

UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA

Corso di Laurea Triennale in Ingegneria Informatica e Automazione

Stream management nell'Industria 5.0: progettazione e sviluppo di un web editor per la modellazione di workflow

Stream Management in Industry 5.0: Design and Development of a Web Editor for Workflow Modeling

Relatore:

Prof. Emanuele Storti

Tesi di laurea di:

Edoardo Tarulli

A.A 2025/2026

«La vita è un processo di flusso di informazioni,
un'organizzazione continua contro il disordine.»

— Norbert Wiener

Sommario

Elenco delle figure **7**

Introduzione **8**

1 Contesto applicativo **11**

1.1 Gestione dei Data Stream e Stream Processing 12

1.2 Ingestione ed inoltro dei dati 12

1.3 Elaborazione in tempo reale 14

1.4 Architettura del framework 15

1.5 Stream Postprocessing e configurazione dei flussi 17

2 Requisiti di progetto e casi d'uso **18**

2.1 Obiettivi specifici 19

2.2 Analisi dei requisiti 20

2.2.1 Requisiti funzionali 21

2.2.2 Requisiti non funzionali 24

2.3 Casi d'uso 26

3 Tecnologie utilizzate **30**

3.1 Visual Studio Code 31

3.2 React 33

3.2.1 React Flow 34

3.3 HTML 36

3.4 JavaScript 37

3.5 CSS 38

3.6 Github 39

4 Progettazione **40**
4.1 Modellazione degli operatori Flink 41
4.1.1 Configurazione delle proprietà 43
4.1.2 Struttura dati 46
4.2 Esempi applicativi di modellazione dei workflow 49
4.2.1 Pipeline di monitoraggio e filtraggio IoT con finestratura 49
4.2.2 Unione di flussi paralleli 50
4.3 Progettazione dell'architettura software 51
4.4 Gestione della logica di workflow e utilità 53
4.5 Mock-up progettuali e UI 55

5 Implementazione **58**
5.1 Schema architetturale e dipendenze dei componenti 59
5.2 Struttura del workspace e organizzazione dei file 61
5.3 Approccio data-driven e configurazione dei nodi 63
5.4 Gestione dello stato e del flusso di lavoro 65
5.5 Validazione topologica 66
5.6 Funzionalità di importazione ed esportazione del workflow . . 68
5.7 Problemi riscontrati e soluzioni adottate 71
5.7.1 Persistenza dello stato al cambio pagina 71
5.7.2 Gestione dinamica degli identificativi e prevenzione dei conflitti 72
5.7.3 Ottimizzazione delle prestazioni 73
5.8 Risultati finali 74
5.8.1 Gestione in tempo reale degli errori topologici 75
5.8.2 Procedura di esportazione e notifiche di sistema 76

6 Conclusioni e sviluppi futuri **77**
6.1 Bilancio dei risultati raggiunti 78
6.2 Sviluppi futuri 79

Riferimenti bibliografici **81**

Elenco delle figure

1 Grafo delle operazioni 9
2 Architettura Apache Kafka 13
3 Gestione flussi Apache Flink 14
4 Architettura a microservizi 15
5 Visual Studio Code 31
6 React Flow UI 34
7 Modalità di esportazione JSON 48
8 Workflow di monitoraggio IoT e filtraggio 49
9 Workflow di unione flussi paralleli 50
10 Mock-up pagina Home 55
11 Mock-up Canvas 56
12 Mock-up Workflow 57
13 Schema architetturale del software 59
14 Albero di lavoro 61
15 Schema delle proprietà 63
16 Funzione di estrapolazione dati 64
17 Gestione nodo Map 65
18 Gestione nodo Aggregate 65
19 Funzione di validità dei collegamenti 66
20 Funzione di calcolo degli errori 67
21 Funzione esportazione 69
22 Funzione importazione 70
23 Funzione di calcolo degli ID 72
24 Hook nativi di React 73
25 Risultato finale del lavoro 74
26 Segnalazione degli errori topologici 75
27 Esportazione in configurazione Flink 76

Introduzione

Industria 5.0 è il termine che definisce un nuovo paradigma evolutivo delle realtà aziendali. Questa rivoluzione serve a portare dei cambiamenti significativi all'interno delle imprese per riuscire a raggiungere nuovi livelli di efficienza e, al contempo, porre al centro delle attività d'impresa i dipendenti e, più in generale, le persone che lavorano all'interno di ogni realtà. Si vuole promuovere un approccio sostenibile, con l'obiettivo di migliorare le prestazioni aziendali attraverso lo sviluppo e l'implementazione di nuove tecnologie utili a supportare un consumo ridotto di energia. A differenza dell'*Industria 4.0*, che si focalizza sull'ottimizzazione dei processi produttivi tramite tecnologie come l'intelligenza artificiale e l'IoT, il cui acronimo sta per *Internet of Things*, il paradigma dell'Impresa 5.0 vuole porre l'attenzione sull'importanza del giusto lavoro uomo-macchina e sulla sostenibilità, sia ambientale che organizzativa. Il suo impatto sulle aziende è significativo perché cerca di ridefinire i processi produttivi utilizzando un approccio innovativo all'organizzazione dei flussi lavorativi e alla progettazione. Inoltre, mira a investire in nuovi macchinari e tecnologie, per migliorare le infrastrutture digitali e fornire una maggiore efficienza dal punto di vista energetico. In questo scenario, le fabbriche e gli impianti sono disseminati di quelli che abbiamo già nominato come sensori IoT, i quali generano una quantità elevata di dati in tempo reale. **Qual è il problema?** Questi dati sono utili solo nel momento in cui vengono trasformati in informazioni nel più breve tempo possibile. Aspettare ore per poter essere analizzati non è più sufficiente. Per rendercene conto, immaginiamo un sensore di temperatura che rileva un picco anomalo: in questo caso è necessario spegnere il macchinario nel più breve tempo possibile.

Il corrente elaborato di tesi, frutto del lavoro svolto durante il periodo di tirocinio presso l'Università Politecnica delle Marche, si colloca in uno dei vari punti fondamentali per il passaggio verso l'Industria 5.0, presentando lo sviluppo di un editor web volto a fornire uno strumento di modellazione no-code per definire i flussi di elaborazione dei dati raccolti dai sensori o dai dispositivi IoT in ambito industriale. Tali dati devono poter essere aggregati, filtrati, uniti o trasformati a seconda delle necessità, così da essere poi pronti per la loro successiva elaborazione. L'editor sviluppato è un tool grafico utile a realizzare uno schema, che possa essere sia sequenziale che ad albero, di operazioni che devono essere applicate ai dati. Ognuna di esse è rappresentata graficamente da un blocco, identificato dal nome dell'operazione e collegato ad altri nodi secondo precise regole strutturali. Lo schema così composto raffigura uno *workflow* di dati che, una volta completato e validato, può essere esportato per configurare l'esecuzione delle operazioni sul flusso. Un esempio astratto di questa sequenza di elaborazione è mostrato nella Figura 1.

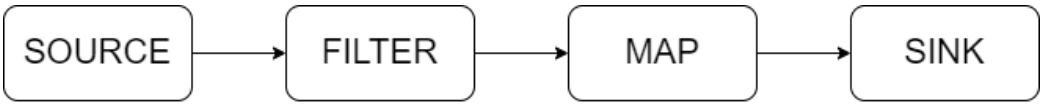


Figura 1: esempio generico di un flusso di operazioni.

Per concludere questa introduzione, verranno di seguito descritti i capitoli che compongono l'elaborato:

- **Capitolo 1: Contesto applicativo** - descrizione dello scenario in cui il progetto di tesi si colloca, mostrando in generale l'obiettivo della sua realizzazione. Inoltre, viene introdotto il concetto di *Data Stream* e il collegamento con *Apache Kafka*.

- **Capitolo 2: Requisiti di progetto e casi d'uso** - analisi preliminare alla progettazione in cui si presentano gli obiettivi specifici del progetto, l'analisi dei requisiti e i casi d'uso.

- **Capitolo 3: Tecnologie utilizzate** - vengono analizzati e delineati gli strumenti tecnologici, in termini di software e linguaggi di programmazione, utilizzati per la creazione dell'editor.

- **Capitolo 4: Progettazione** - presentazione della fase di progettazione dell'editor, nella quale vengono descritti gli operatori di *Apache Flink*, l'architettura del software realizzato e la sua interfaccia tramite mockup.

- **Capitolo 5: Implementazione** - effettiva realizzazione dell'editor attraverso lo sviluppo di codice, la raffigurazione delle funzionalità principali attraverso mock-up e le soluzioni significative applicate.

- **Capitolo 6: Conclusioni e sviluppi futuri** - considerazioni generali finali relative al lavoro svolto e le possibili attività correlate che potranno essere svolte in seguito.

1 Contesto applicativo

Il progetto di ricerca HOMEY [1], il cui titolo completo è *Human-centric IoE-based Framework for Supporting the Transition Towards Industry 5.0*, è un'iniziativa scientifica di rilevanza nazionale finanziata dall'Unione Europea nell'ambito del programma NextGenerationEU (Ministero dell'Università e della Ricerca - MUR). Il progetto mira a ridefinire le modalità di interazione tra gli operatori umani e le tecnologie digitali nei moderni contesti industriali, facilitando la transizione del paradigma tecnologico dell'Industria 4.0 a quello più sostenibile, resiliente e incentrato sull'uomo proprio dell'Industria 5.0, come descritto nell'introduzione dell'elaborato. HOMEY propone un framework basato sull'*Internet of Everything* (IoE), cioè un'evoluzione dell'IoT che integra non solo dispositivi fisici, ma anche persone, processi e flussi di dati in un unico ecosistema.

1.1 Gestione dei Data Stream e Stream Processing

L'evoluzione verso il paradigma dell'Industria 5.0 [2] impone una gestione avanzata dei flussi informativi generati dagli ambienti produttivi. In un ecosistema basato sull'IoE, l'integrazione di sensori IoT, macchinari industriali, processi aziendali e operatori umani, produce una grande quantità di dati continui ed eterogenei. In questo contesto si definisce *Data Stream* una sequenza illimitata (unbounded), ordinata e ad alta frequenza di eventi o misurazioni generati in tempo reale. A differenza delle tradizionali architetture di elaborazione *Batch* , nelle quali i dati vengono prima accumulati in memoria/database e analizzati periodicamente a intervalli prestabiliti, il modello di *Stream Processing* elabora i singoli eventi nell'istante esatto in cui vengono generati o ricevuti. Questa transizione è fondamentale nell'Industria 5.0 per supportare la sostenibilità e la centralità dell'uomo: identificare tempestivamente un'anomalia termica in un macchinario o una situazione di pericolo per l'operatore richiede latenze nell'ordine dei millisecondi, che, solo un'elaborazione in tempo reale su flussi di dati può garantire.

1.2 Ingestione ed inoltro dei dati

Per gestire l'elevata portata e l'eterogeneità dei dati prodotti dai dispositivi IoE, l'infrastruttura richiede uno strato di ingestione robusto, distribuito e a bassa latenza. A tale scopo viene impiegato Apache Kafka [3], una piattaforma open source per il *Distributed Event Streaming* basata sul modello di comunicazione *publisher-subscriber*, pattern di messaggistica asincrona volto al disaccoppiamento tra chi invia le informazioni e chi le riceve. Di fatto,

all'interno dell'architettura del sistema, Kafka funge da *Event Broker*, separando fisicamente e logicamente i produttori di dati (sensori, gateway IoT, processi IT) dai sistemi consumatori. I dati vengono organizzati all'interno di strutture chiamate *topic*, suddivise a loro volta in partizioni per garantire la scalabilità orizzontale e la tolleranza ai guasti, come mostrato in Figura 2. Infine, Kafka garantisce che i flussi di eventi siano memorizzati su disco in modo sicuro e redistribuiti con bassissima latenza, permettendo a molteplici microservizi di consumare e ripubblicare i flussi in maniera indipendente.

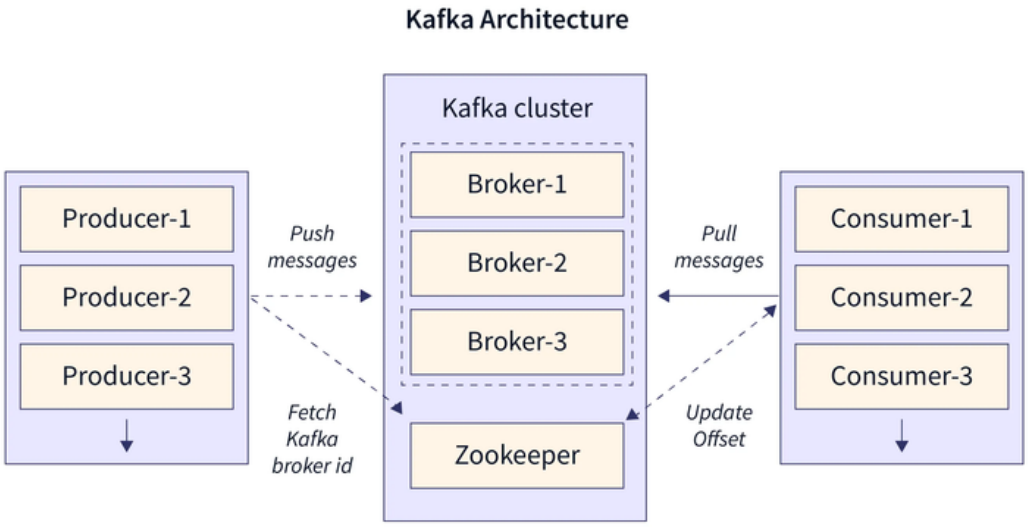


Figura 2: architettura di Apache Kafka e i suoi componenti.

1.3 Elaborazione in tempo reale

Mentre Kafka garantisce il trasporto e l'ingestione affidabile dei flussi, l'applicazione delle logiche di trasformazione, calcolo e filtraggio sui dati in transito è affidata ad Apache Flink [4]. Flink è un framework distribuito di *Stateful Stream Processing* progettato per elaborare flussi dati continui con bassissima latenza e throughput elevato. A differenza dei sistemi di elaborazione a micro-batch, Flink analizza ciascun evento *event-by-event* mantenendo uno stato interno della computazione (stateful processing) e garantendo semantiche di elaborazione *exactly-once*. Tale approccio garantisce che ciascun evento inviato venga elaborato e produca effetti sui sistemi a valle esattamente una volta, evitando sia la perdita di dati sia un'elaborazione duplicata. Come mostrato in Figura 3, Flink gestisce nativamente sia flussi limitati, caratterizzati da un inizio e una fine ben individuabili, sia flussi illimitati, generati continuamente nel tempo senza una fine definita. Quest'ultimi sono quelli di nostro interesse e che necessitano di essere elaborati in tempo reale, come le rilevazioni dei sensori.

