* e una EDA_a, in cui lo scambio affidabile di dati tra numerosi servizi in tempo reale è essenziale (in figura 2.3 è illustrato un caso d'uso esemplificativo di un sistema distribuito basato su Kafka).

L'interesse di Sync Lab nel *software* risiede dunque nell'utilizzo di Kafka come un *Middleware* per soddisfare i problemi di integrazione aziendale e re-ingegnerizzare i flussi di dati preesistenti, ovvero sviluppare un nuovo sistema che consenta la comunicazione tra differenti servizi con un rapido flusso di dati tra di essi.

[1] *Point To Point*

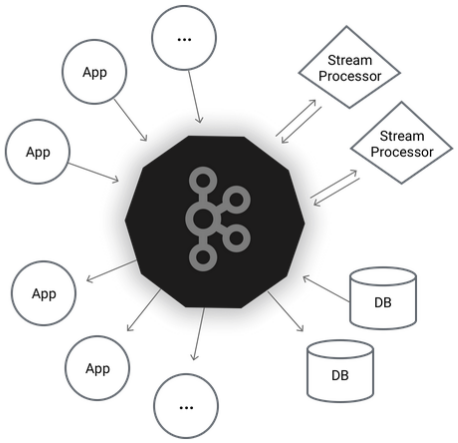

Figura 2.3: Illustrazione di Apache Kafka in un caso d'uso esemplificativo
Fonte: https://iotbyhvm.ooo/apache-kafka-a-distributed-streaming-platform/

L'azienda dunque ha avviato un percorso per testare le capacità di Kafka rispetto agli attuali strumenti utilizzati nel settore, per valutare i vantaggi e svantaggi che l'adozione di tale *software* può fornire al cliente.
Sono numerosi i vantaggi che Kafka può portare nel settore, tra cui:

- gestione rapida e performante di un enorme flusso di dati;
- scalabilità;
- sicurezza riguardo la persistenza dei dati;
- semplice integrazione e affiancamento a sistemi già esistenti;
- l'essere una piattaforma *open source*;
- processazione dei dati in tempo reale integrata.

2.2 Motivazioni e obiettivi personali

2.2.1 Scelta del percorso

Una delle ragioni che mi ha portato a scegliere questo percorso di *stage* è l'interesse verso Apache Kafka. L'utilizzo della piattaforma di *event streaming* è sempre più in crescita, come l'evoluzione verso sistemi sempre più distribuiti e a microservizi$_g$.

Un altro fattore fondamentale per la scelta del percorso sono stati la familiarità con l'azienda, una buona valutazione del metodo di lavoro aziendale e la libertà di sviluppo concessa: ho ritenuto importante la possibilità di elaborare personalmente un'architettura associata al caso d'uso con una visione ad alto livello, anziché il semplice sviluppo di un *software* predeterminato e dal percorso strettamente imposto.

2.2.2 Obiettivi personali

L'obiettivo fondamentale dello *stage* è colmare il divario tra il mondo accademico e quello lavorativo. Grazie al percorso di *stage* in una ditta esterna ho avuto l'opportunità di conoscere l'ambiente di lavoro di un'azienda nel campo ICT_a, facilitandomi l'inserimento nel mondo del lavoro.

Un altro obiettivo è ottenere una formazione riguardo la tecnologia di Kafka, che ritengo possa arricchire fortemente le mie capacità e *skill* professionali. Sono pertanto interessato a sviluppare la mia conoscenza riguardo l'utilizzo e le implicazioni di questa tecnologia in rapida espansione, la cui formazione potrà essermi utile in molti campi anche al di fuori degli obiettivi dell'azienda ospitante lo *stage*.

2.3 Pianificazione del percorso di *stage*

2.3.1 Obiettivi dello *stage*

Il percorso di *stage* offerto dall'azienda si inserisce all'interno della strategia aziendale più ampia descritta sopra. Al fine di esplorare la tecnologia di Apache Kafka nell'ambito di un *Middleware* per l'integrazione aziendale, l'azienda ha proposto un percorso di ***stage*, il cui obiettivo è la re-ingegnerizzazione di un flusso di dati asincrono, utilizzando un'architettura basata su Kafka all'interno di un caso d'uso simulato tramite servizi indipendenti.**

Lo stagista ha il compito di osservare, testare e verificare che il *software* possa svolgere alcuni compiti inerenti all'area del EAI_a, analizzando alcuni casi d'uso presenti in un *Middleware* aziendale in ambito *telco*. Il percorso prevede una durata di 300 ore lavorative.

2.3.2 Prodotti attesi

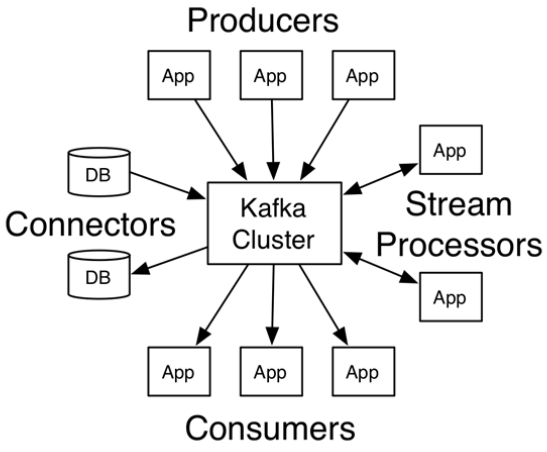

Figura 2.4: Illustrazione di un sistema a servizi con Kafka
Fonte: https://kafka.apache.org/20/documentation.html

I prodotti attesi al termine dello *stage* sono dunque associati alla realizzazione di tre flussi di integrazione, basati su dei casi d'uso reali, per la gestione dei paradigmi di integrazione asincrono,

asincrono con *callback* (due requisiti obbligatori), e sincrono ove fosse disponibile del tempo aggiuntivo se ritenuto opportuno; durante il percorso, considerati il contesto e le opportunità offerte dal *software*, questo obiettivo verrà sostituito per testare delle funzionalità aggiuntive di Kafka.

Il prodotto *software* finale sarà un sistema basato su servizi indipendenti costruito con un'architettura di tipo EDA$_a$ tramite l'utilizzo di Kafka (figura 2.4).

2.3.3 Contenuti formativi previsti

La realizzazione di questi prodotti necessita una sostanziale formazione dello stagista riguardo i principali concetti del settore del *Enterprise Application Integration* e l'utilizzo della piattaforma di *event streaming* Kafka. Più precisamente, i contenuti formativi previsti durante questo percorso di *stage* sono i seguenti:

- Concetti chiave del *Enterprise Application Integration*$_g$;
- *Design architetturali*;
- Cenni di *Networking* applicato alle architetture distribuite;
- Architetture di Integrazione e *Middleware*;
- Apache Kafka.

2.3.4 Interazione tra studente e referenti aziendali

Regolarmente sono previsti incontri *online* settimanali (tramite la piattaforma Google Meet) con il tutor aziendale Francesco Giovanni Sanges, il responsabile dell'area EAI$_a$ Salvatore Dore e gli esperti delle tecnologie affrontate.

Lo scopo di questi incontri è quello di verificare lo stato di avanzamento, chiarire gli obiettivi ove necessario, affinare la ricerca e aggiornare la pianificazione iniziale.

2.3.5 *Way of working* di progetto

Kanban board di progetto

All'inizio dello *stage* ho stabilito un *way of working*, ovvero un metodo di lavoro da mantenere per tutta la durata del percorso, insieme al tutor aziendale, il responsabile dell'area EAI$_a$ e gli esperti del settore.

L'organizzazione efficiente del progetto è dunque garantita dall'utilizzo di vari strumenti a supporto, quali Kanban *board* (come Click Up[2] per la gestione di progetto e Notion[3] per le prenotazioni della postazione di lavoro in sede), *chat* (come Google Chat[4]) per i confronti rapidi con gli altri membri interni al progetto ed e-mail per le comunicazioni con componenti esterni al progetto.

Lo strumento più utilizzato in ambito organizzativo durante il mio percorso è la Kanban *board* di Click Up, che ha permesso la gestione, il confronto, la quantificazione e la verifica del progresso. Tra le tante opzioni disponibili, Click Up possiede numerosi vantaggi rispetto alla concorrenza: la piattaforma è ricca di funzionalità, pulita nell'esposizione dello stato del progetto e la maggior parte delle sue funzioni sono gratuite. La figura seguente illustra, a titolo esemplificativo, uno *screenshot* che raffigura lo stato dell'avanzamento.

[2] *ClickUp™ | One app to replace them all.* URL: https://clickup.com.
[3] *Notion - The all-in-one workspace for your notes, tasks, wikis, and databases.* URL: https://www.notion.os.
[4] *Google Chat.* URL: https://chat.google.com.

