UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

INGINERIE SOFTWARE

Lucrare de disertație

PLATFORMĂ ONLINE, DISTRIBUITĂ, PENTRU DATA ANALYSIS

Absolvent
Nicoleta Ciaușu

Coordonator științific
Conf. dr. Radu Gramatovici

București, septembrie 2024

Rezumat

Lucrarea prezintă un proof-of-concept pentru o platformă online, distribuită, ce permite crearea de fluxuri de analiză de date. În cadrul platformei, utilizatorii pot selecta dintr-o colecție de funcții care fac parte dintr-un repository comun, le pot introduce în suprafața de lucru, și le pot conecta între ele la nivel de input și de output, obținându-se astfel un graf de execuție, o reprezentare vizuală a pașilor prin care un set de date este transformat secvențial, pentru a ajunge la un format procesat, util. În urma configurării fluxului de date de către utilizator, platforma oferă posibilitatea de "a da deploy" acestui graf, transformând-ul dintr-o simplă reprezentare vizuală, într-o rețea de containere Docker ce comunică între ele folosind broker-ul de mesaje RabbitMQ ca intermediar, utilizând cozi de mesaje create dinamic. După pasul de deployment, utilizatorul poate reveni pentru a încărca seturi de date în format CSV în graf și primi rezultatele procesării spre vizualizare.

Astfel, obținem o platformă tip Infrastructure-as-a-Service, în care lucrătorii din domeniul analizei datelor pot să se concentreze pe munca specifică domeniului lor, iar platforma să le ofere posibilități de scalare fără efort adițional. În plus, dorim să susținem reutilizarea codului în domeniu și de-a lungul diferitelor proiecte de Data Science, prin intermediul unui repository public de funcții.

Abstract

The paper presents a proof-of-concept for a distributed online platform that allows the creation of data analysis workflows. Within the platform, users can select from a collection of functions that are part of a shared repository, drag them onto the workspace, and connect them at the input and output level, creating an execution graph. This graph serves as a visual representation of the steps through which a data set is sequentially transformed to arrive at a processed, useful format. After configuring the data flow, the platform allows users to deploy this graph, turning it from a mere visual representation into a network of Docker containers that communicate with each other using RabbitMQ dynamically-allocated message queues as an intermediary. After the deployment step, the user can upload CSV datasets into the graph and receive the processed results for visualization.

Thus, we create an Infrastructure-as-a-Service platform where data analysis professionals can focus on their specific domain work, while the platform provides support for scaling without additional effort. Additionally, we aim to support code reuse and collaboration in the field, through a shared function repository.

Cuprins

Capitolul 1

Introducere

1.1 Motivație

Cantitatea de informație pe care suntem capabili să o generăm, procesăm, și interpretăm a crescut extrem de mult în ultimii 10 ani [59]. Acest fapt a crescut cererea de analiști de date, specialiști capabili să aplice tehnici de procesare și interpretare a datelor pentru a extrage informații [3], care activează în domeniul cunoscut drept "Data Science". În această eră în care informația reprezintă putere, munca acestor specialiști, cât și rezultatele lor, stau la baza multora dintre inovațiile recente, de la noile descoperiri în inteligența artificială, la motivarea deciziilor de business, pentru organizații ce au activități la scară globală.

Ne-am dorit să identificăm o nevoie în piață care să poată fi transformată într-un produs, care să ofere schimbare reală, îmbunătățind procesele dintr-un domeniu de mare actualitate și importanță, și deci, contribuind spre un viitor mai bun. Dată fiind importanța domeniului menționat, orice îmbunătățire a proceselor care îl compun are potențialul de a se propaga și avea influență mare asupra viitorului. Din acest motiv, am ales ca pentru teza noastră, grupul țintă al produsului să fie specialiștii care activează în Data Science.

Pentru a îndeplini acest lucru, am început prin a explora procesele parcurse de către analiștii de date, cu scopul de a înțelege atât activitatea lor, cât și eventualele probleme ("pain points") pe care le au, și ar putea fi îmbunătățite. În urma identificării acestor neajunsuri, am pus bazele proiectului nostru, întâi din perspectivă de produs, apoi din perspectivă arhitecturală, completând soluția prin implementare și testare. Rezultatul a fost crearea unui proof-of-concept pentru o platformă online, scalabilă, care permite analiștilor de date să facă procesare de date în cloud, beneficiind de pe urma avantajelor aduse de scalarea într-un mediu distribuit, fără să fie nevoie să învețe procesele specifice de Ops[16].

1.2 Structura lucrării

Am început lucrarea prin explorarea domeniului țintă, înțelegând procesele, cât și dificultățile prezente în mod curent. Am explorat opțiunile prezente în piață pentru a înțelege avantajele și dezavantajele lor. Am identificat un set de trăsături cheie pe care platforma trebuie să le aibă pentru a avea succes, pentru a crea o imagine cât mai completă asupra proiectului. Am stabilit, din perspectivă de produs, care este misiunea aplicației, ce concepte introduce, și ce funcționalități ar trebui să ofere. Am făcut un plan arhitectural, alegând o suită de tehnologii potrivită, decizându-ne asupra unei arhitecturi bazate pe microservicii, și planificând, la nivel macro, interacțiunea dintre componentele aplicației. Am continuat cu etapă de implementare, în care am trecut prin procesul de dezvoltare a aplicației. În final, am concluzionat lucrarea, uitându-ne în retrospectivă și notând direcții de dezvoltare ulterioară.

Capitolul 2

Analiza domeniului și a contextului de business

2.1 Despre domeniu

Aplicația propusă dorește să se integreze în ecosistemul platformelor și uneltelor folosite în Data Science. Am ales să țintim acest domeniu deoarece el a cunoscut o evoluție rapidă în ultimii ani, devenind esențial în numeroase industrii moderne. Data Science implică o abordare multidisciplinară, ce combina elemente din matematică, statistică, programare, inteligentă artificială și învățare automată, rezultând în extragerea de informații valoroase din datele puse la dispoziție, colectate de către companii. [33] Aceste informații stau la baza procesului de decizie în cadrul firmelor. [17] Dată fiind dificultatea proceselor din domeniu, cât și valoarea informațiilor obținute, domeniul Data Science se pretează bine pentru căutarea de optimizări sau îmbunătățiri de procese, ce își pot demonstra ușor valoarea în piață.

2.1.1 Publicul țintă

Cei care activează în domeniu, numiți Data Scientists, sunt profesioniști ce poseda expertiză tehnică avansată, și sunt implicați în întreg procesul de colectare, explorare și interpretare a datelor. Activitatea lor poate fi descrisă în mod facil drept un ciclu iterativ cunoscut în literatura de specialitate sub denumirea de “Bucla Analizei de Date” (Data Analysis Loop) [26]. Aceasta buclă se conturează în etape etape precum identificarea unei nevoi, colectarea datelor, curățarea datelor, analiza exploratorie, interpretarea prin scrierea unui raport si generarea de vizualizari, și, în final, interpretarea datelor, ce duce la noi nevoi identificate.[26] Fiecare dintre aceste etape are durată variabilă și are loc în medii diferite, folosind tehnologii diferite.