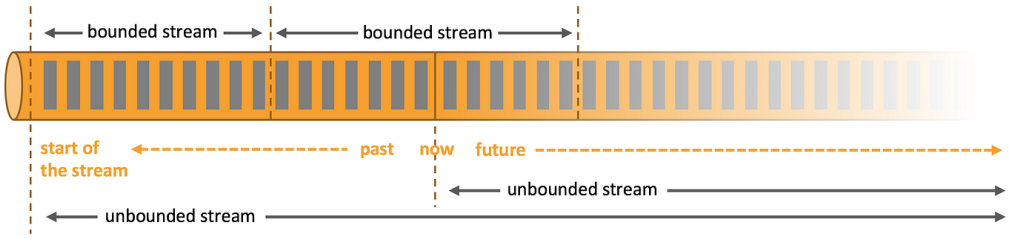


Figura 3: gestione flussi illimitati e limitati di Apache Flink.

1.4 Architettura del framework

L'architettura del framework presentato dal progetto HOMEY è progettata per la gestione dei flussi informativi industriali, attraverso la raccolta, l'elaborazione e la memorizzazione di dati eterogenei provenienti dall'ambiente industriale, con molteplici livelli indipendenti che collaborano tra loro, garantendo simultaneamente un'interazione in tempo reale con l'operatore umano. Ogni componente opera in modo indipendente e segue il paradigma publisher-subscriber, cioè consuma il flusso dal componente precedente, lo elabora e ripubblica il flusso sul componente successivo, come mostrato in Figura 4.

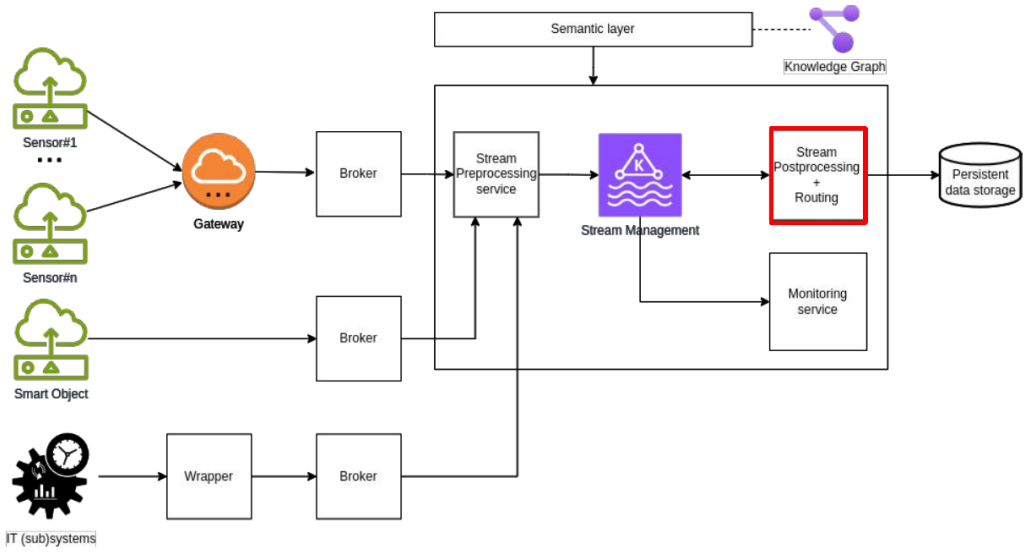


Figura 4: architettura del framework costituita da microservizi.

Come evidenziato in rosso nello schema architetturale, particolare rilievo assume il modulo di *Stream Postprocessing and Routing*, il quale si interfaccia con il sistema di *Stream Management* per applicare regole di trasformazione, aggregazione e instradamento sui dati. L'editor web sviluppato in questo elaborato rappresenta l'interfaccia di configurazione no-code proprio a questo specifico livello della pipeline. Infine, un aspetto chiave di questo framework è il trattamento di ciascun dispositivo o sorgente dati come produttore di flussi di dati (Data Streams). Questo approccio standardizza la gestione dei dati garantendo che essi siano gestiti in modo uniforme. Tale modello basato su flussi semplifica l'integrazione di nuovi dispositivi e sorgenti dati nel sistema, poiché vengono tutti processati e strutturati in modo coerente.

1.5 Stream Postprocessing e configurazione dei flussi

In questo scenario di ricerca, il flusso continuo di dati gestito dal sistema deve poter essere manipolato, filtrato ed elaborato in maniera dinamica e flessibile prima dell'archiviazione o dell'inoltro alle applicazioni a valle. Questa attività viene delegata allo Stream Postprocessing and Routing, il quale rappresenta lo snodo in cui vengono applicate le regole di trasformazione finale e le politiche di persistenza sulla base dei metadati e delle configurazioni estratte dal *Knowledge Graph*. Il suo inserimento a questo punto della pipeline permette di soddisfare diverse esigenze chiave:

- **Ottimizzazione dello storage e filtraggio:** i dati grezzi generati dai sensori possono contenere ridondanze. Il servizio applica filtri mirati per ridurre il volume dei dati memorizzati, salvando esclusivamente le informazioni rilevanti in modo da garantire una maggiore efficienza dell'uso dei database persistenti.

- **Arricchimento e routing intelligente:** i flussi in transito possono essere arricchiti con informazioni contestuali derivate dal Knowledge Graph (come la posizione dei dispositivi o i ruoli degli operatori), assicurando che i dati vengano instradati correttamente verso i sistemi di monitoraggio o le interfacce utente dedicate.

- **Flessibilità e manutenibilità:** grazie alla separazione tra il motore di calcolo principale (Stream Management) e il modulo di Postprocessing, l'architettura risulta altamente modulare. Eventuali variazioni riguardo alle politiche di salvataggio o nelle regole di business possono essere recepite a livello di routing senza compromettere la stabilità dei flussi di elaborazione in tempo reale.

2 Requisiti di progetto e casi d'uso

La fase preliminare alla progettazione è stata dedicata all'analisi di tre aspetti fondamentali: la discussione degli obiettivi specifici, l'analisi dei requisiti e la definizione dei casi d'uso. Lo scopo principale del sistema è la modellazione grafica e intuitiva di workflow di elaborazione dati, fornendo un'interfaccia no-code capace di astrarre la complessità sottostante e generare una rappresentazione strutturata ed esportabile, concepita per configurare i flussi di elaborazione e instradamento all'interno dell'architettura di riferimento.

2.1 Obiettivi specifici

L'editor web non ha il compito di eseguire concretamente le operazioni che sono state definite dal workflow sui dati, bensì quello di modellare e validare la logica del flusso di elaborazione, traducendola in una descrizione formale esportabile. Gli Obiettivi Specifici individuati dall'editor sono i seguenti:

- **Modellazione visuale a grafi:** fornire un canvas interattivo in cui l'utente possa posizionare e connettere nodi rappresentanti operazioni tipiche dello Stream Processing basate sull'ecosistema Apache Flink.
- **Supporto a varie tipologie di flusso:** consentire la creazione sia di strutture di elaborazione puramente sequenziali, sia non lineari con ramificazioni e ricongiungimenti dei nodi.
- **Configurazione delle proprietà dei singoli blocchi:** permettere all'utente di specificare parametri per ogni tipo di operazione attraverso dei pannelli dedicati all'interno dell'interfaccia grafica.
- **Validazione strutturale del workflow:** impostare delle regole per la connessione e presenza dei nodi, così da garantire una corretta realizzazione del grafo.
- **Esportazione e interoperabilità:** convertire la struttura grafica del workflow e i relativi parametri di configurazione in un file descrittivo in formato JSON, interpretabile dai servizi all'interno del framework a microservizi.
- **Usabilità e importanza dell'utente:** rispettare i principi dell'Industria 5.0 ponendo l'operatore umano al centro, riducendo l'aspetto tecnico ed evitando la struttura manuale di codice per la definizione della pipeline di Stream Processing.

2.2 Analisi dei requisiti

Un passaggio fondamentale per guidare in modo consapevole le successive fasi di progettazione e implementazione è l'analisi dei requisiti. Con il termine "requisiti" si intende l'insieme delle funzionalità, dei comportamenti e dei vincoli che il sistema software deve soddisfare per rispondere efficacemente alle esigenze degli utenti e del contesto operativo. I requisiti dell'applicativo sono stati formalizzati e divisi in funzionali, quindi ciò che il sistema deve fare, e non funzionali, che rappresentano i vincoli qualitativi e operativi del sistema.

2.2.1 Requisiti funzionali

I requisiti funzionali sono descritti di seguito in linguaggio naturale.
Lo scopo principale dell'utente è quello di definire un workflow operativo, quindi per iniziare la sua creazione, il sistema deve fornire una lista di blocchi contenuti in un pannello apposito. Ogni blocco deve poter essere selezionato e trascinato all'interno del canvas. Ogni blocco deve avere un identificativo (ID) univoco dato dal nome dell'operazione che il blocco stesso rappresenta e deve poter essere collegato manualmente a uno o più blocchi, seguendo delle regole specifiche definite dal sistema, il quale segnalerà gli errori commessi all'interno di una lista, se non vengono rispettate. Per poter vedere le proprietà di ognuno di essi, il sistema deve mettere a disposizione un ulteriore pannello che contiene le informazioni del blocco, quelle generali e quelle specifiche, le quali cambiano a seconda del tipo di blocco e devono poter essere definite dall'utente attraverso un form di testo/selezione. Infine, il sistema deve fornire una funzionalità per esportare/importare il workflow. Per quanto riguarda l'esportazione, può avvenire in due modi:

- **Bozza:** Il workflow può essere esportato, anche se presenta errori che riguardano l'assenza di determinati blocchi e collegamenti oppure è vuoto, in formato JSON. Il fil esportato contiene le proprietà dei blocchi, i successivi collegamenti di ognuno e le loro posizioni nel canvas.
- **Configurazione Flink:** Il workflow può essere esportato solo se non presenta errori, in formato JSON. Il file esportato comprende esclusivamente le proprietà dei blocchi e i collegamenti successivi di ognuno, struttura coerente con l'elaborazione di Apache Flink.

Anche l'importazione può avvenire in due modalità:

- **Bozza:** È la procedura migliore per continuare a lavorare sul workflow se ne si ha la necessità, in quanto il suo file JSON contiene le posizioni dei blocchi. In questo modo, è possibile ricostruire la stessa struttura che aveva il grafo prima della sua esportazione.
- **Configurazione Flink:** È la procedura sconsigliata per continuare a lavorare sul grafo, ma possibile. Questo perché il corrispettivo file JSON non contiene la posizione dei blocchi, quindi il workflow viene ricostruito con una disposizione a griglia.

I requisiti funzionali vengono formalmente definiti come segue:

ID	Nome Requisito	Descrizione
RF-01	**Aggiunta blocchi**	L'utente deve poter inserire nella canvas di lavoro i blocchi rappresentanti le operazioni supportate.

ID	Nome Requisito	Descrizione
RF-02	**Aggiunta collegamenti**	L'utente deve poter creare collegamenti orientati tra le uscite e gli ingressi dei blocchi per definire la direzione del flusso di dati.

ID	Nome Requisito	Descrizione
RF-03	**Configurazione parametri**	Selezionando un blocco, l'utente deve poter visualizzare e modificare un form con i parametri specifici per l'operazione associata.

ID	Nome Requisito	Descrizione
RF-04	**Cancellazione ed editing**	L'utente deve poter eliminare blocchi o connessioni esistenti e riposizionare liberamente i nodi all'interno della canvas.

ID	Nome Requisito	Descrizione
RF-05	**Esportazione JSON**	Il sistema deve fornire una funzione generare/scaricare un file in formato JSON contenente struttura e le proprietà del workflow.

ID	Nome Requisito	Descrizione
RF-06	**Importazione JSON**	L'utente deve poter caricare un file JSON di configurazione precedentemente salvato per ricostruire la struttura visuale del workflow ed effettuarne modifiche.

ID	Nome Requisito	Descrizione
RF-07	**Validazione del workflow**	Il sistema deve verificare che la struttura creata sia valida, segnalando eventuali errori/assenza di blocchi e/o collegamenti.

2.2.2 Requisiti non funzionali

I requisiti non funzionali sono descritti di seguito in linguaggio naturale.
Essendo l'editor pensato come un applicativo web, deve essere eseguito interamente all'interno dell'ambiente runtime fornito dal browser web dell'utente, garantendo il completo disaccoppiamento tra il frontend di modellazione e il backend di esecuzione (Stream Postprocessing Service). Questo comporta che l'editor debba funzionare correttamente su diverse tipologie di browser, quindi essere un tool grafico leggero e portabile, mettendo a disposizione un'interfaccia fluida e altamente reattiva, azzerando le latenze di rete durante le operazioni eseguite sul canvas. Per garantire una manutenzione efficace ed efficiente del software, e quindi del codice, esso deve essere strutturato in componenti riutilizzabili, ognuno dei quali opera in maniera indipendente, facilitando, all'occorrenza, l'aggiunta di nuovi blocchi e, quindi, operazioni. Infine, dato che lo scopo dell'editor è quello di esportare il workflow creatosi in un file JSON, questo deve avere una struttura sintattica coerente con le API dello Stream Postprocessing Service, per garantirne compatibilità e integrazione.

I requisiti non funzionali sono di seguito formalmente definiti:

ID	Nome Requisito	Categoria	Descrizione
RNF-01	**Esecuzione Client-Side**	**Architettura**	L'editor deve essere completamente fruibile tramite browser web senza dipendere dall'esecuzione locale di motori di stream processing.

ID	Nome Requisito	Categoria	Descrizione
RNF-02	**Usabilità e Responsività**	**Utilità**	L'interfaccia grafica deve garantire tempi di risposta immediati alle azioni di drag-and-drop, zoom e panoramica del canvas.

ID	Nome Requisito	Categoria	Descrizione
RNF-03	**Modularità e Estendibilità**	**Manutenibilità**	Il codice deve essere strutturato in componenti riutilizzabili e indipendenti per facilitare l'aggiunta futura di nuovi tipi di blocchi.

ID	Nome Requisito	Categoria	Descrizione
RNF-04	**Portabilità**	**Compatibilità**	L'applicazione deve funzionare in maniera uniforme sui principali browser web moderni.

ID	Nome Requisito	Categoria	Descrizione
RNF-05	**Conformità JSON**	**Interoperabilità**	Il file esportato deve rispettare la sintassi JSON standard per consentire l'integrazione immediata con le API dello Stream Postprocessing Service.

2.3 Casi d'uso

I casi d'uso (*Use Cases*) descrivono le interazioni tra l'utente dell'editor, cioè il progettista di flussi o operatore, e il sistema software. Ogni caso d'uso verrà descritto singolarmente, in termini di un'azione costituita da una successione di passi, compiuta dall'operatore, vale a dire un utente che necessita di modellare una pipeline di elaborazione su flussi dati provenienti da sensori o dispositivi industriali senza dover scrivere codice a mano.