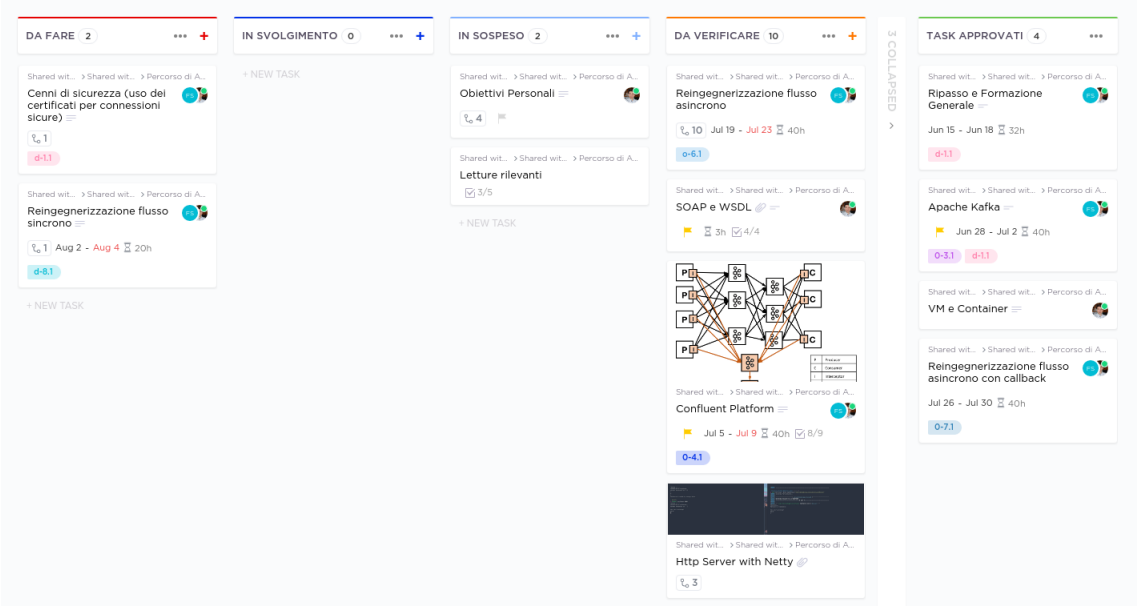

Figura 2.5: Kanban *board* del progetto di *stage*
Fonte: elaborazione personale

Le attività (*task*) vengono inizialmente create nella colonna "DA FARE" dal tutor aziendale o dal sottoscritto, ove ritenuto opportuno. Per dimostrare l'avanzamento il *task* si sposta verso destra a seconda dello stato raggiunto; ho la responsabilità del cambiamento di stato fino alla colonna "DA VERIFICARE", dopodiché la verifica e lo spostamento del *task* in "TASK APPROVATI" è compito del tutor aziendale, con conseguenti approvazione finale e conclusione dell'attività.

Per tenere traccia del lavoro svolto riguardante una specifica attività, ho utilizzato le *card* messe a disposizione dalla piattaforma, che mi hanno consentito di delineare precisamente la pianificazione e la descrizione dettagliata dell'avanzamento del singolo *task*.

Questa *card* (di cui è presente un esempio in figura 2.6) contiene una casella di testo per inserire una descrizione e appunti utili, una *checklist* approfondita e una colonna che mantiene uno storico dei commenti; quest'ultima colonna non solo permette a me di mantenere un'importante resoconto sul lavoro svolto, ma consente anche al tutor aziendale e esperti del settore di quantificare il progresso e di fornire un aiuto rapido e contestuale.

Per la condivisione di codice è stato possibile utilizzare uno strumento a mia discrezione, e pertanto ho utilizzato Git per creare una *repository* di cui successivamente ho eseguito l'*upload* in modalità privata (secondo indicazioni aziendali) su Github. L'inserimento del codice identificativo del `commit` , all'interno di un commento allo scopo di *log*, ha favorito ulteriormente il tracciamento del progresso e reso agevole l'eventuale supporto da parte degli esperti aziendali.

All'inizio del percorso, il tutor aziendale e il responsabile del EAI_a hanno creato delle *card* contenenti le attività previste per ogni settimana (*task*), al fine di fornire una struttura generale del progetto. All'interno di questi *task* vi sono i concetti chiave, attività previste e obiettivi settimanali da seguire per garantire l'efficacia del prodotto finale. Oltre a questi *task* principali, ho potuto creare dei *task* ausiliari e dei *sub-task* per descrivere più adeguatamente l'attività in corso.

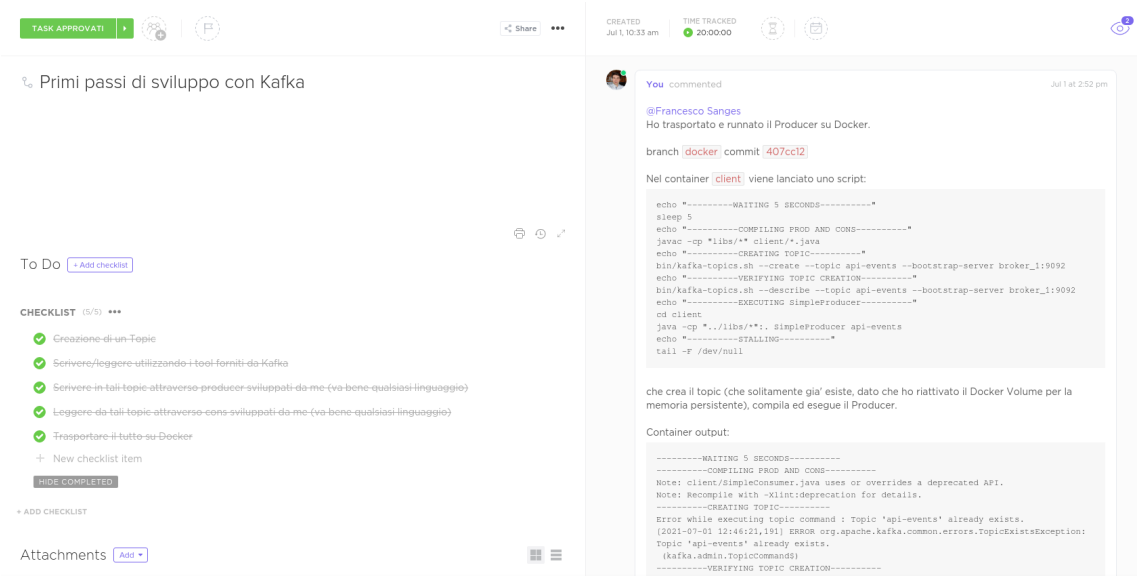

Figura 2.6: Esempio di un'attività del processo di Formazione
Fonte: elaborazione personale

Meeting con gli esperti di Sync Lab

Ogni settimana è previsto un *online meeting* per la verifica del progresso ove necessario, la risposta a eventuali questioni sollevate e spiegazioni riguardo lo sviluppo della settimana successiva. Alcune di queste videoconferenze hanno coinvolto la partecipazione di altri esperti che mi hanno aiutato a comprendere meglio il caso d'uso da re-ingegnerizzare, riassumendo lo stato attuale del sistema d'integrazione per uno dei clienti con relativi *file* utilizzati.

Per mantenere alto il livello di organizzazione, efficienza ed efficacia, all'inizio di ogni giornata lavorativa ho creato un breve piano giornaliero con successivo resoconto a fine giornata. Questo ha permesso al tutor di verificare rapidamente il corretto avanzamento del processo in corso e a me di mantenere il *focus* su di esso.

Auto-miglioramento personale

A partire dalle prime settimane del percorso di *stage*, **ho deciso di integrare nel mio *way of working* il ciclo di Deming, altrimenti detto PDCA$_a$, ovvero un metodo di lavoro legato al concetto di auto-miglioramento (*Plan Do Check Act*).** Ho applicato il metodo PDCA$_a$ per il miglioramento delle mie capacità organizzative, comunicative e formative, in intervalli di 1 o 2 settimane a seconda del miglioramento scelto. Tra i miglioramenti più rilevanti cito l'implementazione della *Pomodoro Technique* per tutta la durata del percorso. La *Pomodoro Technique* è una tecnica di gestione del tempo che prevede la suddivisione del lavoro in intervalli di tempo stabiliti, ed è costituita dai seguenti passaggi:

1. scelta del compito da eseguire;
2. impostazione del *timer* a 25 minuti;
3. lavoro riguardo il relativo compito, senza alcuna distrazione data da agenti esterni, fino allo scadere dei 25 minuti;

4. pausa per un intervallo di 5 minuti;

5. ogni quattro cicli completi ("pomodori" di 30 minuti), prendere una pausa più lunga di circa 15/30 minuti.

La tecnica, che ha presentato delle difficoltà iniziali nell'inserimento all'interno del mio metodo di lavoro, si è rivelata per me un successo nel lungo termine, che ha portato a un minore affaticamento e maggiore concentrazione nel progetto, a cui ha pertanto conseguito una maggiore efficienza lavorativa.

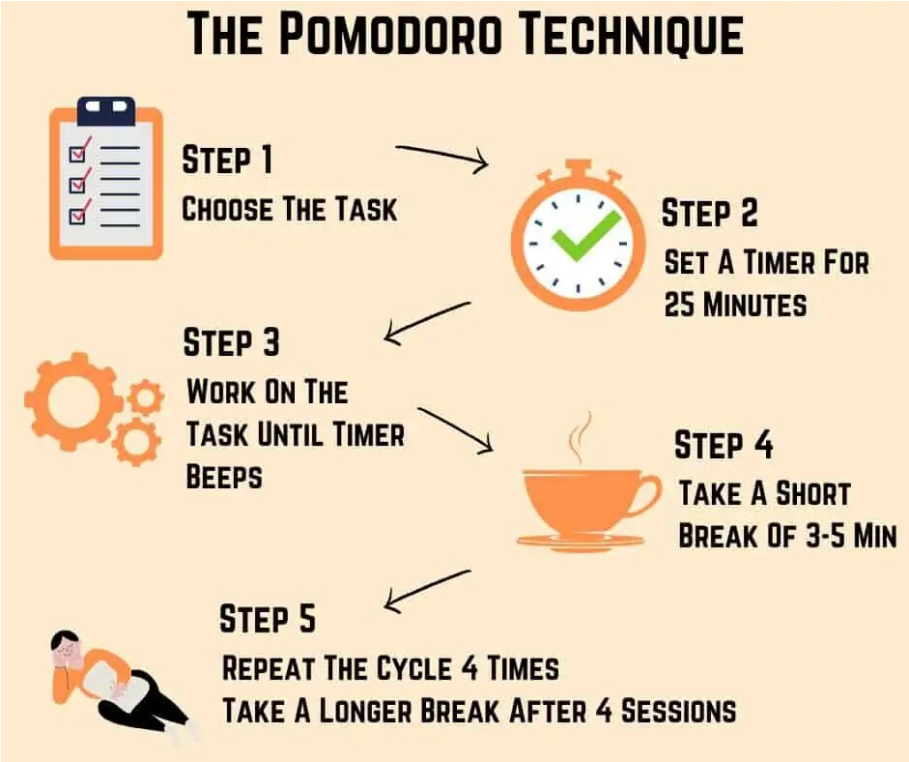

Figura 2.7: *Pomodoro Technique*
Fonte: https://productiveclub.com/pomodoro-technique/

Se correttamente applicata, può portare a numerosi vantaggi, quali:

- consente di mantenere un alto livello di concentrazione.
- dedicare un intervallo all'obiettivo a lungo termine aiuta a visualizzare lo scopo del progetto, decidere se mantenersi al piano o modificarlo, e mantenere la giusta direzione evitando di andare *out of focus* di progetto. Ho particolarmente apprezzato questo punto, che mi ha mantenuto focalizzato sull'obbiettivo finale. All'inizio di ogni settimana ho dedicato un'ora alla pianificazione settimanale, rapportandola a quella del percorso completo e suddividendo i compiti in base ai giorni disponibili.
- riduce considerevolmente le distrazioni nei 25 minuti di lavoro.
- aiuta il tracciamento delle risorse; grazie alla suddivisione in intervalli di tempo ben stabiliti, è semplice annotarsi il tempo impiegato per un determinato *task*. Questo è un altro punto

che porta a molti vantaggi: combinato con una stima iniziale delle risorse previste, permette di paragonare il tempo pianificato con il tempo effettivo impiegato. Nel lungo termine, mi ha permesso una stima sempre più precisa delle risorse necessarie per un determinato *task*, una *skill* molto utile in ambito lavorativo soprattutto all'interno di un *team*.