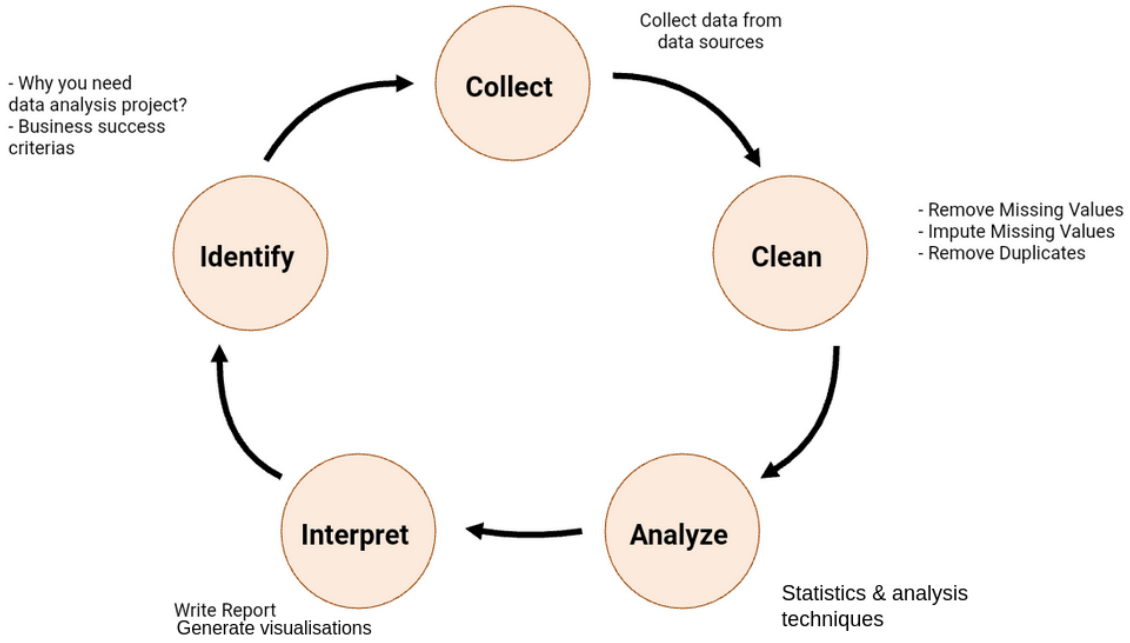

Figura 2.1: Bucla analizei de date ilustrată [26]

2.2 Starea curentă a pieței

În prezent, ecosistemul de Data Science oferă o gamă largă de platforme specifice, fiecare având atât punctele sale forte, cât și dezavantaje. Câteva dintre platformele cele mai populare folosite în cadrul analizei și explorării de date sunt platforme de tip IDE, precum Jupyter Notebook[45], RStudio[44], respectiv DataLab[15]. Acestea sunt platforme de dezvoltare, folosite de specialiști pentru a scripta, folosind limbaje de programare precum R sau Python, cât și bibliotecile asociate ecosistemelor, seria de operații ce trebuie aplicată unui dataset pentru a obține informațiile dorite. Deși utile în etapa exploratoare, aceste unelte au dezavantajul dificultății în scalabilitate. [49]

Acest neajuns a generat un al doilea val de unelte, concentrate pe soluții tip cloud, cel mai proeminent exemplu în acest sens fiind Apache Spark[**spark**], un motor open-source ce permite analiza volumelor mari de date într-un mediu distribuit. Acest motor este în prezent cel mai folosit motor pentru analiza datelor la scară largă. Totuși, dificultatea în utilizare a acestei tehnologii poate fi considerabilă,[1] mai ales pentru echipele care nu au experiență cu gestionarea infrastructurilor distribuite. Deși Apache Spark oferă o viteză mare de procesare, cât și flexibilitate, implementarea și întreținerea unui cluster Spark este o activitate complexă, pentru care este nevoie de pregătire specială, și de familiarizare cu platforma și mediul de lucru.

2.2.1 Dificultăți întâlnite

Procesul de analiză a datelor prezintă o serie de provocări comune, cunoscute în literatura ca și "pain points", pentru data scientists. Printre acestea, se numără:

Pre-procesarea datelor

Gestionarea unor volume mari de date variate poate fi extrem de dificilă, în special atunci când sunt nestructurate sau provin din surse multiple. [67] Procesul de curățare a datelor este esențial pentru a putea produce rezultate de calitate. Acest pas de pre-procesare, deși critic, este considerat neplăcut de către cei ce activează în domeniu. În plus, pre-procesarea datelor tinde să consume mult timp și duce adeseori întârzieri în livrarea rezultatelor.

Comunicarea inter-echipe

Una dintre cele mai frecvente dificultăți întâlnite în companiile care folosesc procese de știința datelor este comunicarea deficitară între echipele de data scientists și alte echipe, cum ar fi dezvoltatorii de software sau stakeholder-ii din zona de business.[37] Aceste trei grupuri de interese provin din background-uri diferite, au seturi de abilitați diferite, și prioritizează lucrurile diferit. Totuși, colaborarea dintre aceste grupuri este esențială oricărei organizații care își dorește să aibă succes. Așadar, orice inițiativă care facilitează comunicarea productivă între aceste grupuri este dezirabilă.

Greutăți în reutilizarea codului

De obicei, în Data Science, reutilizarea codului între diferite proiecte este adesea dificilă, din cauza specificităților fiecărui proiect și a lipsei unei standardizări. Stilul curent, cel mai popular, bazat pe scripting, nu facilitează reutilizarea.[9] Eliminarea acestui efort duplicat poate duce atât la o experiență mai bună pentru dezvoltatori, cât și la productivitate crescută.

Scalabiliate

O problemă critică întâmpinată de mulți data scientists este capacitatea de scalare a soluțiilor lor.[10] Stilul curent, bazat pe notebooks, nu se pretează deloc pentru lucrul cu seturi de date mari, sau pentru consolidarea într-un mediu de producție.[9] Acest lucru devine și mai dificil în cazul lucrului cu modele de învățare automată, unde apar provocări precum menținerea performanței pe termen lung, respectiv gestionarea modificărilor aduse datelor și modelelor.

2.2.2 Trăsături cheie identificate

Analizând soluțiile populare prezente în mod curent pe piață, am identificat următoarele neajunsuri, pe care dorim să le adresăm în platforma noastră: dificultatea în scalare, dificultatea în comunicarea interdisciplinară, cât și capacitatea de reutilizare scăzută a codului.