Caso d'uso UC-01: creazione e modellazione di un nuovo workflow

- Attore principale: utente.
- Scopo: creare un nuovo workflow combinando blocchi sorgente, intermedi e di destinazione.
- Precondizioni: l'applicazione web è aperta nel browser ed è presente un canvas vuoto.
- Sequenza principale degli eventi:
    1. L'utente trascina un blocco di tipo Source dal pannello di sinistra al canvas.
    2. L'utente collega gli output dei blocchi precedenti agli input dei blocchi successivi trascinando i collegamenti tra le corrispettive maniglie di ingresso e uscita.

3. L'utente inserisce un blocco di chiusura di tipo Sink e vi collega l'ultimo blocco inserito.

- Eccezioni / flussi alternativi:

2.1 Connessione non consentita: se l'utente prova ad effettuare un collegamento non valido, il sistema impedisce il collegamento e mostra un avviso di notifica a schermo.

Caso d'uso UC-02: configurazione dei parametri di un blocco

- Attore Principale: utente.

- Scopo: definire le proprietà specifiche di un'operazione associata a un blocco nel workflow.

- Precondizioni: il blocco deve esistere nel canvas ed essere selezionato.

- Sequenza principale degli Eventi:

1. L'utente seleziona un blocco presente nel canvas cliccandoci sopra.
2. Il sistema apre il pannello destro delle proprietà mostrando i campi di configurazione relativi all'operazione del blocco selezionato.
3. L'utente compila o modifica i valori dei parametri specifici.
4. Lo stato interno del blocco viene aggiornato.

Caso d'uso UC-03: esportazione del workflow in formato JSON

- Attore Principale: utente.
- Scopo: generare e scaricare il file JSON contente l'intera definizione strutturale e configurativa del workflow.
- Precondizioni: l'applicazione web è aperta nel browser ed è presente un canvas vuoto.
- Sequenza principale degli eventi:
    1. Il sistema rileva degli errori relativi alla necessità di inserire determinati blocchi e collegamenti, notificandolo a schermo.
    2. L'utente clicca sul pulsante "Esporta Bozza".
    3. Il sistema esegue una scansione dei nodi e degli archi presenti nel canvas.
    4. Il sistema formatta la topologia e i relativi parametri delle operazioni in una struttura JSON.
    5. Il sistema avvia il download del file JSON nel file system dell'utente.
- Eccezioni / flussi Alternativi:
    - 1.1 Il sistema non rileva alcun errore relativo all'assenza di determinati blocchi o collegamenti.
    - 2.1 L'utente clicca sul pulsante "Configurazione Flink".

Caso d'uso UC-04: importazione e modifica di un workflow esistente

- Attore Principale: utente.

- Scopo: importare un file JSON precedentemente esportato per visualizzare o modificare la struttura del grafo.

- Precondizioni: il file che si vuole importare deve essere in formato JSON e deve contenere delle proprietà specifiche per essere correttamente ricostruito in blocchi e collegamenti.

- Sequenza principale degli eventi:

    1. L'utente clicca sul pulsante "Importa File" che si trova nel pannello di sinistra.
    2. L'utente seleziona il file dal proprio file system.
    3. Il sistema analizza la struttura del file JSON.
    4. Il sistema rigenera automaticamente i blocchi, i parametri e le connessioni sul canvas di lavoro.

- Eccezioni / flussi alternativi:

    4.1 Il sistema, a seguito della selezione dell'utente, rileva delle anomalie nell'importazione e blocca il processo mostrando una notifica a schermo.

3 Tecnologie utilizzate

Per la realizzazione dell'editor web sono state utilizzate in totale sei tecnologie, collocate principalmente nell'ambito dello sviluppo web. Per quanto riguarda l'ambiente di sviluppo, è stato utilizzato Visual Studio Code, per la sua versatilità e per le numerose funzionalità che mette a disposizione. Come strumento principale per lo sviluppo dell'editor, è stato adottato React, framework di supporto allo sviluppo web, che fornisce un significativo numero di librerie, tra cui React Flow, e strumenti di appoggio per la scrittura di codice, il quale è stato sviluppato attraverso i tre fondamentali della realizzazione web: HTML, JavaScript e CSS. Infine, per tenere traccia del lavoro svolto, sono stati impiegati Git e GitHub, importantissimi per il backup del progetto e per tenere conto delle sue modifiche in corso d'opera.

3.1 Visual Studio Code

Visual Studio Code [5], comunemente abbreviato in VS Code, è un editor di codice sorgente open source, sviluppato da Microsoft, per i sistemi operativi Windows, macOS e Linux. A differenza di un ambiente di sviluppo integrato (IDE) tradizionale, orientato in genere a uno specifico linguaggio, VS Code combina la reattività di un semplice editor di testo con funzionalità avanzate come il debug del codice, l'integrazione con sistemi di controllo versione e un vasto ecosistema di estensioni. L'interfaccia è semplice e intuitiva, come mostrato dalla Figura 5.

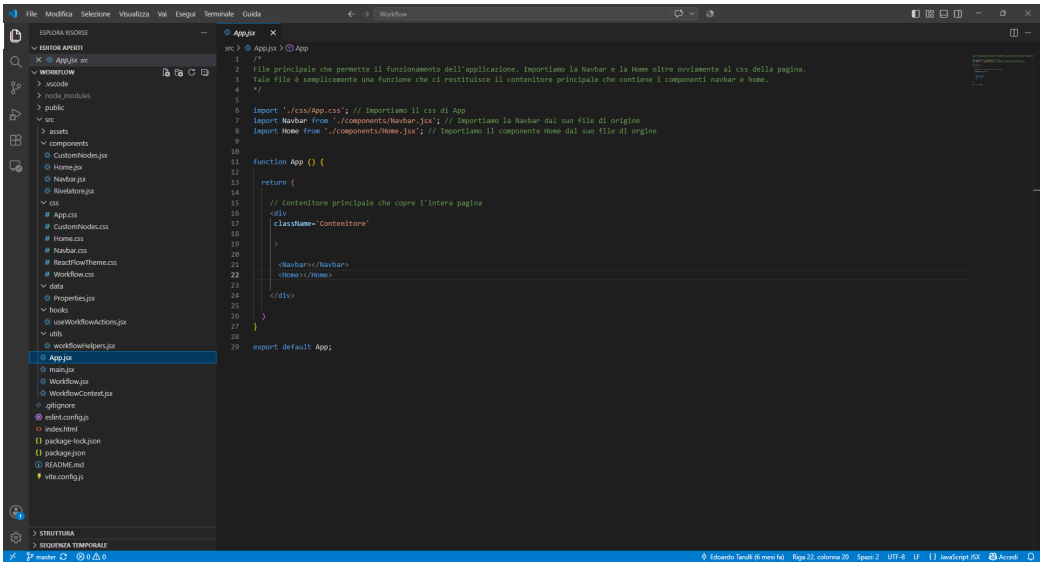


Figura 5: Visual Studio Code, editor per lo sviluppo software

Le funzionalità principali di VS Code, che lo rendono uno degli editor più utilizzati per lo sviluppo software, sono:

- **IntelliSense (autocompletamento intelligente):** fornisce suggerimenti di completamento del codice basati su tipi di variabili e sulle definizioni delle funzioni.
- **Debugging integrato:** permette di eseguire il debug direttamente all'interno dell'editor, con la possibilità di impostare i punti di interruzione (breakpoints).
- **Controllo versione con Git:** integra nativamente il supporto a Git, strumento che verrà analizzato in seguito. Permette di visualizzare le differenze delle modifiche e relativi commit, gestire i branch e caricare codice su repository remoti.
- **Estensioni:** presenta un marketplace molto vario che mette a disposizione un vasto numero di estensioni per supportare nuovi linguaggi, strumenti di formattazione, temi visivi e integrazioni cloud.
- **Terminale integrato:** include un terminale che consente di eseguire comandi di shell direttamente all'interno della finestra di lavoro.
- **Personalizzazione avanzata:** consente la configurazione di file, supporto per tasti di scelta rapida personalizzabili e una vasta scelta di temi grafici.
- **Sviluppo remoto ed estensioni AI:** supporta lo sviluppo in ambienti remoti e l'integrazione nativa o tramite plugin con assistenti basati su intelligenza artificiale.

3.2 React



React [6] è una libreria JavaScript open source, dichiarativa e basata su componenti, sviluppata e mantenuta da Meta insieme a una vasta community di sviluppatori indipendenti e aziende. React è progettato per un compito ben preciso: costruire interfacce utente (UI) dinamiche, reattive ed efficienti, in particolare per la creazione di SPA, il cui acronimo sta per *Single Page Application*. Nel contesto dello sviluppo moderno React si occupa di gestire il livello della vista (*view*), semplificando la creazione e la gestione di componenti grafici interattivi che rispondono in tempo reale ai cambiamenti dei dati. Le caratteristiche fondamentali di React, che contraddistinguono la sua utilità allo sviluppo web, sono:

- **Architettura a componenti:** l'interfaccia utente viene suddivisa in blocchi di codice autonomi e riutilizzabili chiamati componenti, assicurando una efficiente riutilizzabilità del codice.
- **Virtual DOM:** React mantiene in memoria una rappresentazione virtuale del DOM (*Document Object Model*) reale. Quando lo stato dell'applicazione cambia, React confronta il Virtual DOM con quello precedente e aggiorna solo gli elementi del browser effettivamente modificati, migliorando le prestazioni.
- **Flusso dati monodirezionale:** i dati passano dal componente genitore ai componenti figlio tramite le *props*, rendendo il tracciamento dei dati e il *debugging* molto più semplici e garantendo una maggiore flessibilità del codice.

3.2.1 React Flow

React Flow [7] è una libreria altamente personalizzabile progettata specificatamente per creare interfacce basate su nodi, diagrammi di flusso, editor di workflow e mappe concettuali all'interno di applicazioni React. Un esempio è mostrato nella Figura 6.

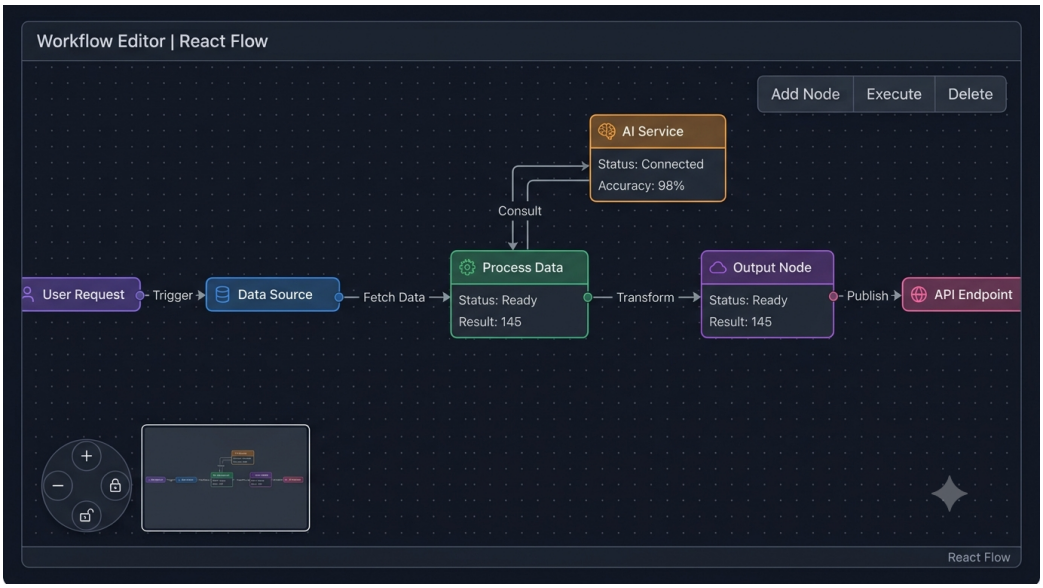


Figura 6: Esempio di un'interfaccia creata con React Flow

Questa libreria rappresenta un'ulteriore motivazione a supporto dell'adozione di React, in quanto mette a disposizione una vasta gamma di risorse per lo sviluppo dell'UI, tra cui:

- **Gestione di nodi ed archi:** fornisce un'astrazione completa per definire gli elementi grafici e le connessioni logiche o visive tra di essi.

- **Interattività integrata:** include già funzionalità di base pronte all'uso come drag-and-drop, zoom, selezione multipla e connessione manuale tra nodi.

- **Componenti di supporto inclusi:** presenta controlli integrati per la gestione dell'interfaccia, come la minimappa, zoom, fissaggio della visuale e sfondi personalizzati.

- **Hook nativi e personalizzabili:** mette a disposizione funzionalità pronte all'uso o creabili da zero tramite la parola chiave "use", per gestire facilmente lo stato del diagramma.

3.3 HTML



HTML [8], il cui acronimo sta per *HyperText Markup Language*, è il linguaggio di contrassegno standard utilizzato per creare e strutturare pagine web. Non è un linguaggio di programmazione, in quanto non gestisce logica o algoritmi, ma un linguaggio di *markup* che indica ai browser come organizzare e visualizzare i contenuti come testi, immagini, collegamenti, form e tutti gli elementi appartenenti a una pagina web. Il funzionamento di HTML è contraddistinto da:

- **Struttura a tag:** la sintassi è costituita da tag racchiusi tra parentesi angolari, come <p>, , per definire gli elementi dell'interfaccia.
- **Architettura ad Albero:** gli elementi HTML sono organizzati in modo gerarchico, quindi con elementi genitori e figli, formando una struttura chiamata DOM.
- **Indipendente dalla piattaforma:** il codice HTML viene interpretato allo stesso modo su qualsiasi dispositivo.
- **Semantica avanzata:** con l'introduzione di HTML5 sono stati introdotti tag semantici, come <header>, <nav>, <footer>, che migliorano la leggibilità del codice, l'accessibilità e l'ottimizzazione per i motori di ricerca.

3.4 JavaScript



JavaScript [9] è un linguaggio di programmazione dinamico, orientato agli oggetti e ad alto livello. È il componente fondamentale che rende le pagine web interattive e dinamiche, gestendo la logica di business lato client (browser). Le peculiarità di JavaScript, che lo rendono uno dei linguaggi più popolari e importanti al mondo per lo sviluppo software e la creazione di siti e applicazioni web, sono:

- **Linguaggio di scripting lato client:** viene eseguito direttamente dal motore JS del browser, consentendo di manipolare il DOM in tempo reale senza dover ricaricare la pagina.
- **Tipizzazione dinamica e debole:** le variabili non richiedono l'esplicita dichiarazione del tipo di dato e possono cambiare tipo durante l'esecuzione.
- **Programmazione asincrona:** utilizza meccanismi come async/await per eseguire operazioni come chiamate API o caricamenti di dati in background senza bloccare l'interfaccia utente.
- **Multi-paradigma:** supporta la programmazione orientata agli oggetti, la programmazione funzionale e quella imperativa.