L'inserimento efficace di questa tecnica nel mio *way of working* ha richiesto approssimativamente due settimane. Talvolta ho sostituito due intervalli di 25/5 minuti con un unico intervallo 50/10, in base al compito o alla disponibilità degli esperti di Sync Lab.

Tra gli altri miglioramenti dati dall'implementazione del Ciclo di Deming nel percorso di *stage* vi sono il miglioramento dell'ambiente *desktop* di lavoro per una codifica più efficiente, la capacità di esporre in modo chiaro e rapido un problema che richiede il supporto degli esperti aziendali, e la *skill* legata al tracciamento del lavoro svolto.

2.3.6 Pianificazione del lavoro

Ad ogni incremento è associato un requisito obbligatorio, desiderabile o facoltativo. A questi requisiti vi è associato un codice identificativo per favorirne il tracciamento futuro, che precede la voce descrittiva dell'incremento. Ogni codice è composto da una lettera seguita da dei numeri interi, secondo il seguente modello:

A-X.Y.Z

ove, da sinistra verso destra:

- **A** rappresenta la lettera che qualifica il requisito come obbligatorio, desiderabile o facoltativo, secondo la seguente notazione:
  - *O* per i requisiti obbligatori, vincolanti in quanto obiettivo primario richiesto dal committente;
  - *D* per i requisiti desiderabili, non vincolanti o strettamente necessari, ma dal riconoscibile valore aggiunto;
  - *F* per i requisiti facoltativi, rappresentanti valore aggiunto non strettamente competitivo.
- **X** rappresenta la settimana in cui viene inizialmente pianificato l'incremento (identificata da un numero incrementale e intero, partendo da 1). Questo consente allo studente, al tutor interno e al tutor esterno una rapida quantificazione dell'avanzamento corrente dello *stage* rispetto a quanto inizialmente pianificato.
- **Y** rappresenta la posizione sequenziale prevista dell'incremento all'interno della settimana (incrementale e intero, partendo da 1).

Di seguito viene presentata la pianificazione settimanale delle ore lavorative previste. Ad ogni settimana sono assegnate le voci contenenti gli incrementi previsti, ove i codici utilizzano la notazione descritta precedentemente.

Tutte le settimane prevedono 40 ore lavorative, fatta eccezione per l'ultima che ne prevede 20.

Settimana	Codice	Task associati
1	O-1.1	Incontro con le persone coinvolte nel progetto per discutere i requisiti e le richieste relative al sistema da sviluppare
	O-1.2	Verifica credenziali e strumenti di lavoro assegnati
	O-1.3	Presa visione dell'infrastruttura esistente
	D-1.1	Ripasso approfondito riguardo i seguenti argomenti: • Ingegneria del *software*; • Sistemi di versionamento; • Architetture *software*; • Cenni di *Networking*.
2	O-2.1	Nozioni fondamentali riguardo EAI_a e SOA_a[5]
	O-2.2	Approfondimenti riguardo le Architetture a Messaggio, in particolare: • *Integration Styles*; • *Channel Patterns*; • *Message Construction Patterns*; • *Routing Patterns*; • *Transformation Patterns*; • *System Management Patterns*.
3	O-3.1	Apache Kafka: • Introduzione a Kafka; • Concetti fondamentali di Kafka; • Avvio e CLI_a[6]; • Programmazione in Kafka con Java.
	D-3.1	Esempi e applicazioni di Apache Kafka
4	O-4.1	Confluent Platform: • *Service registry*; • $REST_a$ *proxy*; • kSQL; • Confluent *connectors*; • *Control center*.

[5] *Service Oriented Architecture*$_g$
[6] *Command Line Interface*

5	O-5.1	Analisi dei casi d'uso reali
	O-5.2	Realizzazione dei componenti per l'esecuzione dei casi di test
6	O-6.1	Analisi re-ingegnerizzazione e collaudo del flusso di integrazione asincrono
7	O-7.1	Analisi e re-ingegnerizzazione e collaudo del flusso di integrazione asincrono con callback
8	O-8.1	Analisi e re-ingegnerizzazione e collaudo del flusso di integrazione sincrono

Tabella 2.1: Pianificazione settimanale dello *stage*

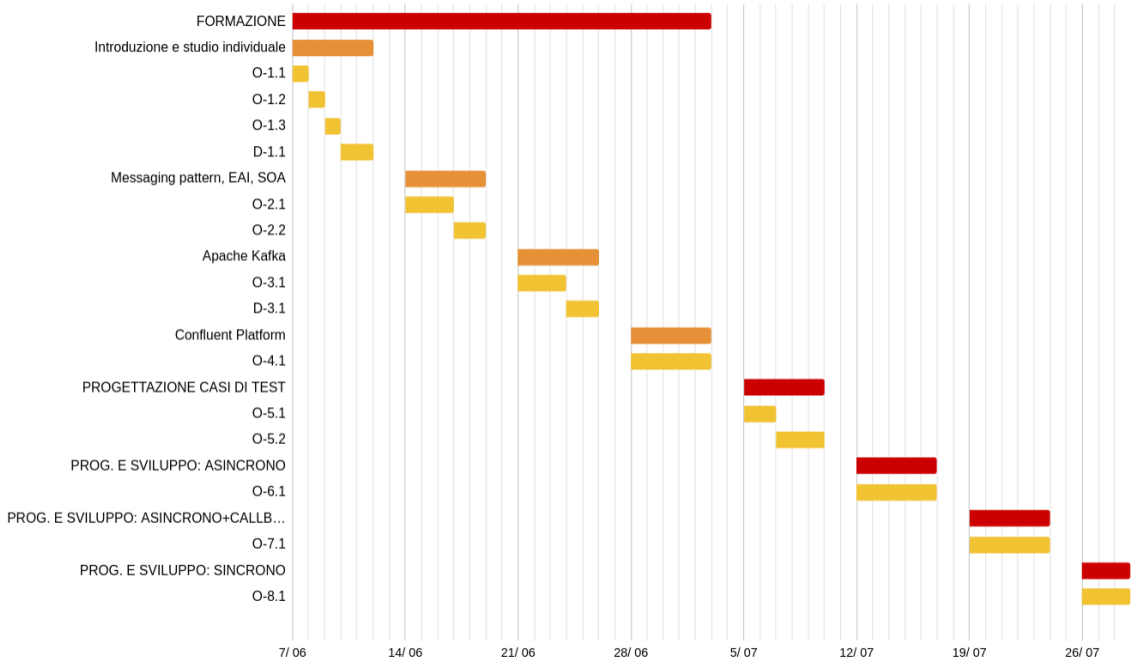

Figura 2.8: Diagramma di Gantt del piano di lavoro
Fonte: elaborazione personale

Secondo questa pianificazione, (di cui la figura 2.8 rappresenta il diagramma di Gantt) le 300 ore di *stage* previste sono approssimativamente divise in:

- 160 ore di Formazione sulle tecnologie;
- 60 ore di Progettazione dei componenti e dei test;
- 60 ore di Sviluppo dei componenti e dei test;
- 20 ore di Valutazioni finali, Collaudo e Presentazione della Demo.

Capitolo 3

Percorso di stage

3.1 Formazione

Il processo di Formazione ha avuto un importante ruolo all'interno dello *stage*, con una durata complessiva di circa quattro settimane. La causa di questo questa lunga durata è data dallo studio di diversi concetti per me nuovi, in particolare il settore del EAI_a e la tecnologia di Kafka.