Astfel, prin aplicația propusă în aceasta lucrare ne dorim să oferim un mediu de lucru ușor de utilizat, scalabil, având reutilizarea codului în minte, care să permită atât experților în știința datelor, cât și utilizatorilor non-tehnici să colaboreze eficient. Ne propunem sa îndeplinim acest lucru prin crearea unei platforme low-code[8] în care utilizatorii își pot crea, prin intermediul unei interfețe grafice, fluxuri de procesare a datelor care rulează în cloud. Centrarea ideii de dezvoltare în jurul funcțiilor inter-conectabile permite dezvoltarea rapidă, prin reducerea numărului de linii de cod ce trebuiesc scrise. Astfel, sporește productivitatea analiștilor, care vor putea ca urmare să se concentreze mai mult pe particularitățile specifice problemelor pe care încearcă să le rezolve.

Un alt aspect cheie este scalabilitatea. Este important ca sistemul propus de platformă să fie gândit astfel încât scalabilitatea să vină natural, încă din etapa de design, pentru a permite utilizatorilor să gestioneze seturi mari de date și să implementeze modele complexe fără a fi restricționați de resursele hardware disponibile local.

2.2.3 Concluzii

În urma analizei efectuate anterior, conturăm următoarele premize. Vedem în acest domeniu o posibilă oportunitate de business în a oferi o platforma online care să unifice întreg procesul dat de buclă de analiză de date, efectuat actualmente într-o manieră nestandardizată, în moduri și cu unelte diverse, neintegrate. Propunem un mediu care să reducă context switching-ul, să crească capacitatea de reutilizare a codului, și să creeze un pattern în felul în care sunt scrise procesatoarele de date, care să permită scalarea cu ușurință. Mai mult decât atât, dorim să construim o infrastructura tip cloud în jurul acestei metodologii, și să automatizam procesul de deployment, astfel încât utilizatorii platformei, data scientists, să beneficieze de puterea de procesare sporită, fără să fie nevoie să învețe tehnici de programare într-un mediu distribuit, sau să colaboreze în mod direct cu inginerii software.

Dezvoltarea din perspectivă de produs a platformei noastre, cu care vom continua, pleacă de la aceste observații finale.

Capitolul 3

Perspectiva de produs

3.1 Misiunea aplicației

În urma analizei de piață efectuată anterior, pentru a adresa nevoile identificate mai sus, stabilim că scopul final al produsului este de a ușura procesul de analiză de date pentru cei ce lucrează în data analysis, respectiv data science. Produsul își vă îndeplini misiunea astfel:

- Îmbunătățind colaborarea dintre analiști prin intermediul folosirii unei platforme online,
- Oferind analiștilor de date posibilitatea de a accesa beneficiile de rulare ale soluțiilor lor într-un mediu distribuit, în același timp coborând gradul de dificultate prin abstractizarea procesului de deployment în spatele platformei low-code,
- Oferind analiștilor o platformă unică prin care pot îndeplini toți pașii din bucla analizei de date,
- Sporind gradului de reutilizare a codului și reducând nevoia de a scrie funcții pentru funcționalități uzuale prin intermediul bibliotecii de funcții.

3.2 Propunerea noastră

Propunem o platformă online, scalabila, ce permite utilizatorilor să creeze în browser fluxuri de date, denumite și data pipelines[57], înlănțuiri de funcții care au ca scop aducerea unui set de date dintr-un format neprocesat A într-un format final B. În lanțul de funcții pot apărea felurite operații, de modificare, agregare, filtrare, sumarizare, operații statistice, generare de grafice, și altele, în funcție de scopul final decis de către creator. Pentru a putea crea aceste pipelines, utilizatorul are la dispoziție o biblioteca de funcții,

pe care le poate integra în flow-ul său prin acțiuni simple de drag and drop în suprafață tip editor. Mai mult decât atât, el își poate scrie funcțiile proprii, iar reprezentarea vizuala în editor se va genera automat. După alegerea funcțiilor, și conectarea lor prin interfața grafică, prin apăsarea unui singur buton, pipeline-ul va fi "deployed", adică vă fi transformat din cod static, într-o serie de noduri mașină care vor accepta input și trimite output exact în ordinea definită de utilizator, la final obținându-se rezultatul dorit de către acesta. Flow-ul fiind activ, el va putea trimite acum date către procesare, tot prin intermediul interfeței. La final, va putea accesa rezultatele procesării.

Pentru a își putea îndeplini scopul ca produs, aplicația stabilește un număr de concepte și funcționalități, pe care le vom explora în continuare. Acestea includ graful computațional, care este o reprezentare vizuală a unui data pipeline, editorul, respectiv biblioteca de funcții.

3.2.1 Concepte introduse și funcționalități

Graful computațional

Definim un "graf computațional" ca un graf orientat fără cicluri, unde nodurile, respectiv muchiile, creează o reprezentare vizuală a unui flux de date. În această reprezentare, nodurile constau în funcții scrise și importate în platformă de către utilizator, iar muchiile orientate reprezintă flux-ul de date de la un nod la altul.[42] Pentru ca un graf computațional să fie valid, și deci transformabil cu succes într-un pipeline interactiv, care primește și transforma date, el trebuie să respecte o serie de restricții la nivel de noduri, respectiv muchii, prezentate în continuare. Verificările efective ale restricțiilor se fac la nivelul editorului de grafuri. O diagramă de activitate care ilustrează aceste lucruri se afla mai jos.