3.5 CSS



CSS [10], il cui acronimo sta per *Cascading Style Sheets* è il linguaggio di stile utilizzato per descrivere la presentazione e l'aspetto visivo di un documento scritto in HTML. Gestisce il layout, i colori, i font, la responsività e le animazioni delle pagine web. Il suo utilizzo è essenziale perché offre svariate modalità di personalizzazione dell'aspetto visivo, in particolare:

- **Ereditarietà a cascata:** applica le regole di stile in base a una gerarchia di priorità ed ereditano le proprietà dagli elementi genitori.
- **Separazione delle responsabilità:** separa nettamente la struttura dei contenuti, gestita dall'HTML, dalla loro resa estetica e visiva.
- **Layout moderni:** offre potenti moduli di layout per disporre gli elementi in modo fluido e bidimensionale all'interno del viewport.
- **Design responsivo:** permette di adattare l'interfaccia a qualsiasi risoluzione dello schermo, quindi per diversi dispositivi.
- **Animazioni e transizioni:** consente di creare effetti visivi fluidi e interazioni senza necessità di utilizzare codice JavaScript.

3.6 Github



GitHub [11] è una piattaforma cloud di hosting per il controllo versione e la collaborazione nello sviluppo software. Si basa su Git, un sistema di controllo versione distribuito ideato da Linus Torvalds, e permette agli sviluppatori di tracciare le modifiche al codice sorgente, gestire progetti e collaborare in modo efficiente all'interno di team di qualsiasi dimensione. GitHub svolge un compito fondamentale e al contempo di grande rilievo, ovvero la memorizzazione dei progressi compiuti e la loro gestione, grazie alle funzionalità che fornisce:

- **Hosting di repository Git:** permette di salvare online sia repository pubblici sia privati, mantenendo l'intera cronologia dei *commit* (modifiche al codice).
- **Gestione del flusso di lavoro:** consente di proporre modifiche al codice tramite Pull Request, permettendo agli altri membri del team, se ci sono, di revisionare, commentare e approvare il codice prima di unire le modifiche (*merge*) al ramo principale.
- **GitHub actions:** piattaforma integrata di *Continuous Integration* e *Continuous Deployment* che permette di automatizzare fasi di test, build e rilascio dell'applicazione direttamente dal repository.
- **Tracciamento di problemi e progetti:** include strumenti nativi per la gestione dei bug, delle funzionalità e della pianificazione del lavoro.

4 Progettazione

A seguito dell'analisi dei requisiti funzionali e non funzionali, la fase di progettazione rappresenta il collegamento a livello concettuale tra le specifiche astratte del sistema e la sua effettiva implementazione software. Lo scopo principale di questa fase è definire l'architettura logica, le strutture dati e le scelte dell'interfaccia per garantire la creazione di un editor web intuitivo, estendibile e perfettamente integrato nell'architettura a microservizi del progetto HOMEY. L'intera fase è stata sviluppata per astrarre la complessità dei paradigmi di Stream Processing tipici di un motore ad alte prestazioni come Apache Flink, fornendo all'utente un ambiente grafico di creazione della pipeline di elaborazione e dei relativi parametri, formalizzati in un modello dati strutturato.

4.1 Modellazione degli operatori Flink

Nella progettazione dell'editor web, l'insieme delle primitive di elaborazione messe a disposizione dall'ecosistema di Apache Flink è stato ristretto e adattato alle specifiche esigenze del progetto HOMEY. Sono stati modellati otto operatori fondamentali, suddivisi in tre categorie principali: sorgenti, operazioni intermedie e destinazioni. Ciascun operatore è associato a un blocco grafico sul canvas e ad un insieme di parametri configurabili tramite il pannello delle proprietà.

1 **Operatori di sorgente:**

- **Source:** rappresenta il punto di ingresso dei dati all'interno della pipeline di Stream Processing, interfacciandosi con i broker di messaggistica per prelevare i flussi contigui generati dai sensori dell'ambiente industriale.

2 **Operatori di elaborazione e trasformazione:**

- **Filter:** applica un predicato booleano a ciascun evento in transito. Gli eventi che soddisfano la condizione vengono inoltrati a valle, mentre gli altri vengono scartati.
- **Map:** esegue una trasformazione uno-a-uno, prendendo in input un evento e restituendo un evento modificato.
- **KeyBy:** separa logicamente il flusso in partizioni logiche basate su una chiave comune, come l'ID, consentendo di eseguire operazioni successive raggruppate per entità.

- **Window:** suddivide i flussi di dati in blocchi finiti di tempo o di elementi, chiamati finestre temporali, permettendone l'analisi su intervalli temporali definiti.
- **Aggregate:** lavora in combinazione con il blocco di finestratura per calcolare metriche riassuntive sui dati raggruppati.
- **Union:** permette di unire due o più flussi di dati omogenei provenienti da rami differenti del grafo in un unico flusso di eventi combinato.

3 **Operatori di destinazione:**

- **Sink:** rappresenta il punto di chiusura della pipeline. Si occupa di instradare e persistere i dati elaborati verso i sistemi di memorizzazione a lungo termine, come un database relazionale, o verso i servizi di monitoraggio.

4.1.1 Configurazione delle proprietà

Ciascun blocco presente nel canvas dell'editor possiede una struttura dati duale, suddivisa in due tipologie di proprietà gestite in modo differenziato dall'interfaccia grafica:

1. **Proprietà comuni:** Sono presenti in ogni nodo indipendentemente dal tipo di operazione e identificano lo stato spaziale e logico all'interno del canvas di React Flow:
   - **id:** identificativo univoco del nodo.
   - **position:** coordinate cartesiane che definiscono il posizionamento grafico del blocco nel canvas.
   - **type:** tipologia di blocco.

2. **Proprietà di configurazione specifica:** Variano in base alla tipologia di nodo selezionato e vengono visualizzate dinamicamente nel pannello di configurazione al clic sul nodo, consentendo all'utente di compilare i form dedicati a ciascuno:
   - Source:
     - **topic:** nome del topic di riferimento da cui attingere i flussi.
     - **bootstrapServers:** indirizzo del server broker Kafka.
     - **timestampField:** campo temporale utilizzato per la gestione degli eventi nel tempo.

– Filter:

  - **filterCondition:** espressione booleana o condizione logica applicata per il filtraggio degli eventi in transito.

– Map:

  - **appliesToFields:** elenco dinamico di campi su cui applicare la trasformazione, con la possibilità per l'utente di aggiungere più righe e quindi più attributi, tramite l'interfaccia.

– KeyBy:

  - **appliesToField:** un singolo campo chiave utilizzato per il partizionamento logico del flusso.

– Window:

  * **windowType:** tipologia di finestratura selezionabile tramite apposito menù a tendina tra tre opzioni: *Tumbling* (finestre a dimensione fissa e non sovrapposte), *Hopping* (finestre scorrevoli con sovrapposizione) o *Count* (finestre basate sul numero di eventi anziché sul tempo).

  * Parametri specifici in base alla selezione:

    - **Tumbling: duration e size** che indicano rispettivamente la durata temporale della finestra e la sua dimensione.
    - **Hopping: duration e slide** che indicano rispettivamente la durata temporale e l'intervallo di scorrimento della finestra.
    - **Count: size e slide** che indicano rispettivamente il numero di elementi e il passo di scorrimento della finestra.

- Aggregate:
  - **aggregations:** elenco dinamico configurabile dall'utente, che permette di aggiungere uno o più attributi su cui calcolare le metriche, associando a ciascun attributo l'operazione statistica desiderata tra Min (minimo), Max (massimo), Sum (somma) o Avg (media).
- Union:
  - **Nessun parametro specifico:** l'operatore si limita a gestire la confluenza logica dei flussi in ingresso senza richiedere configurazioni testuali aggiuntive.
- Sink:
  - **topic:** nome del topic di destinazione verso cui inoltrare il flusso elaborato.
  - **bootstrapServers:** indirizzo del server broker di output.

4.1.2 Struttura dati

Per consentire all'editor di tradurre il grafo visivo disegnato sul canvas in un formato interoperabile con lo Stream Postprocessing Service, la pipeline di operatori viene formalizzata all'interno di una struttura dati basata su uno schema JSON. La struttura dati modella il workflow attraverso due collezioni principali:

1. **nodes** (nodi): contiene l'elenco di tutti i blocchi inseriti nel canvas. Ciascuno nodo racchiude le proprietà generali e le proprietà di configurazione specifica inserite dall'utente tramite il pannello di configurazione.

2. **edges** (archi): racchiude i collegamenti orientati tra le maniglie di input e output dei nodi, definendo la direzione e la sequenza del flusso di dati.

Poiché l'editor gestisce sia la fase di modellazione grafica che l'interoperabilità con il motore di esecuzione sottostante, la struttura JSON prodotta in output si differenzia in base allo stato di validazione del workflow:

1. **Esportazione in modalità Bozza (Draft):**

   - Viene utilizzata quando il workflow è incompleto o presenta errori strutturali, segnalati dal sistema.
   - Il file JSON generato include anche le proprietà di posizionamento spaziale.
   - L'obiettivo è quello di conservare la disposizione grafica esatta dei blocchi sul canvas, risultando ideale per essere reimportato in un secondo momento e consentire all'utente di riprendere il lavoro di modellazione esattamente da dove lo aveva interrotto.

2. **Esportazione in modalità Configurazione Flink:**

   - È consentita esclusivamente quando il workflow risulta formalmente valido e privo di errori.
   - Il file JSON viene ripulito da tutte le informazioni puramente grafiche di layout, come le coordinate spaziali e la data di esportazione, mantenendo unicamente la topologia logica degli operatori e i relativi parametri di configurazione.
   - Lo scopo consiste nell'ottenere una struttura dati snella e rigorosa, perfettamente compatibile con le specifiche e le API dello Stream Preprocessing Service per la traduzione diretta in *job* di Apache Flink.

Nella Figura 7, vengono riportati due esempi rappresentativi di file JSON generato dall'editor per una pipeline di stream processing, sia nella modalità Draft che nella modalità Configurazione Flink. Per questioni di semplicità, i JSON raffigurati descrivono un workflow costituito da tre blocchi, un Source, un Filter e un Sink, collegati in maniera sequenziale e con le proprietà specifiche di ognuno definite in maniera generica.

```
{
  "steps": [
    {
      "id": "source_1",
      "type": "source",
      "position": {
        "x": 226.4474641845186,
        "y": 333.60608704277877
      },
      "next": [
        "filter_1"
      ]
    },
    {
      "id": "filter_1",
      "type": "filter",
      "filter_condition": "\"specific condition\"",
      "position": {
        "x": 481.13521188960453,
        "y": 320.25294626288553
      },
      "next": [
        "sink_1"
      ]
    },
    {
      "id": "sink_1",
      "type": "sink",
      "position": {
        "x": 798.8743619910862,
        "y": 318.55103071316637
      },
      "sinkType": "print"
    }
  ],
  "status": "draft",
  "dataEsportazione": "2026-09-04T11:30:26.747Z"
}
```

```
{
  "steps": [
    {
      "id": "source_1",
      "type": "source",
      "topic": "\"specific topic\"",
      "bootstrap_servers": "\"specific server\"",
      "timestamp_field": "\"specific timestamp\"",
      "next": [
        "filter_1"
      ]
    },
    {
      "id": "filter_1",
      "type": "filter",
      "filter_condition": "\"specific condition\"",
      "next": [
        "sink_1"
      ]
    },
    {
      "id": "sink_1",
      "type": "sink",
      "topic": "\"specific topic\"",
      "bootstrap_servers": "\"specific server\"",
      "sinkType": "print"
    }
  ]
}
```


Figura 7: Confronto dei file JSON in entrambe le modalità: Draft (sopra) e Configurazione Flink (sotto).

4.2 Esempi applicativi di modellazione dei workflow

Al fine di validare concretamente le capacità espressive dell'editor e la sua aderenza ai paradigmi di elaborazione di Apache Flink definiti in fase di progettazione, vengono di seguito presentati due esempi significativi di workflow che rappresentano due contesti applicativi. Coerentemente con la modellazione logica, i grafi rappresentano la topologia pura delle pipeline, evidenziando il concatenamento tra operatori di sorgente, trasformazione e destinazione.

4.2.1 Pipeline di monitoraggio e filtraggio IoT con finestratura

Il primo scenario, come riportato in Figura 8, modula l'acquisizione continua di flussi di dati da sensori industriali, l'isolamento selettivo delle anomalie termiche e la successiva aggregazione temporale.

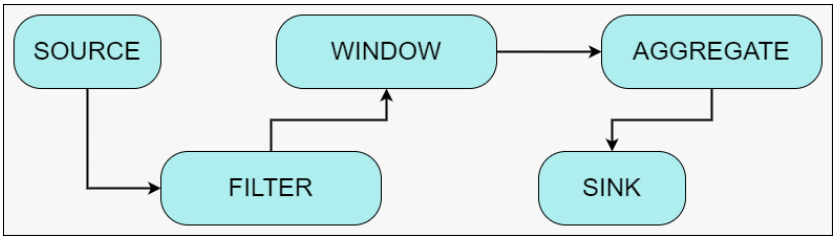


Figura 8: esempio di pipeline per il monitoraggio IoT con filtraggio delle anomalie e finestratura temporale.

Il flusso prende avvio da un nodo Source configurato su un broker Kafka per l'ascolto dei sensori di temperatura. Il flusso grezzo alimenta un blocco Filter, impostato con un predicato booleano per intercettare unicamente i record che soddisfano la condizione definita (temperature $>$ 85.0). I dati filtrati vengono

quindi convogliati in un nodo Window (strutturato a finestra temporale fissa di 60 secondi) e aggregati tramite l'operatore Aggregate per il calcolo delle metriche statistiche di media e massimo, prima di essere inoltrati al nodo Sink finale per la persistenza.

4.2.2 Unione di flussi paralleli

Il secondo scenario, illustrato in Figura 9, mostra la gestione di flussi convergenti, sfruttando la peculiarità dell'operatore Union di accettare molteplici collegamenti in ingresso per fondere flussi provenienti da rami differenti.

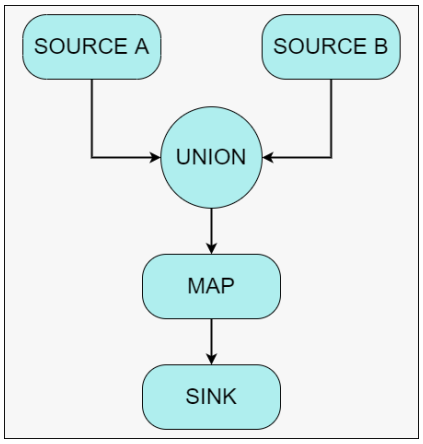


Figura 9: esempio di pipeline con convergenza di flussi multipli tramite l'operatore Union.

In questo caso specifico, due nodi Source indipendenti acquisiscono rispettivamente i dati di assemblaggio da una Linea A e i dati di collaudo da una Linea B. Entrambi i flussi convergono nel nodo Union. L'output combinato viene successivamente normalizzato da un blocco Map prima della serializzazione definitiva verso il sistema di destinazione gestito dal nodo Sink.