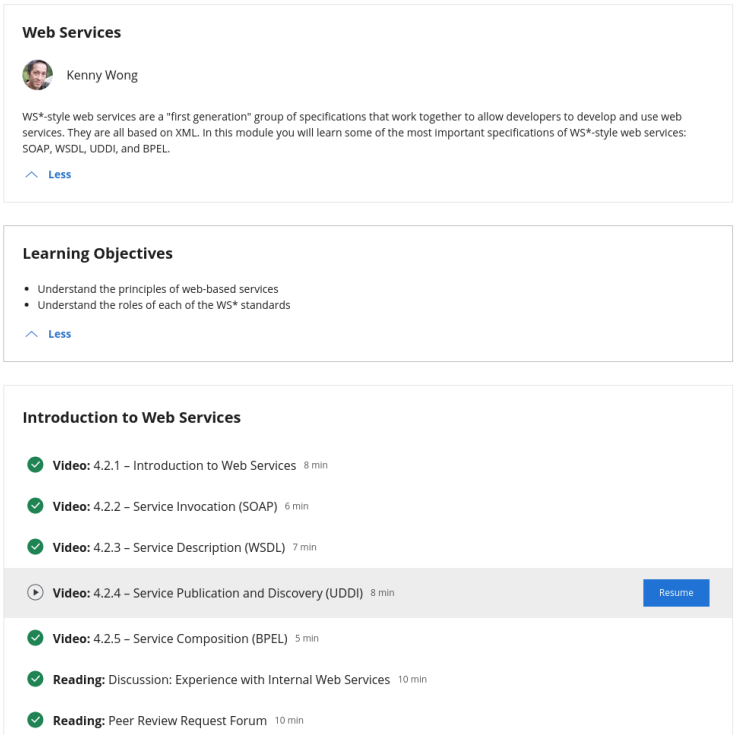

Figura 3.1: *Screenshot* del corso *online Service Oriented Architecture*$_g$ sulla piattaforma Coursera
Fonte: elaborazione personale

Durante questo processo Sync Lab mi ha fornito del materiale didattico per l'apprendimento, quali diapositive e appunti di origine aziendale e l'accesso ad alcuni corsi riguardo *Software architecture*, SOA_a (figura 3.1) (tramite i corsi online su Coursera) e Apache Kafka (tramite i corsi online su Udemy). Il tutor aziendale e responsabile EAI_a hanno fornito durante i *meeting* set-

timanali ulteriori chiarimenti e approfondimenti sul come Sync Lab applica questi concetti nello sviluppo di architetture *software*.

Per i corsi online a maggiore contenuto nozionistico ho redatto degli appunti riassuntivi, con lo scopo di consolidare l'apprendimento e velocizzare la verifica del tutor aziendale.

3.2 Analisi e modellazione di un caso d'uso

Ad alcune videoconferenze ha partecipato anche un esperto *senior* aziendale, esterno al progetto di *stage* in questione, per illustrarmi un caso d'uso in cui l'azienda ha prodotto un prototipo di sistema di integrazione. Ha utilizzato i concetti di *Web Service*, SOAP_a[1] e *request/response*, e mi ha permesso la visualizzazione dei *file* WSDL_a[2], XML_a[3] e XSD_a[4] associati. Ho pertanto generato un caso d'uso adatto agli scopi dello *stage* ispirandomi a quello reale illustrato.

Il caso d'uso modellato tratta una richiesta di credito telefonico da parte di un cliente ad un'azienda di telecomunicazioni tramite *Web Service*, per soddisfare il requisito dello sviluppo associato alla re-ingegnerizzazione del flusso di dati asincrono. La *request* avviene tramite flusso di un *file* JSON_a che viene trasmesso attraverso i vari servizi che compongono il sistema di integrazione, basato sul *Design Pattern* di tipo *publish/subscribe*.

Va precisato che il contenuto di tale JSON_a non è strettamente rilevante allo sviluppo e funzionamento del *Middleware*, ma aiuta a stabilire il contesto di utilizzo.
Il caso d'uso è composto dai seguenti passaggi:

1. il cliente (*WS_a*[5] *Client*) effettua una richiesta di credito tramite invio di un file JSON_a al successivo Servizio Web in ascolto.

2. il servizio composto da REST_a WS_a e *Request Producer* riceve il JSON_a e lo inserisce in Kafka tramite l'apposito *Producer*, assumendo la funzione di *publisher*.

3. il servizio di *Request Consumer*, sottoscritto al *topic* in questione riceve il JSON_a e lo invia al WS_a finale tramite una REST_a *request*.

4. il servizio in coda chiamato WS_a *Provider* riceve il JSON_a; grazie ai dati ricevuti è in grado di fornire il servizio richiesto dal *Client* nello *step* 1.

La modellazione dell'architettura e struttura del sistema da sviluppare seguirà questo prototipo di UC_a[6]. Il modello associato al caso asincrono con *callback* seguirà la stessa struttura e *step* dello UC_a illustrato qui sopra, con l'aggiunta speculare del messaggio di ritorno.

3.3 Progettazione architetturale

3.3.1 *Middleware* basato su un EDA_a

La progettazione architetturale ha portato alla produzione di diversi diagrammi UML_a per rappresentare efficacemente l'architettura del prodotto e fornire un modello da seguire durante la fase

[1] *Simple Object Access Protocol*
[2] *Web Service Description Language*
[3] *eXtensible Markup Language*
[4] *XML Schema Definition*
[5] *Web Service*
[6] *Use Case*

di Codifica. La fase ha richiesto frequenti *meeting* e confronti per raggiungere un risultato finale soddisfacente al fine della sperimentazione.

La progettazione architetturale del prodotto comprende un *Middleware* centrale basato su di un *Event Driven Architecture*$_g$ con l'utilizzo di Apache Kafka, e due componenti di *testing* chiamati *service client* e *service provider*, che simulano due servizi esterni che comunicano con il *Middleware* attraverso REST$_a$ *request*. Il sistema è composto da molteplici microservizi$_g$, che comunicano attraverso la rete di Docker.

L'obiettivo della progettazione è modellare un *Middleware* che sia implementabile in una *Service Oriented Architecture*$_g$, e consentirne la verifica e collaudo grazie ai servizi di *testing*.

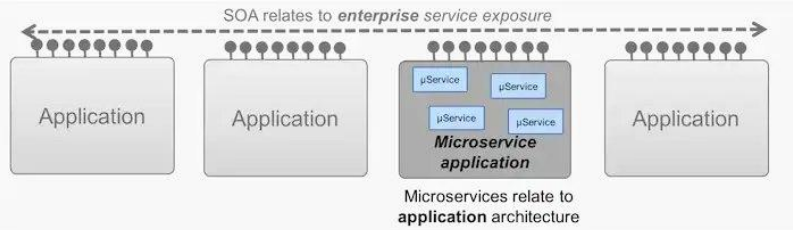

Figura 3.2: Visione ad alto livello delle differenze tra SOA$_a$ e microservizi$_g$
Fonte: https://www.ibm.com/cloud/blog/soa-vs-microservices

In figura 3.2 è rappresentata una visione ad alto livello dell'implementazione di un'applicazione a microservizi$_g$ all'interno di una SOA$_a$, evidenziando la differenza tra i due concetti.

Per rappresentare in modo chiaro ed elegante il modello descritto dal UC$_a$ sopra, ho elaborato diversi diagrammi UML$_a$. Questi diagrammi rappresentano i componenti *color coded*, notazione utilizzata per dare continuità e chiarezza attraverso le diverse tipologie di diagrammi UML$_a$.

3.3.2 UML$_a$ *sequence diagrams*

A partire dallo UC$_a$ descritto nella sezione precedente, ho prodotto un UML$_a$ *sequence diagram* più approfondito per rappresentare il flusso del JSON$_a$ tra i vari componenti.

La progettazione del sistema associato al caso asincrono con *callback* comprende lo stesso schema UML$_a$ del caso asincrono, con aggiunta del flusso di ritorno speculare al flusso di andata

La re-ingegnerizzazione del flusso sincrono è stata scartata in favore dello studio di funzionalità aggiuntive tramite l'utilizzo della piattaforma di *event streaming*. La progettazione di un sistema basato su questo flusso era inizialmente prevista come requisito desiderabile poiché associata ad un caso d'uso reale di cui si è parlato nelle sezioni precedenti. Successivamente è stata giudicata poco opportuna e pertanto fuori dagli scopi di Apache Kafka, un sistema basato sull'asincronismo.

Il tempo associato a tale requisito è stato quindi riproposto per testare un'altra funzione utile in un *Middleware*, quale la trasformazione di alcuni dati presenti nel JSON$_a$. Più precisamente, è stato aggiunto un dato sensibile che, in quanto tale, viene nascosto e sostituito con asterischi "*" dopo la produzione del *topic* in Kafka grazie all'utilizzo di Kafka Streams.

In figura 3.3 si può vedere il nuovo flusso asincrono con protezione (mascheramento) del dato sensibile.

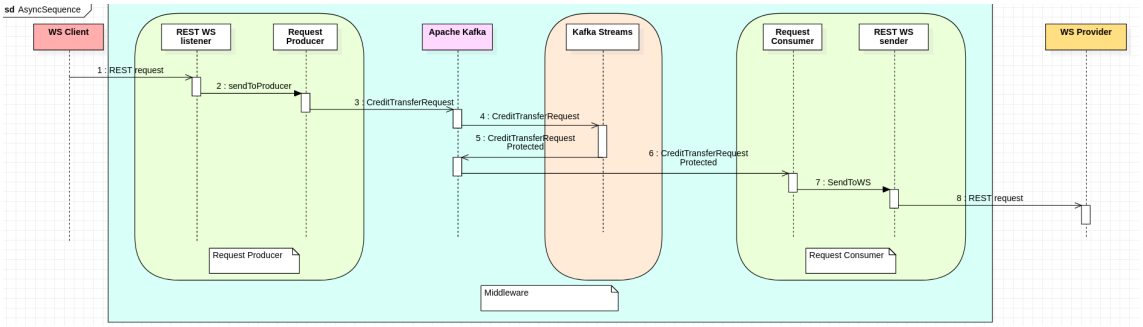

Figura 3.3: UML$_a$ *sequence diagram* per la re-ingegnerizzazione del flusso asincrono (protetto)
Fonte: elaborazione personale

3.3.3 UML$_a$ *deployment diagram*

A supporto di questi UML$_a$ *sequence diagram* che rappresentano efficacemente il flusso di dati, punto focale dell'intero sistema di integrazione, ho prodotto ulteriori diagrammi, tra i quali il *deployment diagram*.