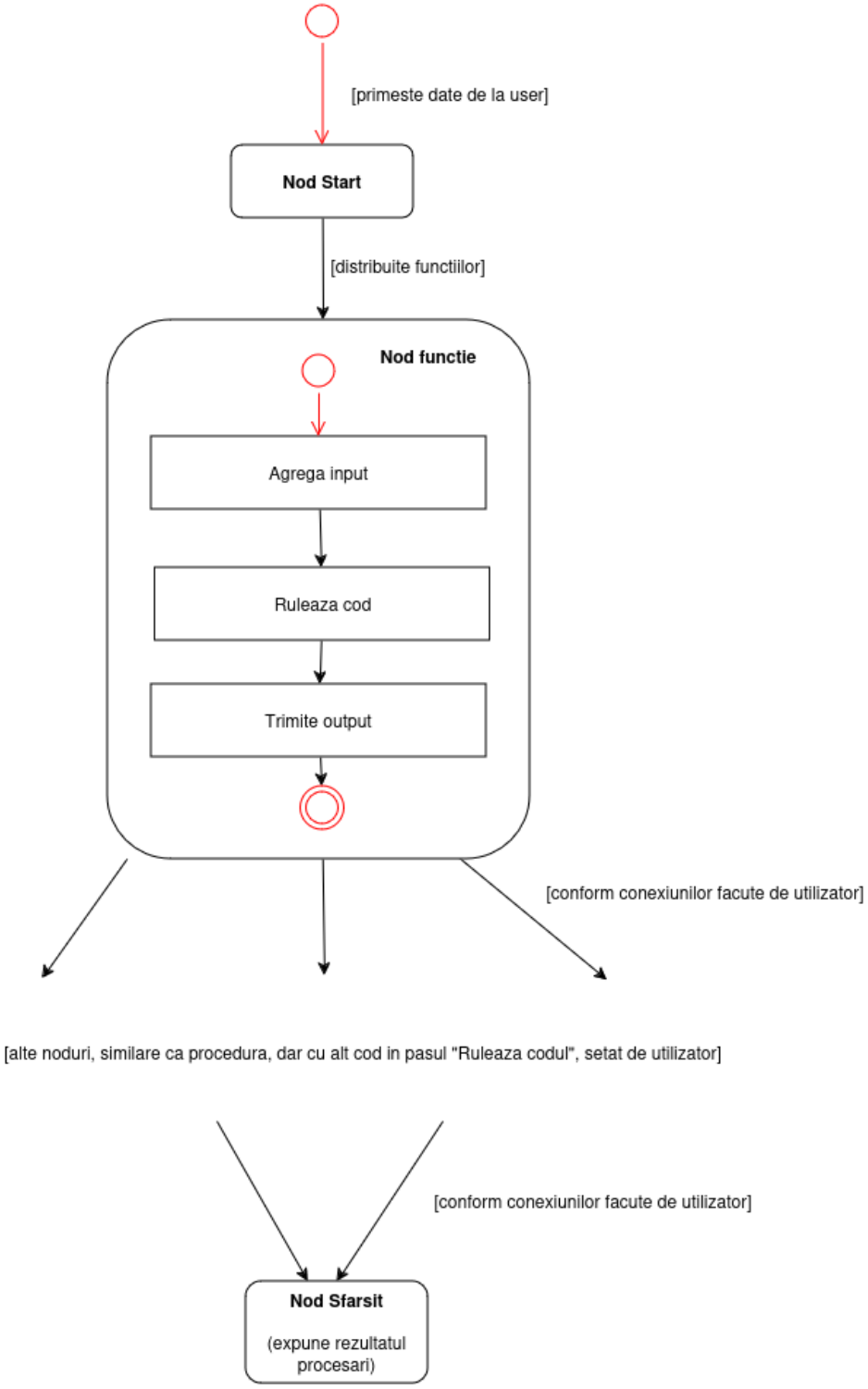

Figura 3.1: Diagramă de activitate care explică structura unui graf computațional generic. Toate nodurile în afară de cel de start și cel final funcționează întocmai ca cel ilustrat în detaliu, cu mențiunea că pasul "rulează cod" diferă în funcție de dorințele utilizatorului. De asemenea, configurația de conectare a nodurilor este decisă tot de către utilizator.

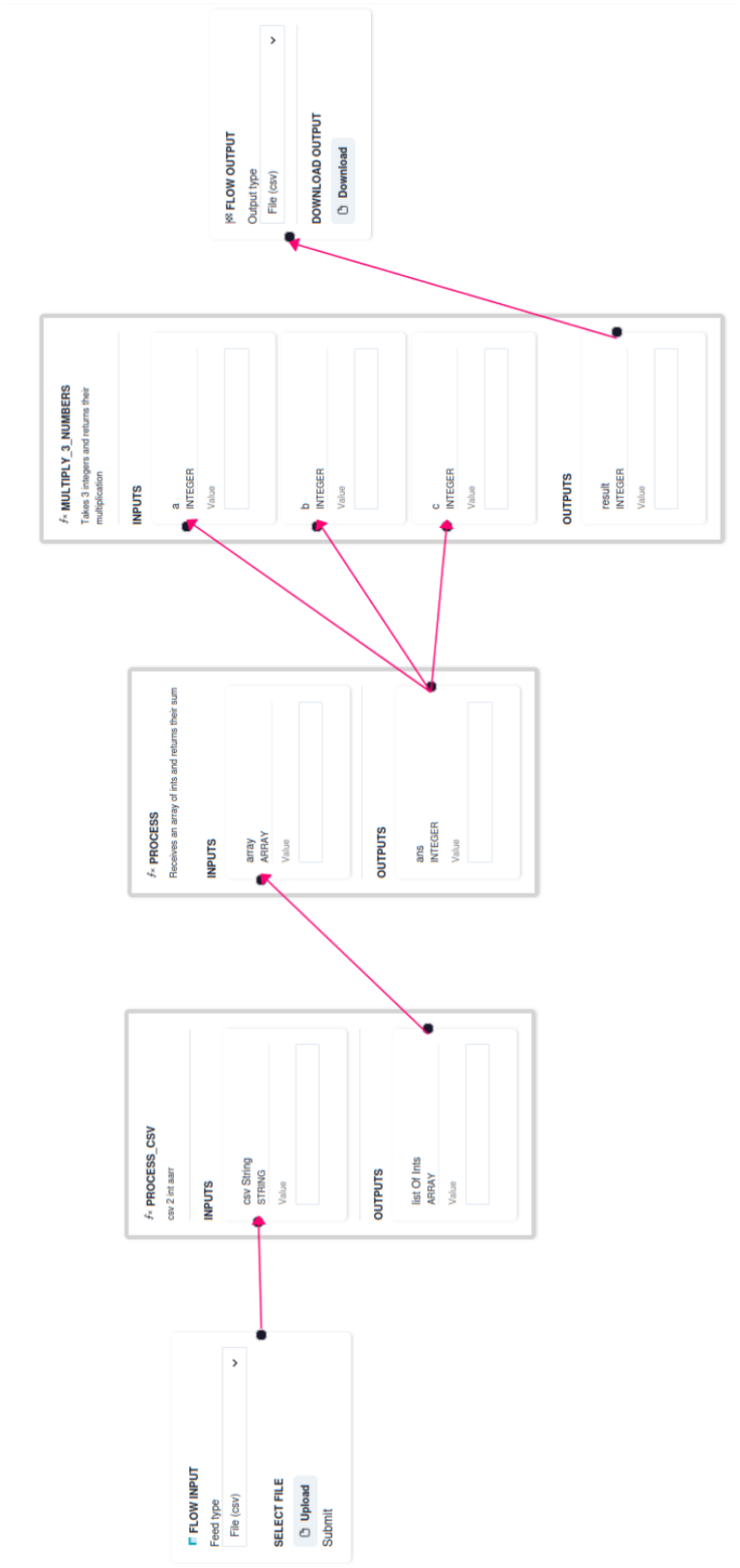

Figura 3.2: Captură de ecran din aplicație, a unui graf computațional complet, care primește informații sub formă de fișier CSV care conține rânduri cu numere, le transformă în array-uri de numere întregi, apoi le însumează și le colectează într-un fișier de output.

Noduri

În contextul aplicației noastre, un nod este o componentă a unui graf computațional ce reprezintă o abstractizare vizuală a unei funcții Python scrisă și încărcată în platformă de către utilizator. În cadrul nodului, apar detalii precum numele funcției, o descriere în limbaj natural a funcționalității ei, setată de către utilizator, cât și o reprezentare grafică a semnăturii funcției (atât argumentele de intrare, cât și cele de ieșire) [42]. Intern, nodul va ține în memoria sa codul funcției respective, așa cum arata ea la momentul când nodul a fost creat, pentru a evita posibilele probleme în pipeline la actualizarea codului în biblioteca de funcții. Cum utilizatorul este obligat să seteze tipuri de date pentru input și output-ul funcțiilor compatibile cu sistemul, ele sunt afișate aici, și se ține cont de ele la conectarea nodurilor, pentru a obține un graf valid și cu sens.