4.3 Progettazione dell'architettura software

L'applicativo è stato sviluppato come una Single Page Application utilizzando la libreria React. L'architettura software segue fedelmente il pattern *Component-Based*, in cui l'interfaccia utente è suddivisa in componenti modulari, riutilizzabili e indipendenti.
Per garantire una netta separazione delle responsabilità (*Separation of Concerns*) e una facile manutenibilità, l'architettura è stata divisa nei seguenti macro-moduli funzionali:

- **Entry point e routing (`main.jsx, App.jsx`):** il punto di ingresso dell'applicazione configura il sistema di navigazione tramite la libreria `react-router-dom`. Questo permette la transizione fluida tra le due viste principali, la `Home.jsx` e la `Workflow.jsx`, senza richiedere il ricaricamento della pagina, garantendo le prestazioni tipiche delle SPA.

- **Gestione dello stato globale (`WorkflowContext.jsx`):** una delle sfide architetturali consiste nel mantenere inalterato lo stato del diagramma (nodi, collegamenti e impostazioni grafiche) quando si passa dalla schemata del workflow alla home e vicecersa. Per risolvere questo problema, è stato implementato un *State Management* centralizzato sfruttando la *Context API* nativa di React (`WorkflowProvider`). Questo layer avvolge l'intero sistema di rotte, disaccoppiando i dati dal ciclo di vita del singolo componente.

- **Presentazione e UI (`Home.jsx`, `Rivelatore.jsx`, `Navbar.jsx`):** moduli dedicati all'interfaccia del sistema. Includono elementi di navigazione persistenti e animazioni fluide (*scroll-reveal*) governate dal componente `Rivelatore`, che gestiscono la comparsa progressiva degli elementi descrittivi per migliorare l'ingaggio dell'utente.

- **Motore core del workflow (`Workflow.jsx`):** il cuore logico e interattivo del sistema, basato sulla libreria `@xyflow/react`. Questo modulo gestisce l'intero spazio di lavoro (canvas) ed è affiancato da pannelli laterali (sidebar) per la configurazione delle proprietà dei blocchi e da appositi bottoni per l'importazione e l'esportazione in formato JSON. La logica di manipolazione dei dati è stata ulteriormente astratta attraverso un hook customizzato e un modulo comprensivo di funzioni di utilità, i quali verranno descritti in maniera più approfondita nella sottosezione successiva.

- **Nodi logici personalizzati (`CustomNodes.jsx`):** un modulo dedicato all'astrazione visiva delle operazioni di flusso. I componenti standard sono stati sovrascritti per creare nodi specializzati (Source, Filter, Map, KeyBy, Window, Aggregate, Union, Sink), ognuno dotato di specifici punti di ancoraggio (*handle*) per regolamentare coerentemente gli input e gli output dei Data Stream.

4.4 Gestione della logica di workflow e utilità

La manipolazione dello stato grafico e la gestione delle regole di validazione del flusso dati sono state disaccoppiate dall'interfaccia utente attraverso un custom hook dedicato chiamato `useWorkflowActions.jsx` e un modulo centrale di funzioni di supporto chiamato `workflowHelpers.jsx`, essenziali per i seguenti aspetti:

- **Gestione del drag-and-drop e identificativi:** il rilascio di nuovi componenti dal pannello laterale sfrutta le API di HTML5 drag-and-drop, convertendo le coordinate del puntatore nello spazio del canvas attraverso la proprietà `screenToFlowPosition`. Per preservare la coerenza e l'unicità del sistema, i blocchi vengono associati a identificativi dinamici progressivi, calcolati automaticamente in base alla tipologia di nodo, garantendo la stabilità dei riferimenti anche a seguito di operazioni di importazione ed esportazione.

- **Configurazione dinamica dei nodi complessi:** l'architettura gestisce in modo reattivo le strutture dati nidificate dei nodi specializzati. Nodi di trasformazione e filtraggio supportano la manipolazione dinamica di elenchi parametrici, mentre gli operatori di aggregazione gestiscono dizionari strutturati associati a funzioni analitiche specifiche. Analogamente, i nodi di elaborazione temporale modulano dinamicamente i propri parametri di configurazione in base alla tipologia di finestra prescelta.

- **Validazione topologica del flusso:** il sistema implementa rigorosi controlli di integrità strutturale volti a prevenire errori prima della generazione del codice di esecuzione:

  - Verifica la presenza obbligatoria di almeno un nodo di origine (Source) e di destinazione (Sink).
  - Regola i flussi in ingresso vincolando il nodo (Union) ad accettare molteplici stream, impedendo al contempo ai nodi intermedi di ricevere ingressi multipli.
  - Rileva e segnala tempestivamente eventuali nodi isolati o rami di elaborazione incompleti.

- **Strategia di importazione e ricostruzione del grafo:** la procedura di importazione effettua il parsing dei file JSON esterni per ricomporre la rete di nodi e collegamenti. Qualora venga caricato un file di produzione privo di coordinate spaziali, l'architettura attiva un algoritmo di posizionamento automatico su una griglia geometrica regolare, preservando la leggibilità iniziale del grafo.

- **Dualismo di esportazione:** come formalizzato nella fase di modellazione concettuale, il sistema traduce il workflow grafico in formato JSON distinguendo tra la modalità Bozza (anche detta Draft, con conservazione dei metadati spaziali) e la modalità di produzione per Apache Flink (anche detta Configurazione Flink, depurata e orientata ai collegamenti).

4.5 Mock-up progettuali e UI

Al fine di validare l'esperienza utente e definire l'ergonomia visiva dell'applicativo prima dello sviluppo definitivo, sono stati realizzati dei modelli grafici preliminari (*mock-up*). L'interfaccia è stata progettata seguendo i principi di chiarezza visiva e immediatezza operativa, riducendo al minimo il carico cognitivo dell'utente durante la costruzione di pipeline di dati complesse. Di seguito vengono illustrate le tre schermate chiave che compongono l'architettura grafica del sistema.

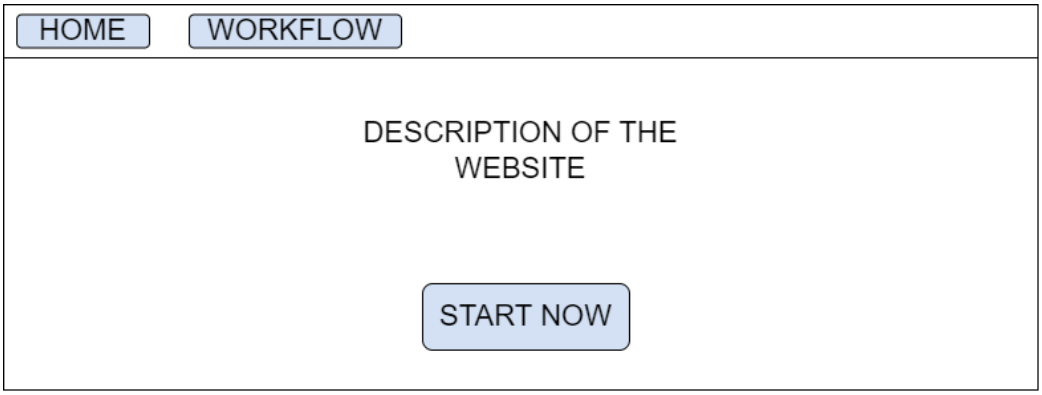


Figura 10: mock-up della home page e della vista di benvenuto.

Il primo mock-up, come mostrato nella Figura 10, rappresenta la home page dell'applicazione. Questa schermata funge da punto d'accesso introduttivo: presenta una barra di navigazione superiore pulita e minimale, un titolo descrittivo incentrato sulle potenzialità di elaborazione dei flussi in tempo reale e un chiaro pulsante di chiamata all'azione (*Call to Action*) per l'avvio immediato dell'editor di workflow.

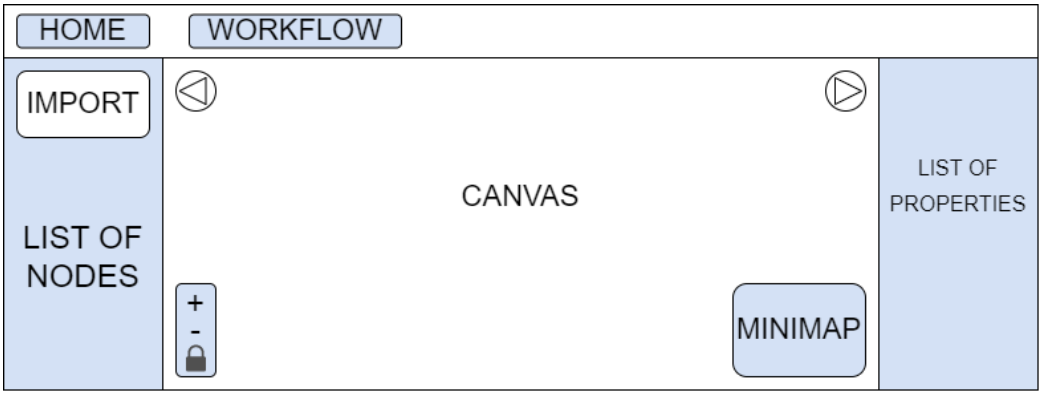


Figura 11: mock-up dell'editor di workflow raffigurante il canvas, le sidebar e i pannelli di supporto.

Il secondo mock-up, mostrato in Figura 11, raffigura il cuore operativo dell'applicazione. La schermata è divisa in tre sezioni logiche integrate:

1. **Sidebar di sinistra:** contiene la libreria dei blocchi trascinabili categorizzati per tipologia e il bottone per l'import da file.

2. **Canvas centrale:** l'area di lavoro interattiva basata su React Flow, dove i nodi vengono posizionati e collegati tramite archi direzionali.

3. **Sidebar di destra:** il pannello dinamico che si adatta al nodo selezionato consentendo la configurazione dei parametri specifici.

Entrambi i pannelli sono inizialmente invisibili e possono essere aperti e chiusi attraverso le apposite frecce. All'interno della sidebar di sinistra, vi è, oltre alla lista dei blocchi da inserire nel canvas, il bottone per l'importazione. Tra gli strumenti di supporto alla creazione del workflow, troviamo la minimappa e il pannello dei controlli per lo zoom e il bloccaggio della visuale. Infine, è bene specificare che la sidebar di destra mostra i parametri generici e specifici

al click del nodo interessato e, per quelli specifici, l'utente deve compilare appositi spazi di input numerici e testuali.

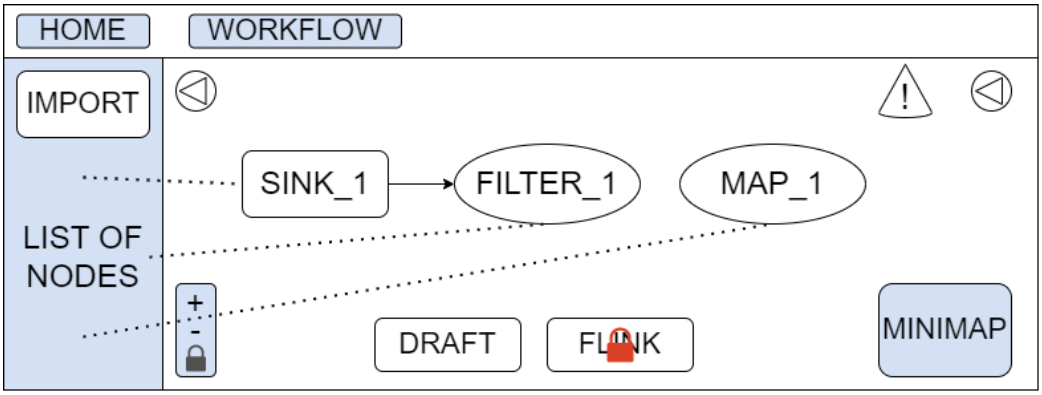


Figura 12: mock-up visivo dei nodi personalizzati e del sistema di validazione.

Il terzo mock-up, come mostrato in Figura 12, si concentra sul dettaglio dei nodi personalizzati e sui feedback visivi di errore. Evidenzia come i blocchi presentino un'interfaccia dedicata all'interno del canvas con punti di ancoraggio ben definiti. Mostra inoltre il meccanismo di notifica visiva in tempo reale, utilizzato per segnalare eventuali violazioni strutturali (ad esempio, un nodo isolato o un flusso sprovvisto di nodi di terminazione), guidando l'utente alla correzione prima della generazione del codice di produzione. Gli errori possono essere visualizzati all'interno di una lista, al clic dell'icona in alto a destra e, nel caso dell'esempio esplicativo mostrato in figura, riguardano la mancanza di un collegamento tra il nodo Filter e il nodo Map e l'assenza di almeno un nodo Sink conclusivo. Infine, viene mostrato come, in presenza di errori, la modalità di esportazione Configurazione Flink non sia disponibile, fino a che essi non vengono risolti.

5 Implementazione

L'implementazione dell'applicativo ha richiesto la traduzione dei requisiti progettuali in un codice modulare, scalabile e manutenibile. Lo sviluppo è avvenuto in ambiente Visual Studio Code adottando React come libreria di base per la costruzione della Single Page Application e integrando librerie specializzate per la gestione dei flussi grafici, tra cui in particolare, React Flow.

5.1 Schema architetturale e dipendenze dei componenti

Per comprendere la struttura del codice sorgente e la suddivisione dei compiti tra i diversi moduli dell'applicazione, la Figura 13 illustra lo schema architetturale di alto livello.

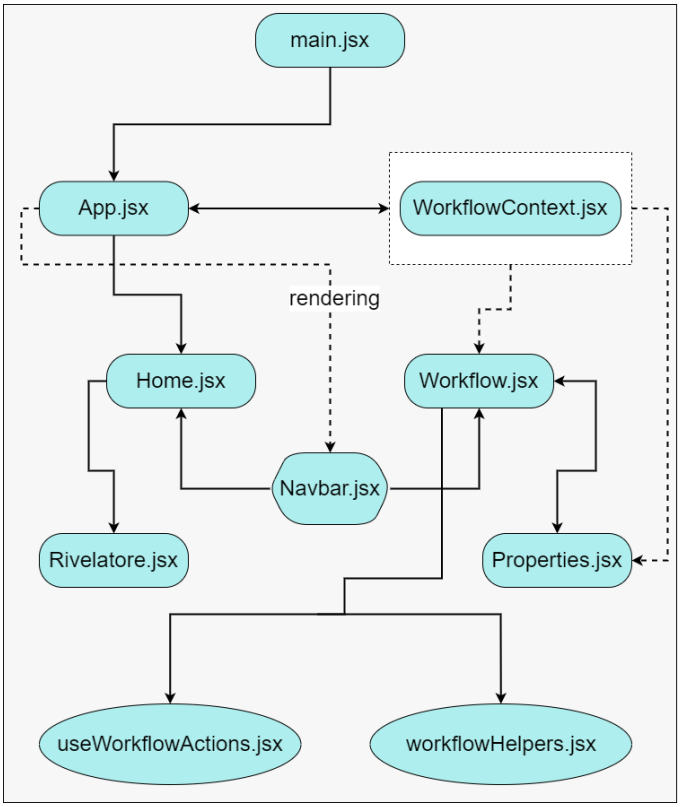


Figura 13: Schema architetturale dei componenti React e dei moduli di supporto. **Legenda:** I rettangoli arrotondati indicano i componenti/pagine React; l'esagono rappresenta il componente strutturale condiviso (`Navbar`); le ellissi indicano i moduli di logica pura e gli hook; il riquadro tratteggiato delimita il contesto di stato globale; le linee continue tracciano il flusso di rendering/chiamata; le linee tratteggiate indicano la sottoscrizione allo stato globale.