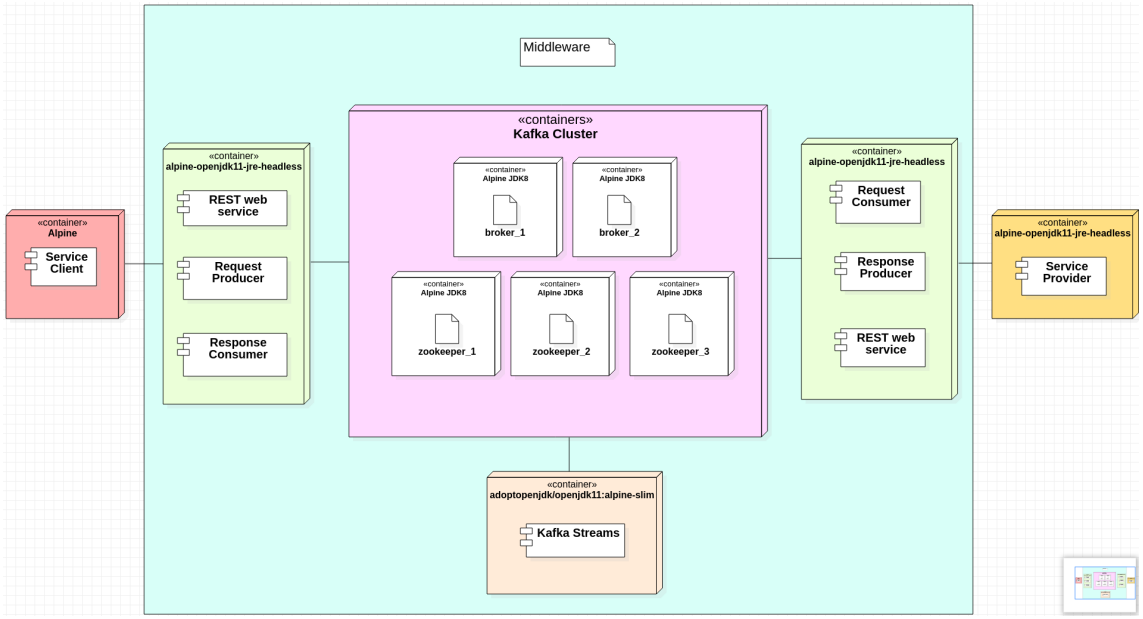

Figura 3.4: UML$_a$ *deployment diagram* per la re-ingegnerizzazione del flusso asincrono (protetto)
Fonte: elaborazione personale

Questo diagramma (figura 3.4) ha lo scopo di rappresentare la configurazione dei processi *run time*, modellando la struttura di base in cui eseguono i diversi servizi; il diagramma esprime l'ambiente in cui i vari componenti risiedono e la loro comunicazione.

Con l'approvazione degli esperti aziendali, ho deciso di appoggiare il sistema di integrazione sulla piattaforma Docker.

Il diagramma di *deployment* vede pertanto l'utilizzo di numerosi *container*$_g$ indipendenti che dialogano attraverso una rete locale all'interno di Docker. Questi *container*$_g$ sono raffigurati dai vari nodi rappresentati dai cubi in rilievo in figura. A questa notazione fa eccezione il nodo

virtuale intitolato "Kafka Cluster", che ha il solo scopo di raggruppare logicamente i vari nodi legati all'ambiente di Kafka con funzione comune, ma che in realtà non compone un *container*$_g$ reale. All'interno di questi nodi sono rappresentati gli artefatti che eseguono nel relativo *container*$_g$, per esplicitare la presenza dei componenti. Si può inoltre notare che l'ambiente di Apache Kafka è composto da un *cluster* composto da due servizi *broker* e tre servizi Zookeeper, allo scopo di simulare un caso d'uso reale in cui i diversi componenti sono distribuiti in sistemi indipendenti e garantiscono l'affidabilità dello *streaming* di eventi.

3.3.4 UML$_a$ *component diagram*

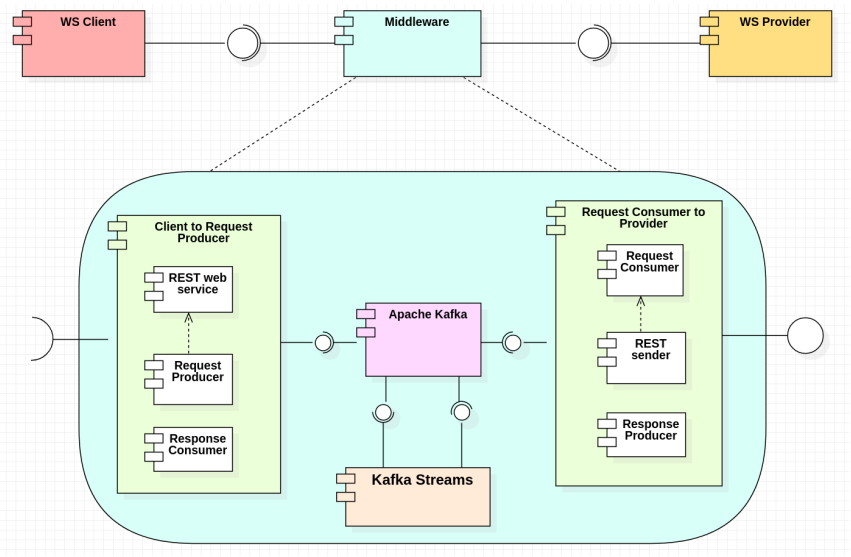

Figura 3.5: UML$_a$ *component diagram* per la re-ingegnerizzazione del flusso asincrono (protetto)
Fonte: elaborazione personale

Allo scopo di riassumere elegantemente i vari componenti del sistema ho elaborato un UML$_a$ *component diagram* (figura 3.5). Esso riassume con una visione ad alto livello i componenti che compongono il sistema e le modalità con cui esse interagiscono attraverso le relative interfacce.

3.4 Codifica

3.4.1 Kafka *cluster*

Il primo passo della fase di Codifica è stato quello del *setup* dell'ambiente di Kafka in locale. La guida rapida *online* fornita direttamente da Apache per un'installazione minimale di Kafka porta al *download* di un pacchetto e l'esecuzione di un servizio Zookeeper e un *broker* di Kafka.

Zookeeper è un servizio attualmente essenziale al funzionamento di Kafka, gestisce i nodi del *cluster* e mantiene una lista dei *topic* e dei messaggi. Le versioni future di Kafka renderanno questo servizio non necessario, ma attualmente sono ancora in fase di sviluppo e non adatte all'ambiente di produzione. I due *broker* si occupano di ricevere i messaggi dai *producer*.

Dopo un breve collaudo del corretto funzionamento dei servizi tramite l'inserimento di un evento in un *topic* e la relativa lettura, ho iniziato ad espandere e trasportare il sistema su dei *container*$_g$ Docker.

Il *cluster* di Kafka utilizzato nel progetto di *stage* è composto dai componenti descritti dalla progettazione architetturale nella sezione precedente, ovvero tre servizi di Zookeeper e due *broker* per rappresentare un sistema di piccole dimensioni che tuttavia si avvicina ad un caso d'uso reale poiché contiene multipli Zookeeper e multipli *broker*.

Questi cinque servizi necessitano dunque di eseguire contemporaneamente in un ambiente "containerizzato" con Docker connessi alla stessa *network* locale, che consente ai diversi servizi di scambiare messaggi.

Ho implementato questo *cluster* tramite un *file* `yml` utilizzato da docker-compose.
Il *file* contiene una lista di servizi, in cui in ognuno viene specificato:

- l'immagine Docker su cui viene costruito il servizio o il `dockerfile` da utilizzare per la sua costruzione;
- il nome del *container*$_g$;
- il nome del servizio;
- le dipendenze funzionali del servizio;
- l'indirizzo IPv4$_a$ statico e le porte attraverso cui è possibile raggiungere il servizio all'interno della rete locale dagli altri *container*$_g$.

3.4.2 *Request producer* e *request consumer*

Il passo successivo è stato quello di sviluppare dei semplici *producer* e *consumer* in grado di inserire e leggere i dati in Kafka. Una volta collaudati questi due eseguibili Java, li ho incapsulati all'interno di un'applicazione utilizzando lo strumento di *build* e gestione delle dipendenze Maven[7].

Per completare il modulo che costituisce il prototipo di *Middleware* è necessario che il *producer* e *consumer* siano in grado di comunicare con dei servizi esterni e fornire le interfacce adatte al ruolo. Nel caso più complesso del flusso asincrono con *callback*, entrambi questi servizi devono essere in grado di restare in ascolto di eventuali REST$_a$ *request* (che nel mio progetto utilizzano il protocollo HTTP$_a$) e al tempo stesso di inviarle.

Ho pertanto creato un HTTP$_a$ *server* minimale in Java attraverso il *framework* Netty[8], in esecuzione in un *thread* Java. Sviluppi futuri vedrebbero probabilmente l'utilizzo di un *framework* più evoluto come Spring Boot[9]. Un altro *thread* si occupa invece dell'invio delle REST$_a$ *request* al servizio di destinazione, una volta ricevuto un evento da parte del relativo *consumer*. Per abbreviare, chiamerò il servizio che si interpone tra il WS$_a$ *Client* e Kafka "*request producer*", e quello che si interpone tra Kafka e WS$_a$ *Provider* "*request consumer*", in base al loro scopo principale: essi inviano e ricevono la *request* relativa al dialogo dal *client* al *provider*. L'insieme di questi componenti formano i blocchi logici illustrati in verde nella sezione precedente, in cui il servizio *request producer* è formato da:

- REST$_a$ *Web Service* (HTTP$_a$ *server* in ascolto della *client request*);

[7] *Maven - Introduction*. URL: https://maven.apache.org/what-is-maven.html.
[8] *Netty: Home*. URL: https://netty.io.
[9] *Spring Boot*. URL: https://spring.io/projects/spring-boot.

- *client request producer*;
- *callback request consumer*;
- REST$_a$ *Web Service* (invio della *callback request*);

e il *request consumer* da:

- *client request consumer*;
- REST$_a$ *Web Service* (invio della *client request*);
- REST$_a$ *Web Service* (HTTP$_a$ *server* in ascolto della *callback request*);
- *callback request producer*.

3.4.3 WS$_a$ *Client* e WS$_a$ *Provider*

Al fine di testare il prototipo di *Middleware* prodotto, ho realizzato due ulteriori *container*$_g$ con all'interno due eseguibili che si occupano di interagire con esso.