Drept exemplu, pentru o funcție simplă, ce acceptă un input de forma de lista de numere întregi, și calculează suma lor, platforma ar genera o reprezentare grafică ilustrată ca în Figura 3.3.

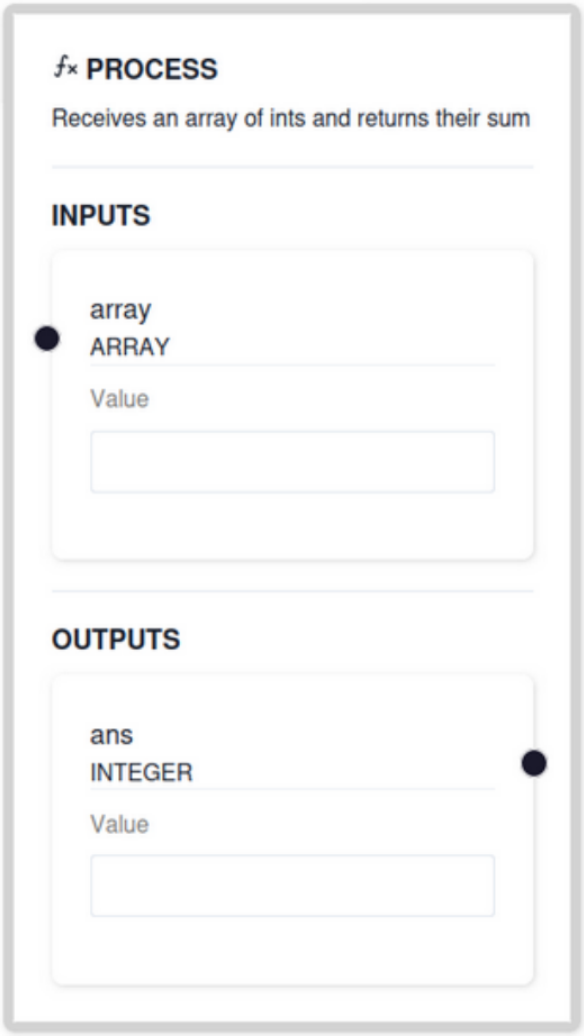

Figura 3.3: Reprezentarea grafică a unei funcții Python care primește ca input un array de numere întregi și returnează suma lor ca întreg.

Muchii

În stânga, respectiv dreapta, nodurilor din graf, se află puncte de conectare pentru datele de intrare, respectiv ieșire, ale funcției. Muchiile permit conectarea exclusiv dintre un punct de conectare tip input cu unul de tip output.

Direcția fluxului de date este dinspre output înspre input. Cu alte cuvinte, o muchie de la A la B semnifică întotdeauna că rezultatul apelului funcției A constituie date de intrare pentru funcția B.

Există suport pentru distribuire și grupare de input. Așadar, o funcție A care returnează 3 valori b, c, d, nu este obligată să trimită toate acele 3 valori către o funcție B, ci le poate distribui oricăror alte funcții din graf care acceptă input de tipul acela, existând mecanisme de sincronizare care garantează că funcția va rula doar atunci când primește tot input-ul de la toate sursele.

Figura 3.4: Nu se pot conecta output-uri cu input-uri unde tipul de date nu se potrivește exact. Fie nodul din stânga, corespunzător unei funcții care transforma un CSV cu o linie într-un array, și nodul din dreapta, care primește trei numere și le returnează suma. Cum tipurile de date nu se potrivesc, funcțiile nu pot fi conectate. Dacă totuși se dorește conectarea, ar trebui scrisă o funcție intermediară, adaptor, care, spre exemplu, ar putea primi un array, și returna primele trei valori din array, drept numere întregi.

Figura 3.5: Input-ul funcției multiply_3_numbers, care primește trei numere întregi și le multiplică, poate fi orice alt rezultat de tip întreg din orice alt nod din graful computațional. Nodul multiply_3_numbers va aștepta ca toate dependințele sale să fie calculate înainte de a fi rulat cu input-ul primit.

Editorul de grafuri

Editorul de grafuri oferă utilizatorului posibilitatea de a își administra, modifica, crea sau șterge grafuri. Panoul de selectare al grafurilor se înfățișează sub formă de listă. Prin selectarea unui graf, identificat după nume și status (dacă este activ sau nu), el deschide interfața de vizualizare, în care va putea vedea elementele care compun graful. Un graf este format din noduri și muchii orientate. Cu excepția nodului de start, respectiv cel de stop, care au funcționalități presetate (de încărcare a datelor neprocesate, respectiv descărcare a rezultatelor), restul nodurilor sunt adăugate de către utilizator. Un nod adăugat de utilizator reprezintă o abstractizare a unei funcții Python, încărcată în aplicație prin intermediul Bibliotecii de funcții.

În interfața tip editor, nodurile adăugate de către utilizator pot fi mutate, respectiv conectate cu alte noduri, pentru a putea crea secvența de procesare a datelor. Pentru ca acest lucru să aibă sens, iar flow-ul să ruleze corect, în editor se fac verificări la nivel de tip de date la conectare, asigurându-ne că putem conecta doar funcții unde acest lucru are sens, unde tipul de date al valorii de return al primei funcții să se potrivească cu tipul de intrare al celei de-a doua funcții.