L'architettura raffigura ed evidenzia il flusso di esecuzione, che parte dal punto d'ingresso `main.jsx`, attraversa il routing e le viste principali, e si appoggia ai moduli di logica e gestione dello stato globale. Inoltre, come si evince dallo schema, l'applicazione adotta una netta separazione tra:

- **Il livello di root e stato globale:** composto da `main.jsx`, `App.jsx` e dal contesto condiviso `WorkflowContext.jsx`, che preserva lo stato del canvas durante i cambi di schermata.
- **Il livello di interfaccia e navigazione:** costituito dalle viste `Home.jsx` (supportata dal componente visivo `Rivelatore.jsx` per lo scorrimento fluido) e `Workflow.jsx`, entrambe accomunate dalla presenza della `Navbar.jsx` e affiancate dallo schema che definisce le caratteristiche comuni e specifiche dei nodi `Properties.jsx`.
- **Il livello di logica e utilità:** posizionato alla base e costituito dall'hook personalizzato `useWorkflowActions.jsx` e dal modulo di servizio `workflowHelpers.jsx`, richiamati dall'editor per la gestione delle azioni e della validazione topologica.

5.2 Struttura del workspace e organizzazione dei file

Il progetto adotta una suddivisione delle directory pulita e rigorosa, basata sulla separazione delle responsabilità logiche.

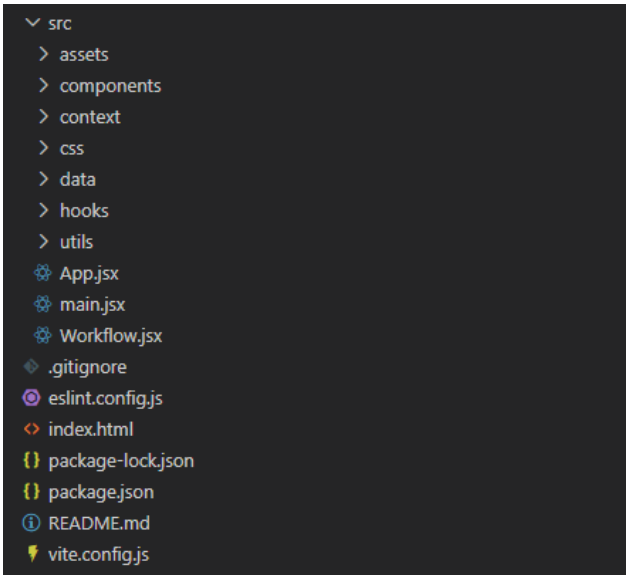


Figura 14: albero delle directory del progetto in Visual Studio Code.

Come mostrato in Figura 14, il codice sorgente all'interno della cartella `src/` è organizzato nei seguenti moduli funzionali:

- `components/`: raccoglie i componenti visivi dell'interfaccia, inclusi i nodi personalizzati (`CustomNodes.jsx`) e i pannelli di navigazione.
- `context/`: contiene il gestore dello stato globale (`WorkflowContext.jsx`), fondamentale per preservare i dati del canvas indipendentemente dal routing.

- `data/`: ospita i file di configurazione statica, tra cui `Properties.jsx`, che definisce lo schema strutturale dei parametri per ogni tipologia di nodo.

- `hooks/`: gestisce la logica di business e le azioni di manipolazione dello stato tramite l'hook personalizzato `useWorkflowActions.jsx`.

- `utils/`: raccoglie le funzioni di supporto nel modulo `workflowHelpers.jsx` dedicate alla validazione topologica, alla gestione dinamica degli ID, al recupero dei campi condizionali in base al tipo di blocco e alle logiche di importazione/esportazione JSON.

Al di fuori della cartella sorgente `src/`, la radice del progetto ospita i file di configurazione e di build fondamentali per l'ecosistema dell'applicazione:

- `package.json` e `package-lock.json`: definiscono i metadati del progetto e gestiscono il versionamento delle dipendenze esterne (`react`, `react-router-dom`, `@xyflow/react`, ecc.), oltre a configurare gli script di automazione per lo sviluppo e la compilazione.

- `vite.config.js`: file di configurazione di Vite, il tool di build e sviluppo rapido utilizzato per compilare e gestire l'esecuzione dell'applicazione.

- `index.html`: la pagina HTML di base che funge da contenitore principale per il montaggio della SPA tramite il nodo radice (id="root").

- `README.md`: il documento di presentazione del repository, contenente le istruzioni per l'installazione delle dipendenze e l'avvio locale del sistema.

5.3 Approccio data-driven e configurazione dei nodi

La configurazione dei parametri associati a ciascun blocco è stata centralizzata nel modulo `Properties.jsx`. Questa scelta architetturale consente di disaccoppiare la definizione dei campi dall'interfaccia grafica del pannello di configurazione laterale.

Il modulo, come mostrato nella Figura 15, mappa puntualmente le proprietà specifiche per ogni operatore di Stream Processing:

```
export const Properties = {
  source: [
    { label: "Type", key: "type", tipo: "fisso", default: "kafka" },
    { label: "Topic", key: "topic", tipo: "testo", placeholder: "Scrivi qui..." },
    { label: "Bootstrap Servers", key: "bootstrap_servers", tipo: "testo", placeholder: "Scrivi qui..." },
    { label: "Timestamp Field", key: "timestamp_field", tipo: "testo", placeholder: "Scrivi qui..."}
  ],
  filter: [
    { label: "Filter Condition", key: "filter_condition", tipo: "testo", placeholder: "Inserisci condizione..." }
  ],
  map: [
    { label: "Applies To Fields", key: "appliesToFields", tipo: "array_stringhe", placeholder: "Scrivi qui..." }
  ],
  keyby: [
    { label: "Applies To Fields", key: "appliesToFields", tipo: "testo", placeholder: "Scrivi qui..." }
  ],
  window: {
    comuni: [
      { label: "Window Type", key: "windowType", tipo: "select", opzioni: ["tumbling", "hopping", "count"], placeholder: "Seleziona tipo..." }
    ],
    specifici: {
      tumbling: [
        { label: "Duration", key: "duration", tipo: "numero", placeholder: "Inserisci numero intero..." },
        { label: "Size", key: "size", tipo: "numero", placeholder: "Inserisci numero intero..." }
      ],
      hopping: [
        { label: "Duration", key: "duration", tipo: "numero", placeholder: "Inserisci numero intero..." },
        { label: "Slide", key: "slide", tipo: "numero", placeholder: "Inserisci numero intero..."}
      ],
      count: [
        { label: "Size", key: "size", tipo: "numero", placeholder: "Inserisci numero intero..." },
        { label: "Slide", key: "slide", tipo: "numero", placeholder: "Inserisci numero intero" }
      ]
    }
  },
  aggregate: [
    { label: "Aggregations", key: "aggregations", tipo: "dizionario_aggregazioni", funzioni: ["min", "max", "avg", "sum"], placeholder: 'Inserisci attributo'
  ],
  union: [
    { label: "Operation", key: "op", tipo: "fisso", default: "UNION ALL" }
  ],
  sink: [
    { label: "Type", key: "type", tipo: "fisso", default: "kafka" },
    { label: "Topic", key: "topic", tipo: "testo", placeholder: "Scrivi qui..." },
    { label: "Bootstrap Servers", key: "bootstrap_servers", tipo: "testo", placeholder: "Scrivi qui..." },
  ]
};
```

Figura 15: schema delle proprietà generiche e specifiche di ciascun blocco.

- **Nodi di i/o** (Source e Sink): configurati con parametri di connessione fissi o testuali (es topic, bootstrapServers, timestampField).
- **Nodi di trasformazione e filtraggio** (Filter, Map e KeyBy): gestiscono condizioni logiche o vettori di campi applicativi (appliesToFields).
- **Nodi temporali** (Window): sfruttano una struttura gerarchica che distingue proprietà comuni (il selettore del tipo di finestra tra tumbling, hopping e count) da parametri specifici di durata e scorrimento (duration, size, slide).
- **Nodi di aggregazione e unione** (Aggregate e Union): manipolano dizionari di funzioni analitiche (min, max, avg, sum) o operazioni logiche di flusso.

Quando l'utente seleziona un nodo nel canvas, l'interfaccia legge dinamicamente questo schema strutturato, renderizzando i campi di input appropriati senza logiche hardcoded nel componente grafico, attraverso la funzione `getCampiFinali` che, come mostrato in Figura 16, consente di estrarre le proprietà dei nodi, in particolare del `window`, strutturato come un oggetto dotato di un form di selezione dedicato.

```
export const getCampiFinali = (nodo, schema) => {
  if (!nodo) return [];
  if (nodo.type === 'window') {
    const tipoScelto = nodo.data.windowType;
    return [
      ...schema.window.comuni,
      ...(schema.window.specifici[tipoScelto] || [])
    ];
  }
  return schema[nodo.type] || [];
};
```

Figura 16: funzione per gestire i dati dei nodi, in particolare del Window.

5.4 Gestione dello stato e del flusso di lavoro

Il coordinamento tra la rappresentazione visiva su React Flow e i dati sottostanti è affidato all'hook `useWorkflowActions.jsx`, che gestisce le mutazioni dello stato globale. Le strutture dati nidificate dei nodi complessi vengono manipolate tramite funzioni apposite:

- **`gestisciArrayStringhe`:** per la gestione dinamica delle liste di campi nei nodi di tipo Map (Figura 17).

```
const gestisciArrayStringhe = (chiave, azione, indice, valore) => {
  SettaBlocchi((nds) => nds.map((nodo) => {
    if (nodo.id === nodoSelezionatoId) {
      const arrayAttuale = Array.isArray(nodo.data[chiave]) ? [...nodo.data[chiave]] : [""];
      if (azione === "MODIFICA") arrayAttuale[indice] = valore;
      if (azione === "AGGIUNGI") arrayAttuale.push("");
      if (azione === "RIMUOVI") arrayAttuale.splice(indice, 1);
      return { ...nodo, data: { ...nodo.data, [chiave]: arrayAttuale } };
    }
    return nodo;
  }));
};
```

Figura 17: funzione che gestisce i dati del nodo Map.

- **`gestisciDizionarioAggregazioni`:** per associare attributi e funzioni di aggregazione nei nodi Aggregate (Figura 18).

```
const gestisciDizionarioAggregazioni = (chiave, azione, indice, campoModificato, valore) => {
  SettaBlocchi((nds) => nds.map((nodo) => {
    if (nodo.id === nodoSelezionatoId) {
      const aggregazioniAttuali = Array.isArray(nodo.data[chiave])
        ? [...nodo.data[chiave]]
        : [{ field: "", op: "min" }];

      if (azione === "MODIFICA") {
        aggregazioniAttuali[indice] = { ...aggregazioniAttuali[indice], [campoModificato]: valore };
      }
      if (azione === "AGGIUNGI") aggregazioniAttuali.push({ field: "", op: "min" });
      if (azione === "RIMUOVI") aggregazioniAttuali.splice(indice, 1);

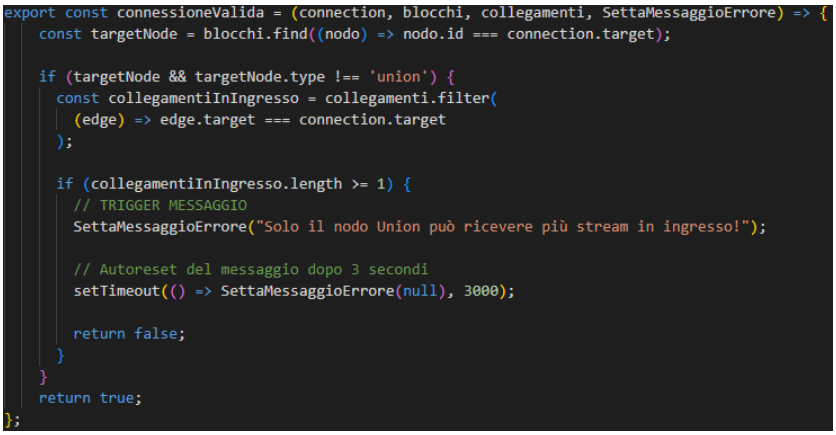
      return { ...nodo, data: { ...nodo.data, [chiave]: aggregazioniAttuali } };
    }
    return nodo;
  }));
};
```

Figura 18: funzione che gestisce i dati del nodo Aggregate.

5.5 Validazione topologica

Per garantire l'integrità e la correttezza dei flussi prima della fase di esportazione, il sistema implementa dei rigorosi controlli attraverso il modulo `workflowHelpers.jsx`, all'interno del quale sono state definite due funzioni che gestiscono dinamicamente gli errori presenti nel workflow:

- `connessioneValida`: per la gestione dei molteplici collegamenti in ingresso consentiti solo al nodo Union (Figura 19).



```
export const connessioneValida = (connection, blocchi, collegamenti, SettaMessaggioErrore) => {
    const targetNode = blocchi.find((nodo) => nodo.id === connection.target);

    if (targetNode && targetNode.type !== 'union') {
      const collegamentiInIngresso = collegamenti.filter(
        (edge) => edge.target === connection.target
      );

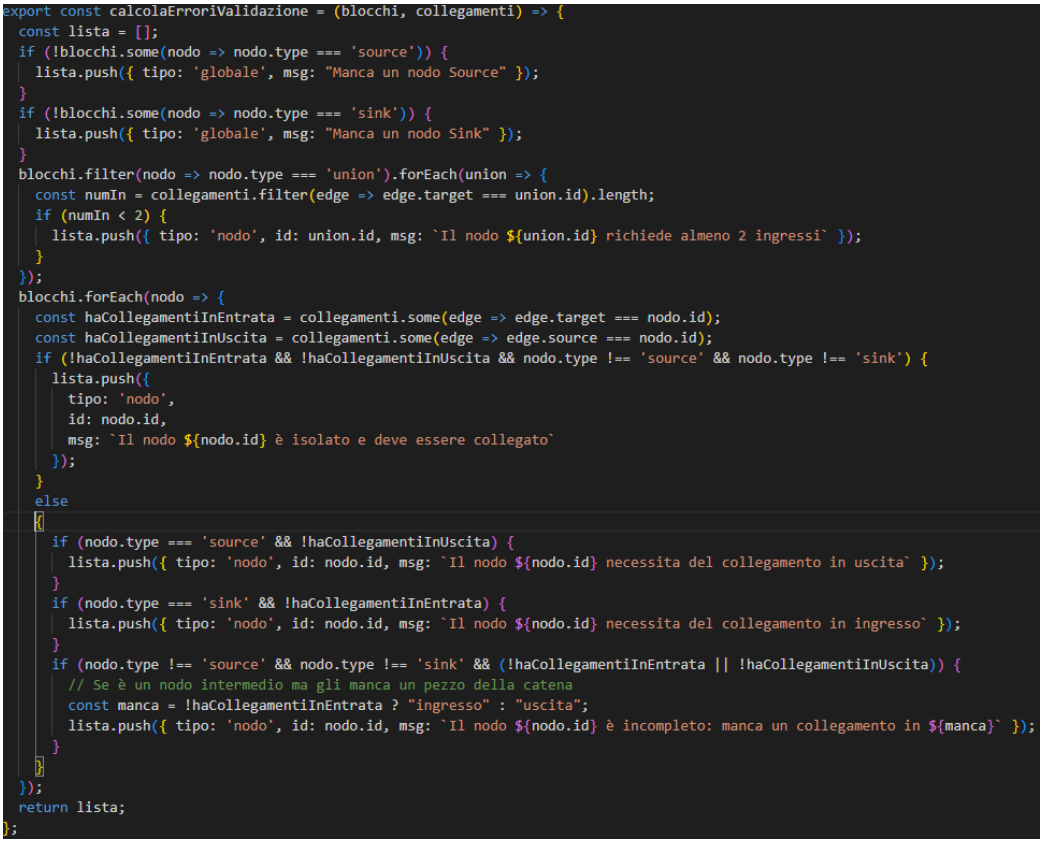
      if (collegamentiInIngresso.length >= 1) {
        // TRIGGER MESSAGGIO
        SettaMessaggioErrore("Solo il nodo Union può ricevere più stream in ingresso!");

        // Autoreset del messaggio dopo 3 secondi
        setTimeout(() => SettaMessaggioErrore(null), 3000);

        return false;
      }
    }
    return true;
};
```

Figura 19: funzione per la validazione dei possibili collegamenti che si effettuano.