Il servizio intitolato WS$_a$ *Provider* è fondamentalmente simile ai servizi descritti nella sottosezione precedente: anch'esso possiede un HTTP$_a$ *server* per rimanere in ascolto delle REST$_a$ *request* inviate dal *request consumer*, in aggiunta ad un metodo per elaborare la REST$_a$ *request* di risposta associata al *callback*.

L'altro servizio, che idealmente potrebbe essere molto simile al precedente servizio di test, in realtà presenta delle considerevoli differenze. La causa di ciò non è una differenza reale in termini di funzionalità, quanto una questione di rapidità di *testing* e codifica.

Il servizio infatti è molto più leggero in termini di memoria, poiché composto da un semplice *container*$_g$ con una distribuzione di Alpine Linux[10]; esso possiede un'installazione del *software* `curl` (accessibile via CLI$_a$), ma è privo di ulteriori *software* aggiuntivi da me prodotti.

Gli obiettivi di questo WS$_a$ *Client* sono legati a due *network utility*: `curl` e `netcat`. La prima, interagendo manualmente con il *container*$_g$ tramite l'interfaccia CLI$_a$, mi permette di eseguire REST$_a$ *request* con il JSON$_a$ di partenza. La seconda mi consente di restare in ascolto di eventuali *request* su di una porta a mia scelta.

3.4.4 Protezione dei dati sensibili con Kafka Streams

Come visto nella sezione 3.3, alcuni dei dati trasmessi nel JSON$_a$ vengono trasformati per mascherare i dati sensibili. Precisamente, il JSON$_a$ passa dall'avere la seguente forma (figura 3.6)

```
{
  "CallerSystem": "Sistema chiamante 1",
  "PhoneNumber": "012345679",
  "Currency": "EUR",
  "Amount": "5",
  "Info": "Causale del Trasferimento Credito Residuo",
  "DebitType":"Bancomat",
  "CreditTransferDate": "2021-07-21",
  "CreditCardNumber": "1234567890123456"
}
```

Figura 3.6: JSON$_a$ inviato al *Middleware*
Fonte: elaborazione personale

[10] *index | Alpine Linux*. URL: https://www.alpinelinux.org.

alla seguente forma rielaborata (figura 3.7), in cui l'ultimo dato ha subito la modifica descritta.

```
{
  "CallerSystem": "Sistema chiamante 1",
  "PhoneNumber": "012345679",
  "Currency": "EUR",
  "Amount": "5",
  "Info": "Causale del Trasferimento Credito Residuo",
  "DebitType":"Bancomat",
  "CreditTransferDate": "2021-07-21",
  "CreditCardNumber": "****************"
}
```

Figura 3.7: JSON_a protetto, ricevuto al termine del *callback*
Fonte: elaborazione personale

Ho implementato questa funzione utilizzando Kafka Streams[11], che permette di leggere un *topic* e modificarlo istantaneamente per un'elaborazione in *real time*.

3.4.5 Efficienza nello sviluppo

Durante la codifica, ho adottato diverse misure per minimizzare il tempo necessario. Questi provvedimenti riguardano principalmente l'ottimizzazione del sistema di $container_g$, non allo scopo di migliorare l'efficienza, rapidità d'esecuzione e di *build* del prodotto finale (qualità comunque raggiunte), ma a quello di ridurre il tempo di *testing*, riducendo di conseguenza le ore necessarie allo sviluppo.

Tutte le misure prese sono strettamente legate alla mia familiarità con alcune tecnologie e alle risorse personali richieste per l'apprendimento e sviluppo di una nuova funzionalità.
Riporto i principali provvedimenti intrapresi per minimizzare il tempo di sviluppo:

- **ottimizzazione delle risorse utilizzate dai *container*$_g$**. Molti dei $container_g$ prodotti possiedono delle versioni *premade* sulla libreria di DockerHub[12] (ad esempio i $container_g$ che formano il *cluster* di Kafka). Tuttavia, utilizzare delle versioni costruite *ad-hoc*, mi ha permesso di ridurre considerevolmente le dimensioni del $container_g$ e mantenere solamente le funzioni a me necessarie, e conseguentemente ridurre il tempo della *build* delle immagini e la loro esecuzione. Date le numerose operazioni di *build* e *testing*, nel lungo termine questa operazione ha portato ad un risparmio di tempo significativo.

- **invio della richiesta REST_a del *service client* tramite CLI_a**. I vantaggi di questa modalità manuale risiedono come anticipato nella rapidità di sviluppo, non solo di questo servizio ma anche dei restanti. Infatti, grazie alle funzioni offerte da docker-compose, è semplice eseguire il *restart* di un singolo $container_g$ contenente un servizio per testarne la nuova versione, e successivamente è sufficiente eseguire manualmente la *request* dal WS_a *Client* per verificare il funzionamento del sistema. È sicuramente possibile l'automatizzazione di questo processo (ad esempio effettuando richieste continue in modo automatico) ma l'implementazione di questa funzione avrebbe, secondo la mia stima personale, richiesto più tempo rispetto alla soluzione attuata o posto problemi nel filtro dell'*output*.

[11] *Kafka Streams: Introduction.* URL: https://kafka.apache.org/28/documentation/streams/.
[12] *Docker Hub Container Image Library | App Containerization.* URL: https://hub.docker.com.

- **la *build* delle applicazioni costruite con Maven avviene in locale**. Questo porta a due vantaggi. Il primo è che la *build* impiega meno tempo, poiché non è necessario lanciare alcun *container*$_g$ e Maven non deve sincronizzare o controllare la versione delle dipendenze *online* più di volte (è possibile disattivare la sincronizzazione, ma richiede comunque più tempo). La seconda, più importante, è che l'immagine Docker stessa non richiede né *build* né Maven. Il *container*$_g$ si occupa solamente di copiare l'eseguibile pre-costruito al suo interno e di eseguirlo con Java.

3.5 Prodotto finale, verifica e collaudo

3.5.1 Prodotto finale

Il prodotto finale del progetto è composto dai diversi componenti illustrati nella sotto-sezione 3.3.4, per un totale di dieci servizi, ognuno nel proprio *container*$_g$ Docker (conformi con il *deployment diagram* alla sotto-sezione 3.3.3). Una visione ad alto livello può esprimere il prodotto risultante dallo *stage* come **due servizi di test quali WS$_a$ *Client* e WS$_a$ *Provider* che scambiano messaggi sotto forma di `file` JSON$_a$, tramite un *Middleware* basato su Apache Kafka in un'architettura focalizzata sugli eventi.** Il *Middleware* presenta inoltre la funzionalità aggiuntiva di trasformazione dei dati grazie all'utilizzo di Kafka Streams.

Figura 3.8: *Folder tree* dell'applicativo relativo al servizio *request producer*

Fonte: elaborazione personale

Tra i diversi servizi prodotti, tre risultano più strutturati e complessi, ovvero il *request producer* (figura 3.8), *request consumer* e il WS$_a$ *Provider*. Questi infatti sono degli applicativi *multi-thread*, con struttura generata da Maven e formati da diversi `file` Java che si occupano di inviare e ricevere REST$_a$ *request*, "produrre" e "consumare" i dati utilizzando Kafka.

I cinque servizi del *cluster* di Apache Kafka risultano relativamente semplici: sono composti da un'immagine di Alpine Linux con un'installazione di Java JRE$_a$[13] versione 8, con all'interno gli eseguibili di Kafka e i file di configurazione personalizzati per il *setup* del *cluster* (porte e indirizzi).

Il servizio che compone il WS$_a$ *Client* è quello di dimensioni più ridotte, essendo composto da un'immagine minimale di Linux con all'interno le *network utility* trattate in precedenza.

Infine, il servizio che utilizza Kafka Streams per il mascheramento dei dati sensibili è simile ai primi tre, con la differenza che esso non richiede dei processi *multi-thread* ma solamente un'unica classe Java per il processo di *streaming* dei messaggi.

Nella figura 3.9 è presente una bozza riassuntiva dei componenti e della loro comunicazione, presa dalla *board* di progetto Click Up.

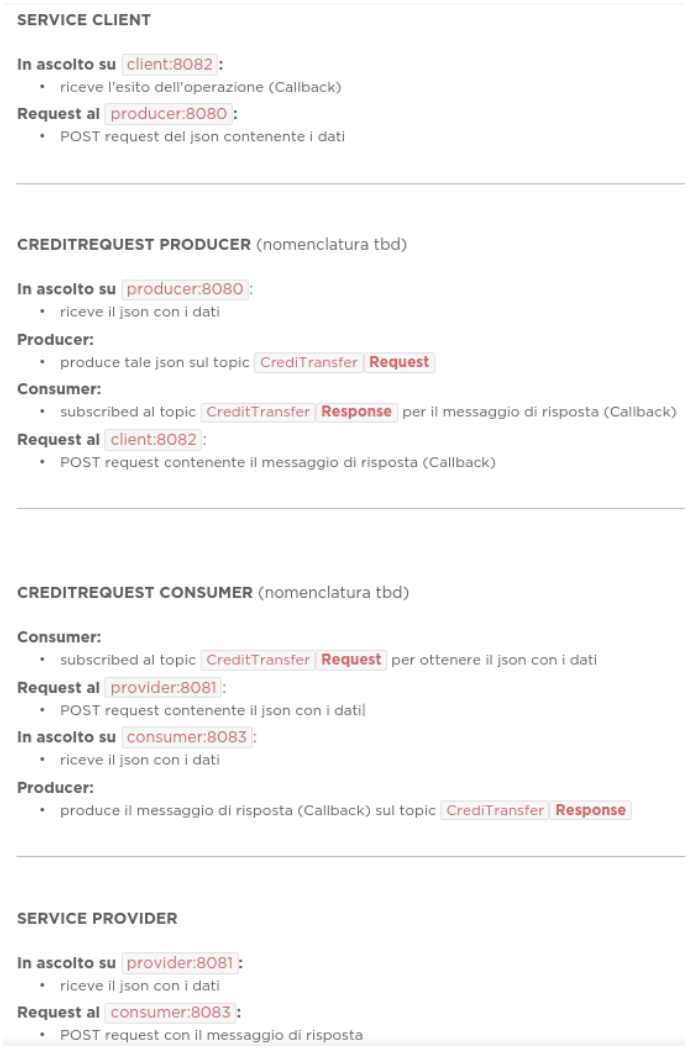

Figura 3.9: UML$_a$ Riassunto dei componenti prodotti nel caso asincrono con *callback*
Fonte: elaborazione personale

L'immagine illustra i servizi prodotti per il flusso asincrono con *callback* e di come essi comunichino tra di loro attraverso le varie porte, elaborata durante la Progettazione iniziale (si può

[13] *Java Runtime Environment*

notare come non sia ancora presente la protezione del dato sensibile con Kafka Streams, funzionalità aggiunta negli ultimi giorni di *stage*).