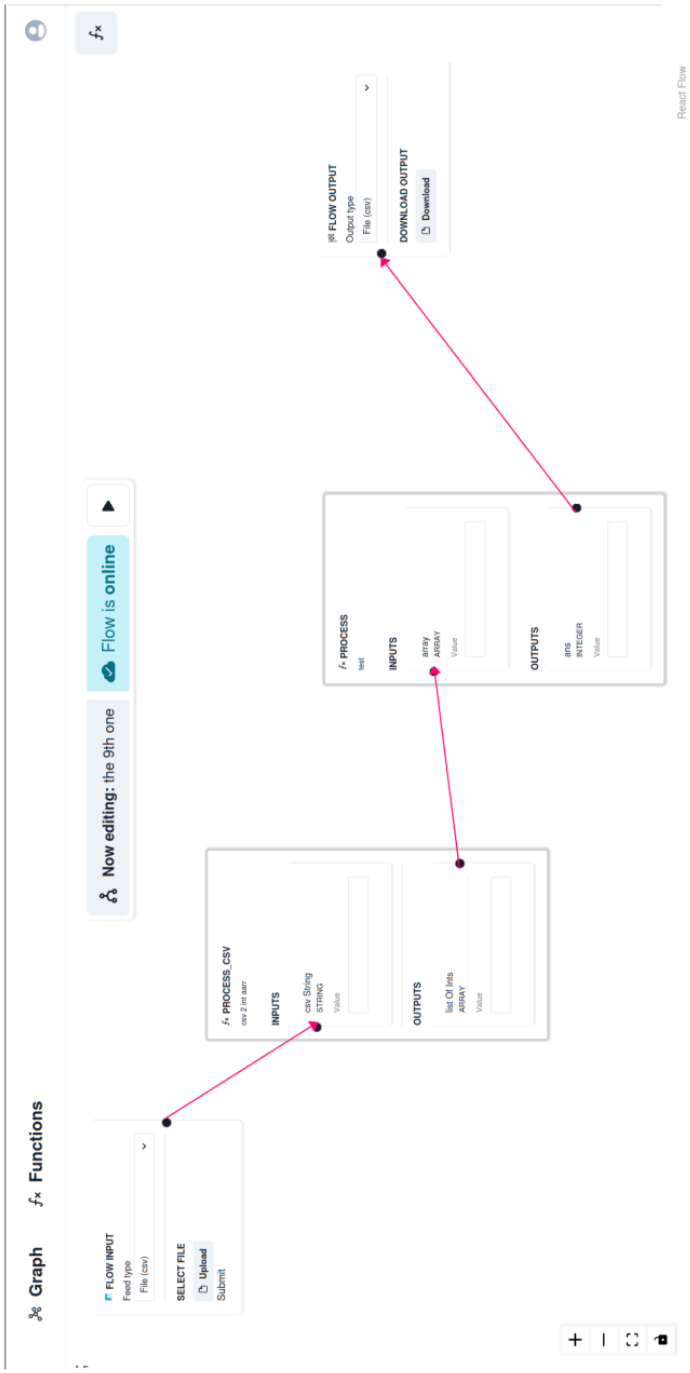

Figura 3.6: Captură de ecran prezentând funcționalitatea de editor a aplicației. În cadranul centru-sus se poate observa bara de status, care indică utilizatorului numele grafului pe care îl vizualizează, respectiv status-ul grafului (care poate fi online atunci când este funcțional și pornit, sau offline altfel). Bara din dreapta permite accesarea panoului cu biblioteca de funcții. Bara de sus permite deschiderea meniului pentru selectarea unui alt graf. Direcția muchiilor roz arată direcția în care datele curg de la nodul de Start la cel de Final, noduri speciale cu funcționalitate presetata.

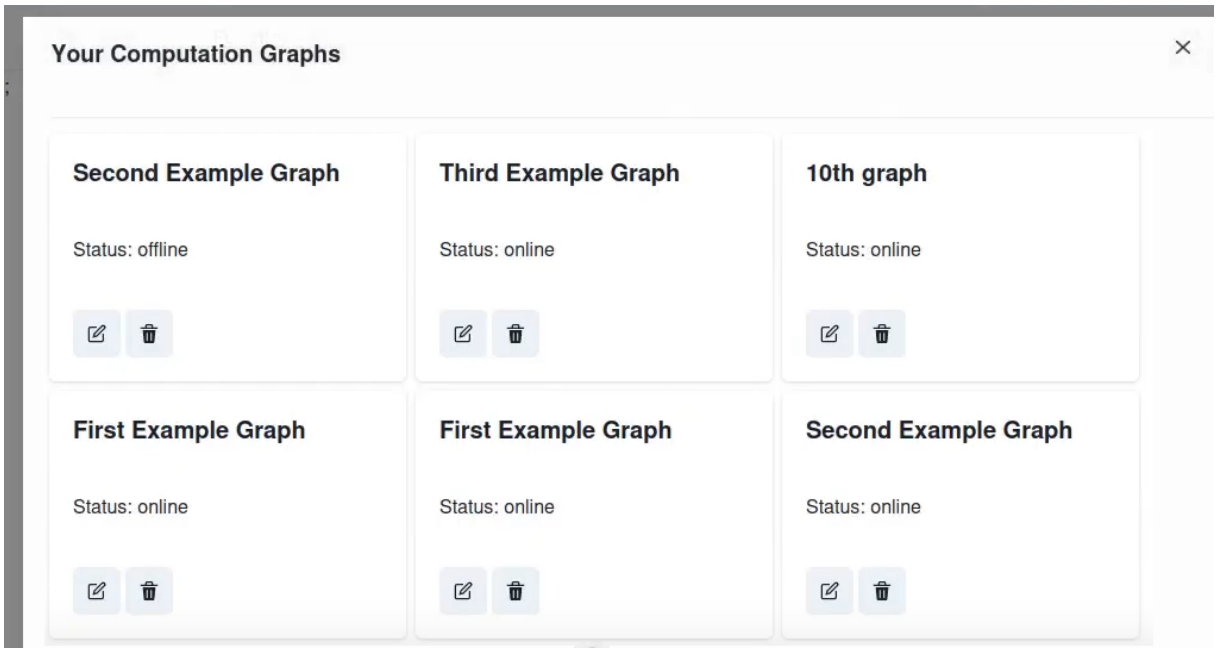

Figura 3.7: Interfață care permite utilizatorului să vizualizeze statutul, să editeze, sau să șteargă grafurile pe care le-a creat. De asemenea, utilizatorul poate vizualiza status-ul de deployment al grafului, dacă acesta este online sau offline.

Biblioteca de funcții

Biblioteca de funcții este un repository de GitHub ce conține funcții de Python scrise de către utilizatori. Am ales să folosim GitHub în detrimentul stocării într-o bază de date internă aplicației deoarece:

- Oferă un mediu familiar, ușor de folosit de către utilizatori,
- Permisiunile pot fi controlate prin intermediul access tokens, iar conținutul poate fi încărcat în aplicație prin API-ul lor,
- Introduce în aplicație cu ușurință un sistem de versionare/backup, indispensabil programării moderne,
- Oferă posibilitatea de dezvoltare integrală în browser prin conectarea la suite-uri precum Visual Studio Online, ce permit scrierea și testarea funcțiilor fără a fi nevoie de setarea unui mediu local.

Pentru a putea fi considerate valide și acceptate de sistemul nostru, funcțiile trebuie să respecte șablonul oferit, în care va trece în Input parametrii de intrare ai funcțiilor, obligatoriu cu tipurile de date, și în Output parametrii de ieșire, pentru a putea garanta flow-uri valide. În rest, utilizatorul își poate organiza codul cum consideră de cuviință, cu condiția că se rezumă la un singur fișier și folosește punctul de intrare stabilit.

```python
class Input(BaseModel):
    a: int
    b: int
    c: int

class Output(BaseModel):
    result: int

def multiply_3_numbers(input: Input) -> Output:
    result = input.a * input.b * input.c
    return Output(result=result)
```

Figura 3.8: Exemplu de cod pentru o funcție care multiplică trei numere. De remarcat formatul de intrare și ieșire "strongly typed", deși limbajul de programare este Python.

După ce funcția este scrisă și urcată în repository, ea poate fi adăugată și în interfața grafică a editorului de grafuri, unde vă fi vizibilă într-un meniu ce permite operații de tip drag and drop în editor.