- `calcolaErroriValidazione`: per notificare la mancanza di collegamenti o la necessità di inserire nodi essenziali di ingresso (Source) e uscita (Sink) (Figura 20).



```
export const calcolaErroriValidazione = (blocchi, collegamenti) => {
  const lista = [];
  if (!blocchi.some(nodo => nodo.type === 'source')) {
    lista.push({ tipo: 'globale', msg: "Manca un nodo Source" });
  }
  if (!blocchi.some(nodo => nodo.type === 'sink')) {
    lista.push({ tipo: 'globale', msg: "Manca un nodo Sink" });
  }
  blocchi.filter(nodo => nodo.type === 'union').forEach(union => {
    const numIn = collegamenti.filter(edge => edge.target === union.id).length;
    if (numIn < 2) {
      lista.push({ tipo: 'nodo', id: union.id, msg: `Il nodo ${union.id} richiede almeno 2 ingressi` });
    }
  });
  blocchi.forEach(nodo => {
    const haCollegamentiInEntrata = collegamenti.some(edge => edge.target === nodo.id);
    const haCollegamentiInUscita = collegamenti.some(edge => edge.source === nodo.id);
    if (!haCollegamentiInEntrata && !haCollegamentiInUscita && nodo.type !== 'source' && nodo.type !== 'sink') {
      lista.push({
        tipo: 'nodo',
        id: nodo.id,
        msg: `Il nodo ${nodo.id} è isolato e deve essere collegato`
      });
    }
    else
    {
      if (nodo.type === 'source' && !haCollegamentiInUscita) {
        lista.push({ tipo: 'nodo', id: nodo.id, msg: `Il nodo ${nodo.id} necessita del collegamento in uscita` });
      }
      if (nodo.type === 'sink' && !haCollegamentiInEntrata) {
        lista.push({ tipo: 'nodo', id: nodo.id, msg: `Il nodo ${nodo.id} necessita del collegamento in ingresso` });
      }
      if (nodo.type !== 'source' && nodo.type !== 'sink' && (!haCollegamentiInEntrata || !haCollegamentiInUscita)) {
        // Se è un nodo intermedio ma gli manca un pezzo della catena
        const manca = !haCollegamentiInEntrata ? "ingresso" : "uscita";
        lista.push({ tipo: 'nodo', id: nodo.id, msg: `Il nodo ${nodo.id} è incompleto: manca un collegamento in ${manca}` });
      }
    }
  });
  return lista;
};
```

Figura 20: funzione per il calcolo dinamico dei possibili errori presenti nel workflow corrente.

Questa funzione, oltre a mostrare gli errori commessi durante la costruzione del workflow, li modifica dinamicamente. Di conseguenza, nel momento in cui vengono inseriti dei nuovi blocchi o aggiunti dei nuovi collegamenti nel canvas, la lista degli errori si aggiorna in automatico, inserendo i nuovi eventuali errori commessi nella lista. Un discorso del tutto analogo avviene per la loro cancellazione.

5.6 Funzionalità di importazione ed esportazione del workflow

La comunicazione bidirezionale dello stato del workflow verso file esterni in formato JSON rappresenta un pilastro fondamentale per la persistenza dei progetti e la loro successiva esecuzione. Il sistema gestisce l'importazione e l'esportazione attraverso funzioni di supporto dedicate.

Esportazione e validazione del grafo: la procedura di esportazione non si limita alla serializzazione dei dati, ma integra un controllo bloccante basato sulla validazione topologica. Prima di generare il file JSON definitivo, sia esso in formato Draft o orientato ad Apache Flink, l'applicazione verifica l'integrità del grafo. In presenza di errori strutturali, l'export in modalità Configurazione Flink viene inibito attraverso delle proprietà CSS dinamiche. Nella Figura 21 è mostrata la funzione per l'esportazione in codice React, la quale verifica che non siano presenti errori nel caso sia stato cliccato il bottone di esportazione nella modalità orientata a Flink. Se ci sono degli errori, consente l'esportazione solo in modalità Draft e, in tal caso, aggiunge la data dell'avvenimento al JSON. Nel caso non ci siano errori, è consentita l'esportazione in modalità Flink e la funzione va a ripulire il JSON dalle proprietà non necessarie in tale contesto, come le posizioni e la data dell'esportazione, consentendo la generazione di un file pronto per l'elaborazione successiva. Se l'esportazione in modalità Draft avviene con successo, verrà stampato a schermo un messaggio per puntualizzare che un file JSON in modalità Configurazione Flink è sprovvisto di posizioni, sconsigliandone pertanto la reimportazione successiva.

```
export const gestisciEsportazione = (isDraft, blocchi, collegamenti, erroriValidazione, SettaMessaggioErrore) => {
  if (!isDraft && erroriValidazione.length > 0) {
    SettaMessaggioErrore("Impossibile esportare: risolvi prima tutti i problemi nel workflow.");
    setTimeout(() => SettaMessaggioErrore(null), 4000);
    return;
  }
  try {
    const steps = blocchi.map((nodo) => {
      const collegamentiInUscita = collegamenti
        .filter((edge) => edge.source === nodo.id)
        .map((edge) => edge.target);
      const { label, ...datiPuliti } = nodo.data || {};
      let stepDati = {id: nodo.id, type: nodo.type, ...datiPuliti,};
      if (isDraft) {
        stepDati.position = nodo.position;
      }
      if (nodo.type !== 'sink') {
        stepDati.next = collegamentiInUscita;
      } else if (!stepDati.sinkType) {
        stepDati.sinkType = "print";
      }
      return stepDati;
    });
    const workflowFinale = isDraft
      ? { steps, status: 'draft', dataEsportazione: new Date().toISOString() }
      : { steps };
    const dataStr = JSON.stringify(workflowFinale, null, 2);
    const blob = new Blob([dataStr], { type: 'application/json' });
    const url = URL.createObjectURL(blob);
    const link = document.createElement('a');
    link.href = url;
    const prefix = isDraft ? 'BOZZA_workflow' : 'workflow_flink';
    link.download = `${prefix}_${new Date().toISOString().slice(0,10)}.json`;
    document.body.appendChild(link);
    link.click();
    document.body.removeChild(link);
    URL.revokeObjectURL(url);
    if (!isDraft) {
      alert("Esportazione per Flink completata.\n\nNOTA: Questo file è ottimizzato per la produzione e NON contiene le posizioni dei nodi");
    }
  } catch (error) {
    console.error("Errore durante l'esportazione:", error);
  }
};
```

Figura 21: funzione per la gestione dell'esportazione del workflow.

Importazione e parsing strutturale: il processo di importazione permette di caricare un file JSON precedentemente salvato per ripristinare istantaneamente lo stato dell'editor. La funzione di importazione, come mostrato nella Figura 22, esegue il parsing del file, validandone la struttura e ricostruendo l'array dei nodi e dei collegamenti all'interno del contesto globale. Questo consente all'utente di riprendere l'attività di editing o di procedere direttamente alla generazione del flusso di elaborazione. Nello specifico, la funzione inizialmente recupera il file dalla lista dei file del browser e verifica che, oltre ad essere in formato JSON, abbia la proprietà "steps" che definisce i collegamenti successivi tra i blocchi che compongono il workflow. In

seguito, esegue un controllo sulla presenza o meno delle posizioni dei blocchi; se sono presenti, allora verrà ricostruito un workflow con blocchi e relativi collegamenti, identico a quello esportato coincidente con il file e nella stessa posizione. In caso le posizioni non fossero presenti, la funzione utilizza un algoritmo di emergenza che ricostruisce il workflow attraverso una struttura a griglia.

```
export const gestisciImportazione = (event, SettaBlocchi, SettaCollegamenti, SettaMessaggioErrore) => {
  const file = event.target.files[0];
  if (!file) return;
  const reader = new FileReader();
  reader.onload = (event) => {
    try {
      const json = JSON.parse(event.target.result);
      if (!json.steps) throw new Error("Formato non valido");
      const nuoviNodi = json.steps.map((step, indice) => {
        // Algoritmo di emergenza: Griglia 3xN se manca la posizione
        const xDefault = (indice % 3) * 300;
        const yDefault = Math.floor(indice / 3) * 200;
        return {
          id: step.id,
          type: step.type,
          position: step.position || { x: xDefault, y: yDefault },
          data: {
            label: step.id.toUpperCase(),
            ...Object.keys(step)
              .filter(key => !['id', 'type', 'position', 'next'].includes(key))
              .reduce((obj, key) => ({ ...obj, [key]: step[key] }), {})
          },
        };
      });
      const nuoviEdges = [];
      json.steps.forEach((step) => {
        if (step.next && Array.isArray(step.next)) {
          step.next.forEach((targetId) => {
            nuoviEdges.push({
              id: `e-${step.id}-${targetId}`,
              source: step.id,
              target: targetId,
              animated: false,
              style: { strokeWidth: 2 }
            });
          });
        }
      });
      SettaBlocchi(nuoviNodi);
      SettaCollegamenti(nuoviEdges);
      event.target.value = '';
    } catch (err) {
      SettaMessaggioErrore("Errore nell'importazione del file JSON.");
      setTimeout(() => SettaMessaggioErrore(null), 3000);  }
  };
  reader.readAsText(file);
};
```

Figura 22: funzione per l'importazione del workflow e conseguente ricostruzione.

5.7 Problemi riscontrati e soluzioni adottate

Durante lo sviluppo dell'applicativo, l'evoluzione dei requisiti e la fase di testing hanno fatto emergere alcune criticità architetturali e funzionali. Di seguito vengono descritti i problemi principali riscontrati e le soluzioni implementate per risolverli.

5.7.1 Persistenza dello stato al cambio pagina

Il problema: nelle prime fasi di sviluppo, l'architettura si basava su un routing di base tramite *React Router*. Ciascuna vista viveva in componenti isolati senza un layer di stato condiviso. Di conseguenza, ogni qualvolta l'utente navigava temporaneamente fuori dall'editor (ad esempio tornando alla home) e poi rientrava, l'intero canvas di React Flow veniva rismontato e l'intero stato dei blocchi e dei collegamenti andava irrimediabilmente perduto.

La soluzione: per ovviare a questo problema, è stato introdotto un livello di gestione dello stato globale basato sulla context API di React (`WorkflowContext.jsx`). Spostando l'array dei blocchi, dei collegamenti e le funzioni di aggiornamento in un contesto condiviso a livello di root (`App.jsx`), lo stato del workflow viene preservato in memoria anche durante i cambi di pagina, garantendo un'esperienza utente fluida e senza interruzioni.

5.7.2 Gestione dinamica degli identificativi e prevenzione dei conflitti

Il problema: inizialmente, la generazione degli ID dei nuovi nodi rilasciati sul canvas era affidata a un contatore statico basato unicamente sulla lunghezza dell'array dei blocchi. Questo approccio generava gravi bug in due scenari specifici:

1. Dopo aver eliminato alcuni nodi intermedi, l'indice progressivo si "sovrapponeva" ai nodi esistenti, creando ID duplicati.

2. A seguito di un'operazione di importazione da file JSON esterno, l'inserimento di un nuovo nodo causava collisioni immediate con gli identificativi appena importati.

La soluzione: la funzione statica è stata completamente riprogettata introducendo un algoritmo di calcolo dinamico basato sull'analisi del valore numerico massimo già presente per ciascuna tipologia di nodo. Come mostrato nella Figura 23, l'algoritmo filtra i nodi per tipologia, estrae la componente numerica tramite parsing della stringa dell'id, individua il valore massimo esistente e restituisce una stringa incrementata in modo sicuro, azzerando qualsiasi rischio di collisione sia in fase di editing libero che dopo l'importazione di file esterni.

```
export const calcolaNuovoId = (tipo, blocchi) => {
  const massimoId = blocchi
    .filter((nodo) => nodo.type === tipo)
    .reduce((max, nodo) => {
      const parti = nodo.id.split('_');
      const numero = parseInt(parti[parti.length - 1]);
      return !isNaN(numero) && numero > max ? numero : max;
    }, 0);
  return `${tipo}_${massimoId + 1}`;
};
```

Figura 23: algoritmo per la generazione dinamica e sicura degli identificativi dei nodi.

5.7.3 Ottimizzazione delle prestazioni

Il problema: man mano che la complessità del workflow cresce e il numero di nodi e collegamenti sul canvas aumenta, le funzioni di controllo topologico e di validazione richiedono una maggiore potenza computazionale. Eseguire cicli di scansione sull'intero grafo ad ogni singolo aggiornamento o movimento del mouse comporterebbe inevitabili rallentamenti dell'interfaccia utente.

La soluzione: per scongiurare colli di bottiglia prestazionali, all'interno del modulo principale `Workflow.jsx` sono stati impiegati gli hook nativi di ottimizzazione di React, in particolare `useMemo` e `useCallback`:

```
const erroriValidazione = useMemo(() =>
  calcolaErroriValidazione(blocchi, collegamenti),
[blocchi, collegamenti]);


const onConnectValid = useCallback((conn) =>
  connessioneValida(conn, blocchi, collegamenti, SettaMessaggioErrore),
[blocchi, collegamenti]);
```

Figura 24: hook nativi di supporto per la gestione delle prestazioni.