3.5.2 Verifica

Il tempo dedicato alla fase di Verifica è stato relativamente breve rispetto alla maggior parte dei prodotti *software*: la causa di questa durata minore è la natura sperimentale del progetto. Come visto nella sotto-sezione 2.3.1, lo *stage* è incentrato sulla re-ingegnerizzazione del flusso di dati senza l'obiettivo di produrre un *Middleware* da implementare in produzione; quest'ultimo è un obiettivo aziendale nel lungo termine, ma non di questo specifico progetto di sperimentazione. La fase di Verifica è stata applicata successivamente alla Codifica, con particolare attenzione nelle fasi finali.

Il prodotto è stato più volte verificato con la supervisione del tutor aziendale e il responsabile del EAI_a. Tutti i $container_g$ prodotti sono stati revisionati, compresa la loro corretta esecuzione. I passaggi principali della Verifica hanno visto la verifica manuale dell'*input* e *output* di ogni singolo $container_g$ sia nel flusso di dati in andata che nel flusso di ritorno di *callback*, con eccezione del *cluster* Kafka. La stampa su *console* del movimento del dato, contenuta nei servizi scritti in Java, ha aiutato la verifica costante della funzionalità principale, permettendo il rilevamento immediato di casi di regressione. Il *cluster* Kafka invece è stato il soggetto di una verifica differente: ho analizzato approfonditamente l'*output* in *console* dei vari $container_g$, per assicurarmi la corretta comunicazione tra di essi. Tramite l'utilizzo degli eseguibili di supporto messi a disposizione dal *team* di Apache Kafka, ho periodicamente controllato i contenuti dei *topic* generati durante il progetto ed il corretto funzionamento del *cluster*.

3.5.3 Collaudo

Come per la fase di Verifica, anche il Collaudo ha visto un utilizzo di risorse ridotto rispetto a quelle richieste per un *software* da distribuire in produzione. Il tutor e gli esperti aziendali hanno contribuito a questa fase.

Il collaudo ha riguardato il processo di *build* e successivo avvio simultaneo di ogni singolo servizio all'interno del *cluster*, tramite l'apposito comando di docker-compose "`docker-compose up -build`".

Dopodiché, al fine di testare che il *software* eseguisse effettivamente le funzioni richieste dal progetto di sperimentazione (*test* d'efficacia), ho effettuato delle HTTP_a REST_a *request* contenenti il JSON_a con i dati. Ho eseguito questo collaudo tramite una connessione manuale al $container_g$ contenente il servizio WS_a *client*, per poi dare manualmente il comando di *request* grazie all'*utility* `curl`. L'analisi del JSON_a in entrata e in uscita dal sistema hanno garantito l'efficacia del *software* sperimentale prodotto: la richiesta contenente i dati necessari viene correttamente trasmessa dal cliente al fornitore di tale servizio, e infine il cliente riceve un messaggio di risposta.

Per dimostrare i risultati raggiunti e le capacità del prodotto finale, esso è stato presentato all'azienda in conclusione del percorso di *stage* in un *online meeting*. Il *meeting* si è tenuto in due fasi distinte: la prima costituita da un'introduzione concettuale dell'architettura e degli scopi del percorso tramite l'aiuto di alcune diapositive da me prodotte, e la seconda da una dimostrazione *live* del *software* in esecuzione tramite la condivisione dello schermo.

Capitolo 4

Valutazione retrospettiva

4.1 Obiettivi aziendali raggiunti

L'azienda ha potuto testare, con questo percorso di *stage*, le potenzialità di Apache Kafka nell'ambito dei *Middleware* e dei sistemi di integrazione. La piattaforma di *event streaming* risulta molto promettente da integrare negli attuali e futuri sistemi di integrazione basati su di un *Event Driven Architecture*$_g$, al fine di soddisfare i bisogni del cliente della gestione di un flusso di dati sempre maggiore, in tempo reale. Il *software* può portare dunque a una grande innovazione nel settore EAI$_a$ e permettere a Sync Lab di fornire prodotti all'avanguardia ai suoi clienti.

Inoltre, durante questi mesi di *stage*, l'azienda ha avuto modo di conoscere il mio metodo di lavoro e le mie capacità.

Il tutor aziendale e gli esperti del settore mi hanno fornito un riscontro molto positivo, sia riguardo i risultati raggiunti dallo *stage* che riguardo le mie capacità e maturazione professionale osservate durante il percorso.

4.2 Obiettivi dello *stage* raggiunti

Ho raggiunto gli obiettivi principali dello stage con successo.

I microservizi$_g$ che compongono il prodotto finale hanno raggiunto efficacemente il risultato inizialmente previsto, creando il sistema richiesto dalla sperimentazione; i due servizi di test quali WS$_a$ *Client* e WS$_a$ *Provider* si scambiano messaggi tramite un *Middleware* basato su Apache Kafka.

La sperimentazione ha testato alcune delle capacità di Apache Kafka con esito positivo, fornendo le basi per ulteriori percorsi di approfondimento che possono portare all'implementazione della piattaforma di *event streaming* all'interno degli attuali sistemi di integrazione con il ruolo di *Middleware*.

Un possibile percorso potrebbe ad esempio modellare e sviluppare un caso d'uso molto più complesso, con simulazione di un flusso di dati continuo e di grandi dimensioni, con dati provenienti da fonti multiple, un numero maggiore di *producer* e *consumer*, e la sperimentazione di ulteriori funzionalità presenti nei *Middleware* attualmente utilizzati.
Di seguito riprendo parte della tabella 2.1 vista nella sezione 2.3.6.

Obiettivo	Task associati	Raggiunto
O-5.1	Analisi dei casi d'uso reali	si
O-5.2	Realizzazione dei componenti per l'esecuzione dei casi di test	si
O-6.1	Analisi re-ingegnerizzazione e collaudo del flusso di integrazione asincrono	si
O-7.1	Analisi e re-ingegnerizzazione e collaudo del flusso di integrazione asincrono con callback	si
O-8.1	Analisi e re-ingegnerizzazione e collaudo del flusso di integrazione sincrono	no

Tabella 4.1: Obiettivi dello *stage* raggiunti

Come detto in precedenza, ho scartato l'obiettivo O-8.1 in favore della sperimentazione di alcune funzioni aggiuntive di Kafka, inizialmente non pianificate.

4.3 Contenuti formativi acquisiti

Il processo di Formazione ha riguardato principalmente i concetti inerenti al settore del *Enterprise Application Integration*$_g$ e le tecnologie legate ad Apache Kafka.

Di seguito espongo i requisiti formativi soddisfatti, in relazione al piano di lavoro iniziale.

Obiettivo	Task associati	Raggiunto
O-2.1	Nozioni fondamentali riguardo EAI$_a$ e SOA$_a$	si
O-2.2	Approfondimenti riguardo le Architetture a Messaggio	si
O-3.1	Apache Kafka	si
D-3.1	Esempi e applicazioni di Apache Kafka	si
O-4.1	Confluent Platform	si

Tabella 4.2: Contenuti formativi acquisiti

4.4 Obiettivi personali raggiunti

Valuto i risultati personali raggiunti dal percorso di *stage* molto soddisfacenti, soprattutto dal punto di vista di una maturazione professionale.

La formazione ricevuta nel settore del *Enterprise Application Integration*$_g$ ha allargato le mie conoscenze tecnologiche nell'ambito dell'ingegneria del *software*; in particolare ho apprezzato l'approfondimento riguardo le architecture *software* moderne e i sistemi di integrazione associati al mondo del *Big Data*. Ho apprezzato molto lo studio delle tecnologie emergenti per un'innovazione aziendale nel EAI$_a$, quali Apache Kafka e Confluent, e le conseguenze importanti dovute alla migrazione di un sistema verso una *Event Driven Architecture*$_g$.

L'alto livello di organizzazione personale che ho tenuto durante il percorso ha garantito un buona qualità nel *way of working*, preparandomi all'inserimento nel mondo del lavoro e a un contesto aziendale innovativo e all'avanguardia.

Ho appreso molto dagli esperti aziendali, che mi hanno fornito il supporto e le linee guida necessarie al compimento dello *stage*; in particolare, ho imparato gli importanti passi e considerazioni necessarie che guidano la sperimentazione associata all'implementazione di tecnologie innovative, sia in ambito tecnico che architetturale.