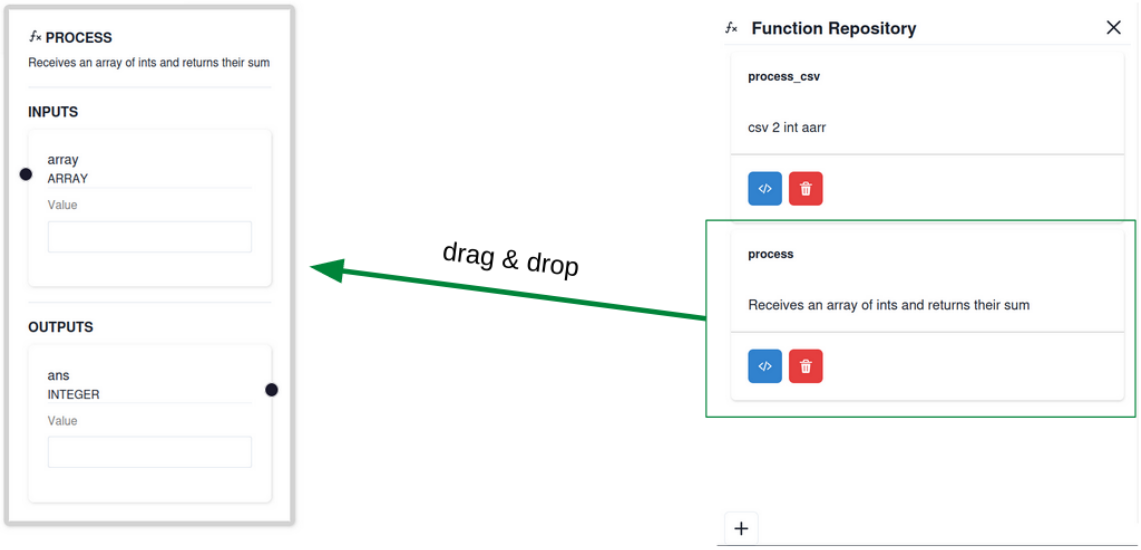

Figura 3.9: Interfața bibliotecii de funcții din aplicație, vizibilă în dreapta, permite acțiunea de drag and drop în editor, generându-se noduri care permit conectarea cu alte noduri.

Capitolul 4

Proiectarea platformei

4.1 Cerințe funcționale și nefuncționale

În urma efectuării planului de produs, am identificat setul de cerințe care trebuiesc îndeplinite pentru a obține un produs conform viziunii noastre.

4.1.1 Cerințe funcționale

1. Platforma trebuie să expună pentru utilizator o interfață intuitivă, ușor de folosit, sub forma de editor de graf.
2. Platforma trebuie să permită utilizatorului să administreze grafurile computaționale.
3. Platforma trebuie să permită deployment-ul grafurilor într-un mediu distribuit.
4. Funcțiile trebuie să poată comunica între ele și să își trimită date în ordinea corectă.
5. Platforma trebuie să permită utilizatorului să încarce funcții în aplicație.
6. Platforma trebuie să permită actualizarea codului funcțiilor din graf la cererea utilizatorului.
7. Platforma trebuie să prevină utilizatorul să genereze configurații de fluxuri de date incorecte, cu tipuri incompatibile
8. Platforma trebuie să prevină generarea de bucle în fluxurile de date create de către utilizatori.
9. Platforma trebuie să se conecteze la un repository de GitHub și să permită încărcarea funcțiilor definite în acel repository în platforma.

10. Platforma trebuie să genereze cu succes reprezentări grafice ale funcțiilor, marcând datele de intrare și ieșire, cât și tipurile lor, pentru funcțiile din repository-ul de GitHub conectat.

11. Utilizatorii trebuie să poată să contribuie cu funcții în repository-ul de GitHub, care, în urma aprobării, să poată fi încărcate și în platformă.

12. Platforma trebuie să permită conectarea și deconectarea funcțiilor în funcție de tipurile de date ale parametrilor de intrare și ieșire ale acelor funcții.

13. Platforma trebuie să permită utilizatorilor prin intermediul interfeței grafice să încarce seturi de date ce pot fi procesate, și să primească rezultatul procesării.

Cerințe nefuncționale

1. Platforma trebuie să fie intuitivă unui utilizator cu cunoștințe tehnice în domeniul analizei de date.

2. Platforma trebuie să fie bine optimizată astfel încât să funcționeze facil pe un dispozitiv cu specificații tehnice actuale.

3. Platforma trebuie să funcționeze pe toate sistemele de operare desktop moderne.

În urma acestor cerințe, cât și a analizei de produs efectuată anterior, se conturează următoarele zone de interes tehnic: frontend, backend, stocare, noduri de execuție și mijloacele de comunicare între noduri (fiind un mediu distribuit). În continuare, vom prezenta arhitectura aplicației, având aceste puncte în vedere. Vom începe prin prezentarea stack-ul tehnologic ales, concentrându-ne pe motivația din spate, cât și pe avantajele, respectiv dezavantajele aduse de alegerile făcute. Vom continua apoi cu diagrame ce descriu sistemul și interacțiunile dintre componentele sale, urmând ca în capitolul de Implementare **??** să detaliem procesul de implementare, cu nuanțele sale tehnice.

4.2 Alegerea tehnologiilor

4.2.1 Observații inițiale

Natura proiectului, o platformă de management al unei rețele de noduri de execuție conectate între ele, oferă niște provocări tehnice specifice. Pe partea de client-side, dată fiind structura flow-urilor de execuție, sub formă de grafuri orientate aciclice, este nevoie de o componentă de randare care să permită gestionarea acestor grafuri, cu operațiuni facile de modificare, conectare, ștergere, etc. În plus, pentru a putea garanta corectitudinea grafurilor prin intermediul restricțiilor de conectare (unde utilizatorul poate conecta doar

input-uri cu output-uri de același tip de date), este necesară dezvoltarea unui mecanism care să extragă tipurile de date din semnătura funcțiilor scrise de utilizator, generându-se o reprezentare corectă a acesteia.

În schimb, în server-side, provocarea principală este dată de mediul distribuit spre care țintim. Așadar, se conturează probleme precum stabilirea unui mecanism de transport de date, sincronizarea, persistența datelor, toleranța la erori, managementul nodurilor de execuție. În plus, dată fiind natura proiectului, ce permite rularea în mașini a codului scris de utilizatori, intervin dificultăți la nivelul de garantare a securității aplicației.

4.2.2 Frontend

React

React [51] este o bibliotecă populară utilizată în dezvoltarea interfețelor grafice pentru web. Dezvoltată de Facebook, având peste 10 ani în piață, este o tehnologie matură care a trecut testul timpului, fiind standard-ul de facto folosit în majoritatea aplicațiilor comerciale moderne. Câteva dintre punctele forte care ne-au făcut să considerăm o interfață în React ca fiind alegerea cea mai potrivită pentru platforma noastră sunt ecosistemul matur [66], cât și flexibilitatea la nivel de manager de stări, React fiind un motor grafic gândit sub paradigma de programare declarativă ce poate fi atașat oricărui sistem de stocare conforme, fapt de o deosebita importanță în cadrul aplicației noastre deoarece lucrăm cu structuri cu formă de graf, alături de reprezentarea lor în plan.