Come illustrato nella Figura 24, l'utilizzo di `useMemo` consente di memorizzare il risultato della validazione, imponendo a React di rieseguire la funzione di calcolo degli errori solamente nel momento in cui si verificano variazioni effettive nell'array dei blocchi o dei collegamenti. Analogamente, `useCallback` stabilizza il riferimento alla funzione di controllo dei collegamenti, evitando la ricreazione superflua della funzione ad ogni ciclo di rendering e garantendo un'interagibilità fluida sul canvas anche in presenza di workflow strutturati ed estesi.

5.8 Risultati finali

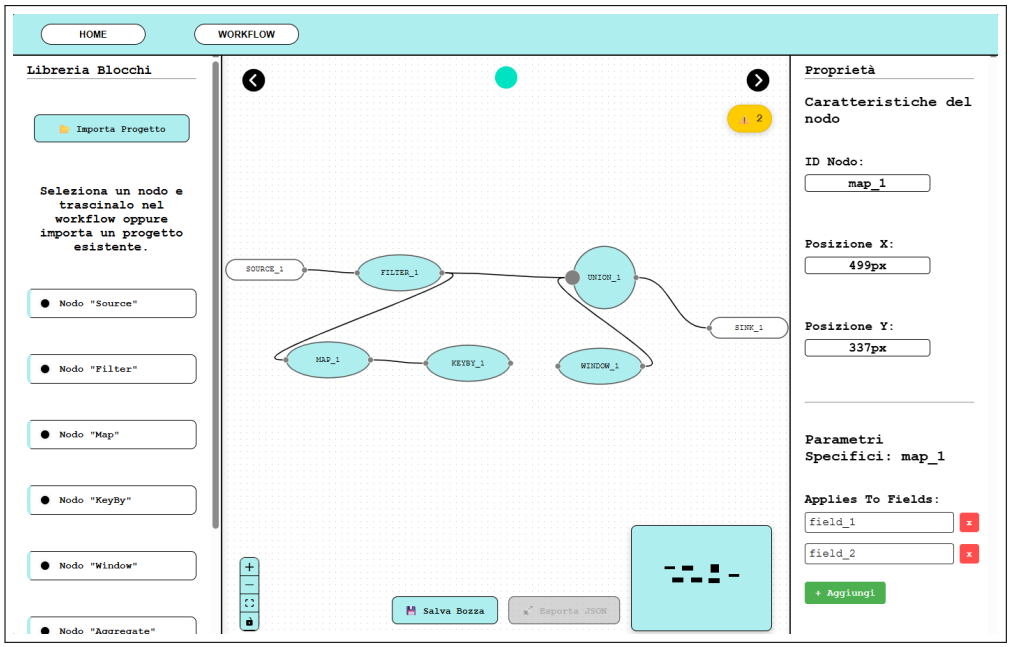


Figura 25: Panoramica complessiva dell'interfaccia finale con workflow attivo e pannelli di controllo aperti.

La schermata finale dell'applicativo, in Figura 25 mostra l'editor di workflow pienamente operativo in tutte le sue componenti integrate. Al centro, il canvas interattivo ospita una pipeline di elaborazione completa, composta da un nodo sorgente, nodi intermedi e nodo conclusivo, connessi tra loro. Sulla sinistra, la libreria dei blocchi consente l'accesso rapido ai componenti trascinabili, mentre sulla destra il pannello di configurazione mostra i campi parametrici attivi relativi all'operatore selezionato nel grafo. In alto, oltre alla navbar utile per il routing con la pagina home, vi è la lista degli errori presenti nel canvas e in basso troviamo i bottoni di export e i pannelli di supporto.

5.8.1 Gestione in tempo reale degli errori topologici

Per garantire la robustezza del sistema e prevenire configurazioni non valide, l'interfaccia integra un meccanismo di controllo dinamico degli errori. Nel caso in cui il diagramma presenti violazioni strutturali — come un nodo intermedio disconnesso o l'assenza di un blocco di chiusura (Sink) — il sistema notifica tempestivamente l'anomalia.

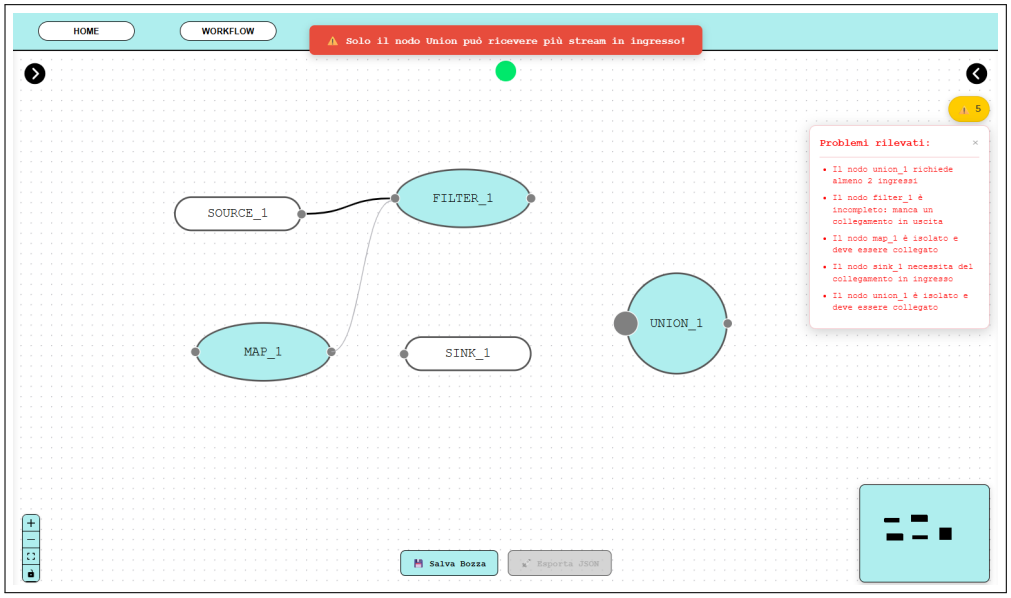


Figura 26: Esempio di rilevamento dinamico degli errori e inibizione dell'esportazione per Flink.

Come illustrato nella Figura 26, la lista degli errori in alto si aggiorna in tempo reale ad ogni modifica del grafo. Tale meccanismo guida l'utente nella correzione e inibisce automaticamente l'esportazione verso la configurazione di produzione (Flink) fintanto che permangono errori bloccanti.

5.8.2 Procedura di esportazione e notifiche di sistema

Il secondo scenario operativo riguarda la generazione del file JSON di configurazione per Apache Flink e i relativi feedback visivi forniti all'utente al momento dell'esportazione.

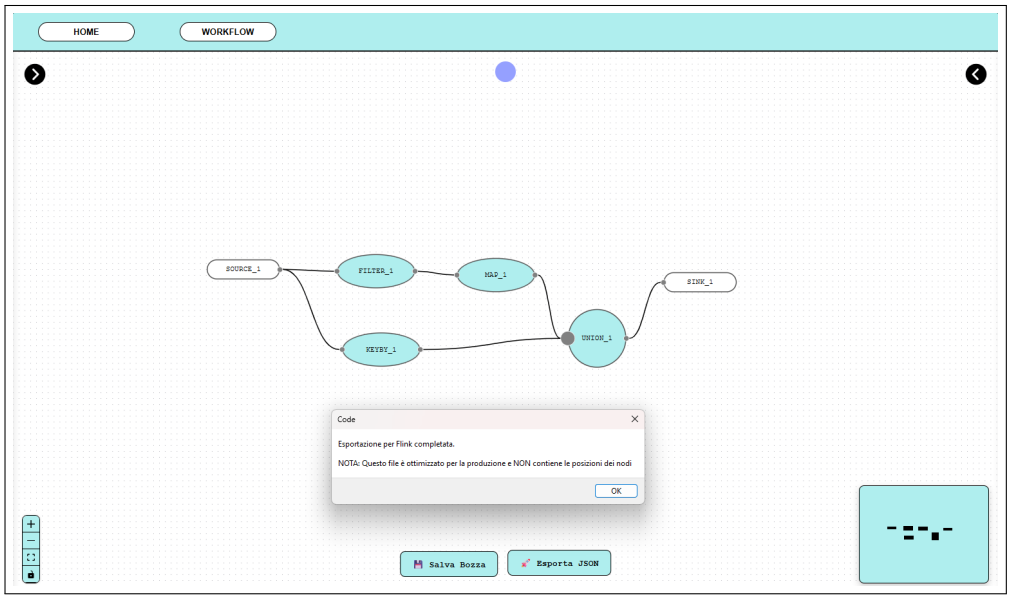


Figura 27: Esportazione del workflow in modalità Flink con pulsante attivo e messaggio di avviso associato.

Come mostrato nella Figura 27, in assenza di errori topologici il bottone di esportazione in modalità Flink risulta correttamente abilitato. A seguito del click, il sistema genera la configurazione pulita da metadati spaziali e mostra contestualmente un messaggio di notifica a schermo. Tale avviso informa l'utente che il file esportato è ottimizzato per l'esecuzione diretta sul motore di stream processing ed è sprovvisto delle coordinate geometriche di layout, sconsigliandone pertanto la successiva reimportazione ai fini di editing grafico.

6 Conclusioni e sviluppi futuri

Il presente elaborato ha affrontato la progettazione e lo sviluppo di un editor web interattivo basato su React e React Flow, concepito per semplificare e formalizzare la creazione di pipeline di elaborazione dati all'interno del progetto HOMEY. Partendo dalla complessità e dalle alte prestazioni dei paradigmi di Stream Processing offerti da Apache Flink, il lavoro ha colmato il divario tra la programmazione di basso livello e l'esigenza di un'interfaccia utente visiva, intuitiva e rigorosa, inserendosi perfettamente nel contesto dello Stream Postprocessing Service.

6.1 Bilancio dei risultati raggiunti

Durante il percorso di sviluppo, sono stati conseguiti gli obiettivi prefissati in fase di analisi e progettazione:

- **Modellazione e usabilità:** sono stati definiti e implementati otto operatori fondamentali dotati di un sistema di configurazione dinamica dei parametri tramite un approccio data-driven.
- **Integrità topologica:** l'introduzione di rigorosi controlli di validazione in tempo reale ha permesso di prevenire errori strutturali prima della generazione del codice, guidando l'utente attraverso feedback visivi immediati.
- **Interoperabilità ed esportazione:** il sistema garantisce una flessibilità d'uso unica grazie al dualismo di esportazione: la modalità Draft, ideale per la persistenza dei metadati grafici e spaziali, e la modalità Configurazione Flink, depurata e pronta per l'esecuzione sul motore di Stream Processing.
- **Robustezza architetturale:** l'adozione di un State Management centralizzato (Context API), di hook personalizzati e di strategie mirate all'ottimizzazione delle prestazioni (`useMemo` e `useCallback`) ha assicurato stabilità, scalabilità e una fluidità d'interazione costante anche in presenza di workflow complessi.

6.2 Sviluppi futuri

Nonostante l'applicativo soddisfi pienamente i requisiti stabiliti e si presenti come uno strumento solido e operativo, l'architettura modulare adottata lascia spazio a diverse evoluzioni e miglioramenti futuri, tra cui:

- **Collaborazione in tempo reale:** l'introduzione di meccanismi di editing collaborativo multi-utente (simili a strumenti come Google Docs o Figma), sfruttando tecnologie WebSocket per permettere a più sviluppatori di lavorare contemporaneamente sullo stesso grafo.
- **Test di esecuzione e simulazione (*Dry-Run*):** l'integrazione di un ambiente di simulazione client-side che permetta di visualizzare un'anteprima del flusso dei dati e dei campioni in transito direttamente sui nodi del canvas, ancor prima di esportare il file di configurazione finale.
- **Estensione degli operatori:** l'aggiunta di ulteriori operatori avanzati nativi di Flink, come finestre temporali personalizzate basate su eventi complessi (*Complex Event Processing* - CEP) o join tra flussi multipli più articolati.
- **Generazione automatica di codice:** l'ampliamento delle funzionalità di export per tradurre il grafo JSON non solo in configurazioni per lo Stream Postprocessing Service, ma direttamente in codice sorgente (es. Java o Scala) pronto per essere compilato ed eseguito su un cluster Apache Flink.
- **Generazione automatica da testo (*Text-to-Workflow* tramite IA):** un'evoluzione di grande rilievo strategico consiste nell'integrazione di modelli di intelligenza artificiale generativa. Sfruttando interfacce

basate su linguaggio naturale, l'utente potrebbe descrivere testualmente la logica della pipeline desiderata (es. "Crea un flusso che legge da Kafka, filtra i valori anomali di temperatura, raggruppali per finestra temporale e salva il risultato su database") e consentire all'editor di interpretare la richiesta, posizionare automaticamente i nodi sul canvas, collegarli e configurare i parametri in modo del tutto autonomo.

Riferimenti bibliografici

[1] M. M. Sciarroni, M. Esposito, P. Pierleoni, and E. Storti, "Monitoring data streams in industry 5.0: a knowledge graph approach," in *2024 IEEE 8th Forum on Research and Technologies for Society and Industry Innovation (RTSI)*, Milano, Italy, 2024, pp. 566–571.

[2] European Commission, "Industry 5.0: Towards a sustainable, human-centric and resilient european industry," Publications Office of the European Union, Policy Brief, 2021.

[3] Apache Software Foundation, "Apache Kafka documentation," https://kafka.apache.org/documentation/, 2026, accessed: 2026-08-07.

[4] Apache Software Foundation, "Apache Flink: Architecture and stateful stream processing," https://flink.apache.org/what-is-flink/flink-architecture/, 2026, accessed: 2026-08-07.

[5] Microsoft. (2026) Visual studio code documentation. Documentazione ufficiale di Visual Studio Code. [Online]. Available: https://code.visualstudio.com/docs

[6] Meta Open Source. (2026) React – a javascript library for building user interfaces. Documentazione ufficiale di React. [Online]. Available: https://react.dev/

[7] xyflow GmbH. (2026) React flow – a customizable react component for building node-based uis and interactive diagrams. Documentazione ufficiale di React Flow. [Online]. Available: https://reactflow.dev/

[8] Mozilla Developer Network. (2026) Html: Hypertext markup language. Documentazione ufficiale di riferimento per HTML. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML

[9] Mozilla Developer Network. (2026) Javascript – dynamic client-side scripting. Documentazione ufficiale di riferimento per JavaScript. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[10] Mozilla Developer Network. (2026) Css: Cascading style sheets. Documentazione ufficiale di riferimento per CSS. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/CSS

[11] GitHub, Inc. (2026) Github documentation – getting started with github. Documentazione ufficiale di riferimento per la piattaforma GitHub. [Online]. Available: https://docs.github.com/