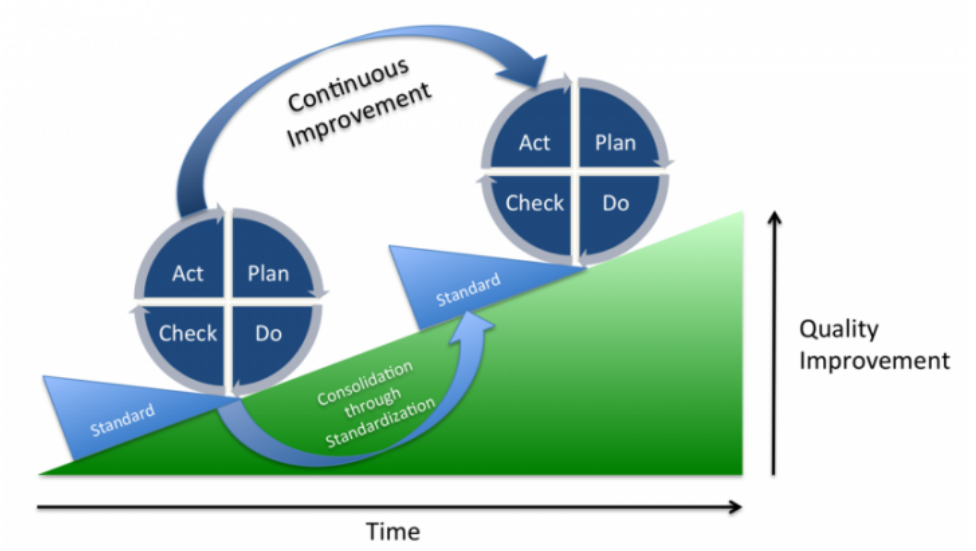

Figura 4.1: *Plan Do Check Act*

Fonte: http://quickstart-indonesia.com/siklus-pdca/

Il metodo PDCA$_a$[1] (figura 4.1) mi ha portato a miglioramenti personali in diversi ambiti, soprattutto quelli legati alla gestione del tempo (come visto nella sotto-sezione 2.3.5), efficienza organizzativa ed efficacia di sviluppo.

Ritengo dunque di aver raggiunto tutti gli obiettivi personali posti, con risultati addirittura migliori delle mie aspettative iniziali.

4.5 Distanza rispetto ai contenuti del corso di studi

Nonostante abbia trovato nuove tecnologie e concetti nello *stage*, non previsti negli insegnamenti del Corso di Laurea, **ritengo che il percorso di studi abbia completamente soddisfatto i requisiti necessari per il completamento dello *stage*.**

[1] *Plan Do Check Act*

Infatti, sebbene gli insegnamenti non abbiano specificatamente trattato le moderne tecnologie utilizzate, mi hanno fornito la preparazione necessaria al loro rapido apprendimento; in un settore lavorativo in rapida evoluzione come quello dell'informatica, la capacità di adattamento all'utilizzo di strumenti moderni come Apache Kafka e quella di comprendere facilmente le architetture innovative è di fondamentale importanza e personalmente molto apprezzata.

Mi ritengo pertanto pienamente soddisfatto del percorso di studi e soprattutto del percorso di *stage*; ritengo essi abbiano dato una rapida accelerazione alla mia formazione professionale e abbia fornito i presupposti necessari per un mio inserimento nell'ambiente lavorativo.

Acronimi

API Application Programming Interface. 17, 20

CLI Command Line Interface. 32, 41, 42

COBRA Common Object Request Broker Architecture. 10

EAI *Enterprise Application Integration*$_g$. 3, 7, 11, 13, 14, 16–18, 21, 23, 26–28, 32, 35, 45, 47–49

EDA *Event Driven Architecture*$_g$. 4, 7, 21, 24, 27, 36, 37, 47, 49

EJB Enterprise Java Bean. 10

ESB Enterprise Service Bus. 3, 7, 14–16

HTTP HyperText Transfer Protocol. 14, 40, 41, 45

ICT Information and Communication Technologies. 9, 26

IPv4 Internet Protocol version 4. 40

J2EE Java 2 Platform Enterprise Edition. 10

JMS Java Message Service. 10

JRE *Java Runtime Environment*. 44

JSON JavaScript Object Notation. 7, 14, 36, 37, 41–43, 45

MOM Message Oriented Middleware. 10, 15

O-O Object Oriented. 10

OS Operating System. 17

P2P Point To Point. 7, 16, 20, 23, 24

PDCA *Plan Do Check Act*. 7, 29, 49

REST REpresentational State Transfer. 20, 32, 36, 37, 40–43, 45

SOA *Service Oriented Architecture*$_g$. 3, 7, 14, 15, 32, 35, 37, 48

SOAP Simple Object Access Protocol. 14, 36

SSO Single Sign On. 17

TCP Transmission Control Protocol. 20

UC Use Case. 36, 37

UML Unified Modeling Language. 4, 7, 10, 36–39, 44

VM Virtual Machine. 7, 18–20

WS Web Service. 4, 36, 40–45, 47

WSDL Web Service Description Language. 36

XML eXtensible Markup Language. 36

XSD XML Schema Definition. 36

Glossario

Enterprise Application Integration

Il termine si riferisce al processo d'integrazione tra diversi tipi di sistemi informatici di un'azienda attraverso l'utilizzo di software e soluzioni architetturali.

Fonte: https://it.wikipedia.org/wiki/Enterprise_Application_Integration . 16, 23, 27, 48, 49, 51

Event Driven Architecture

Una Event Driven Architecture è costituita da produttori di eventi che generano un flusso di eventi e consumer eventi che sono in ascolto degli eventi.

Gli eventi vengono recapitati praticamente in tempo reale, in modo che i consumer possano rispondervi immediatamente non appena si verificano. I producer sono separati dai consumer: un producer è all'oscuro dei consumer in ascolto. Anche i consumer sono separati tra loro e ognuno visualizza tutti gli eventi. Questo comportamento differisce da un modello con consumer concorrenti, in cui i consumer eseguono il pull di messaggi da una coda e un messaggio viene elaborato solo una volta (presupponendo l'assenza di errori). In alcuni sistemi, ad esempio nei sistemi IoT, gli eventi devono essere inseriti a volumi molto elevati.

Un'architettura guidata dagli eventi può usare un modello di pubblicazione/sottoscrizione o un modello di flusso di eventi.

Fonte: https://docs.microsoft.com/it-it/azure/architecture/guide/architecture-styles/event-driven . 21, 24, 37, 47, 49, 51

Service Oriented Architecture

La Service Oriented Architecture definisce un modo per rendere i componenti software riutilizzabili tramite interfacce di servizio. Queste interfacce utilizzano standard di comunicazione comuni in modo da poter essere rapidamente integrate in nuove applicazioni senza dover eseguire ogni volta una profonda integrazione.

Ogni servizio in una SOA incorpora il codice e le integrazioni dei dati necessari per eseguire una funzione aziendale completa e discreta (ad esempio, il controllo del credito del cliente, il calcolo di un pagamento di un prestito mensile o l'elaborazione di un'applicazione ipotecaria). Le interfacce di servizio forniscono un accoppiamento libero, il che significa che possono essere richiamate con poca o nessuna conoscenza della sottostante modalità di implementazione dell'integrazione.

I servizi sono esposti utilizzando protocolli di rete standard - come SOAP (simple object access protocol)/HTTP o JSON/HTTP - per inviare richieste di lettura o modifica dei dati. I servizi sono pubblicati per consentire agli sviluppatori di trovarli rapidamente e riutilizzarli per assemblare nuove applicazioni.

Fonte: https://www.ibm.com/it-it/cloud/learn/soa . 3, 7, 14, 15, 32, 35, 37, 51

container

I container sono delle unità eseguibili di software in cui viene impacchettato il codice applicativo, insieme alle sue librerie e dipendenze, con modalità comuni in modo da poter essere eseguito ovunque, sia su desktop che su IT tradizionale o cloud.

Per farlo, i container usufruiscono di una forma di virtualizzazione del sistema operativo (SO), in cui le funzioni del SO (ossia, nel caso del kernel Linux, i namespace e le primitive dei cgroup) vengono utilizzate efficacemente sia per isolare i processi che per controllare la quantità di CPU, memoria e disco a cui tali processi hanno accesso.

I container sono piccoli, veloci e portatili perché, diversamente da una VM, non hanno bisogno di includere un sistema operativo guest in ogni istanza e possono invece sfruttare semplicemente le funzioni e le risorse del sistema operativo host.

I container sono apparsi per la prima volta decenni fa con versioni come le jail FreeBSD e le partizioni di carico di lavoro AIX (WPAR), ma la maggior parte degli sviluppatori moderni ricorda il 2013 come inizio dell'era dei container moderni con l'introduzione di Docker.

Fonte: https://www.ibm.com/it-it/cloud/learn/containers . 13, 17–20, 38–43, 45

microservizi

I microservizi sono un approccio per sviluppare e organizzare l'architettura dei software secondo cui quest'ultimi sono composti di servizi indipendenti di piccole dimensioni che comunicano tra loro tramite API ben definite. Questi servizi sono controllati da piccoli team autonomi.

Le architetture dei microservizi permettono di scalare e sviluppare le applicazioni in modo più rapido e semplice, permettendo di promuovere l'innovazione e accelerare il time-to-market di nuove funzionalità.

Fonte: https://aws.amazon.com/it/microservices/ . 7, 15, 17–20, 25, 37, 47

Bibliografia

[1] *ClickUp™ | One app to replace them all.* URL: https://clickup.com.

[2] *Cos'è il Middleware?* URL: https://www.redhat.com/it/topics/middleware/what-is-middleware.

[3] *Coursera | Build Skills with Online Courses.* URL: https://www.coursera.org.

[4] *Docker Hub Container Image Library | App Containerization.* URL: https://hub.docker.com.

[5] *Empowering App Development for Developers | Docker.* URL: https://www.docker.com/.

[6] *Google Chat.* URL: https://chat.google.com.

[7] *index | Alpine Linux.* URL: https://www.alpinelinux.org.

[8] *Kafka Streams: Introduction.* URL: https://kafka.apache.org/28/documentation/streams/.

[9] *Maven - Introduction.* URL: https://maven.apache.org/what-is-maven.html.

[10] *Netty: Home.* URL: https://netty.io.

[11] *Notion - The all-in-one workspace for your notes, tasks, wikis, and databases.* URL: https://www.notion.os.

[12] *Online Courses - Learn Anything, On Your Schedule | Udemy.* URL: https://www.udemy.com.

[13] *Oracle VM Virtual Box.* URL: https://www.virtualbox.org/.

[14] *Overview of Docker Compose | Docker Documentation.* URL: https://docs.docker.com/compose/.

[15] *Spring Boot.* URL: https://spring.io/projects/spring-boot.

[16] *Sync Lab.* URL: https://www.synclab.it/.