Un alt factor care diferențiază React de celelalte framework-uri moderne de dezvoltare de interfețe web precum Angular, sau Vue, este că biblioteca se bazează pe principii de programare declarativă, și se ocupă exclusiv de randarea componentelor[60]. Acest lucru este deosebit de important în cadrul arhitecturii noastre deoarece ne-a permis să modelam data layer-ul într-un format customizat, potrivit specificului neobișnuit al aplicației, în loc să folosim o paradigmă stabilită de către framework (cum ar fi fost situația pentru Angular). Acest lucru ne-a permis să scădem timpul de dezvoltare și să păstram performanța aplicației.

React-flow

Deoarece filosofia aplicației este centrată în jurul conceptului de graf de execuție, interfața grafică trebuie să permită crearea facilă a acestor grafuri. Așadar, este nevoie de o biblioteca care să permită acest lucru. React-flow[18] a fost alegerea noastră deoarece este cea mai populară bibliotecă din ecosistemul React care implementează acest concept[19]. Am ales această bibliotecă în detrimentul altor opțiuni de pe piață [**opțiuni**] pentru că a fost gândită într-un mod generic, astfel încât să poată fi extinsă pentru orice tip de reprezentare de graf, spre deosebire de alte biblioteci populare dar mai de nișă.

Astfel, biblioteca a oferit o bază bună peste care să putem crea funcționalitatea de generare a reprezentării grafice pe baza semnăturilor funcțiilor. Am putut adăuga mai multe puncte de conectare într-un singur nod, seta verificări la conectare, și restricționa conectarea doar între input-uri și output-uri.

JavaScript

JavaScript poate fi considerat, în ziua de astăzi, limbajul de facto al internetului. [35]. Este singurul limbaj de programare interpretabil în mod nativ de către toate browserele moderne (excluzând HTML și CSS). Pe lângă componenta de client, framework-urile moderne de JS permit crearea de backend-uri complexe într-un timp rapid, fapt ce se pretează bine pentru prototiparea unei platforme noi. În plus, folosirea aceluiași limbaj atât în frontend cât și în backend reduce fricțiunea din punct de vedere de dezvoltare, reducându-se context switching-ul. Acești factori au făcut ca scrierea în JavaScript a platformei să fie o alegere naturală.

4.2.3 Backend

Meteor.js

Meteor.js [30] este un framework de JavaScript recomandat utilizării pentru crearea aplicațiilor cu grad de interactivitate crescută. El introduce conceptul de Distributed Data Protocol [28], un protocol bidirecțional construit peste WebSockets care permite crearea cu ușurință de stream-uri de comunicare bidirecțională între server și client. În plus, framework-ul oferă suport pentru operațiuni de tip Remote Procedure Call[63], extrem de utile pentru declanșarea din client a anumitor operațiuni pe server. Am ales acest framework deoarece am considerat că se pretează foarte bine specificului platformei, iar îmbunătățirile de Developer Experience [58] vor scădea timpii de dezvoltare ai platformei.

MongoDB

În cadrul aplicației, avem nevoie de o bază de date care să stocheze structura grafurilor, funcțiile, datele despre utilizatori, etc. Cum putem avea grafuri de dimensiuni și cu configurații foarte variabile, am optat spre paradigma centrată pe documente și colecții, unde un document conține un graf, în detrimentul unei baze de date relaționale, care ar presupune operațiuni complexe de JOIN în cele mai simple situații. Astfel, deși bazele de date NoSQL sunt în mod general considerate ca fiind mai puțin eficiente [11], specificul aplicației contează foarte mult, economisind atât timp de calcul cât și timp de dezvoltare alegând paradigma cea mai potrivită.

Ne-am orientat spre MongoDB deoarece este printre cele mai populare baze de date NoSQL. În plus, framework-ul Meteor.js este foarte bine integrat cu Mongo, ușurând

experiența de dezvoltare.

RabbitMQ

RabbitMQ[56] este un broker de mesaje [34] care poate fi folosit în implementarea pattern-ului de comunicare bazat pe cozi și mesaje. El funcționează ca un intermediar, definind canale de comunicare numite cozi, și trimițând mesajele de la un serviciu la altul prin acestea.

Cozile de mesaje sunt des folosite în programarea distribuită pentru a facilita comunicarea dintre servicii care rulează independent, decuplat. În cazul platformei noastre, cozile de mesaje sunt un punct critic. La deployment-ul unui graf creat de utilizator, nodurile de execuție care primesc codul scris de aceștia, sunt conectate în mod dinamic, la runtime, prin cozi. Modul în care a fost implementat acest lucru în cadrul aplicației este o problemă tehnică interesantă, fiind un mod neobișnuit de a lucra cu cozi de mesaje, și va fi explorată în secțiunile următoare.

Am ales RabbitMQ deoarece este un broker minimalist, fiabil, care și-a dovedit în timp rezistența. [6]

Docker

Docker [13] este o platformă care permite dezvoltatorilor de aplicații să ruleze cod în environment-uri numite containere. Spre deosebire de mașinile virtuale, aceste containere se folosesc de kernel-ul sistemului de operare al mașinii gazdă, și oferă un mediu în care aplicația alături de dependințele ei sunt izolate de restul sistemului. [14]

Infrastructura distribuită a aplicației este construită peste Docker, existând containere separate pentru backend, baza de date, respectiv message queue broker. În plus, fiecare nod de execuție rulează într-un container separat, pentru a crea izolare între zonele de cod scrise de utilizatori, care se afla în execuție, și restul platformei. Toate aceste containere formează o rețea, definită printr-un fișier docker-compose [12]. Topologia rețelei este variabilă datorită mecanismului de scalare pentru nodurile de execuție, numărul acestora scalând în funcție de cât de solicitat este sistemul.

Python

Python este cel mai popular limbaj folosit pentru efectuarea operațiunilor din știința/analiza datelor. [7] Așadar, este limbajul cel mai familiar pentru grupul țintă de utilizatori al platformei. Din acest motiv, am ales ca, deși restul infrastructurii este scrisă în JavaScript, nodurile de execuție să fie scrise în Python, pentru a putea permite utilizatorilor să folosească un limbaj și un sistem de pachete foarte familiar lor, în scopul de a ușura adopția platformei și de a o integra în ecosistemul de unelte al domeniului. Fiind

scris în Python, nodul de execuție poate integra și rula în mod nativ codul de Python scris de către utilizator.

Pydantic

Pydantic este o bibliotecă pentru Python folosită pentru validarea datelor [47]. În cazul platformei noastre, ea este necesară pentru a garanta corectitudinea la nivel de flux de date. Îndeplinim acest lucru setând restricția ca utilizatorii care își creează funcții, să fie nevoiți să seteze tipurile de date pentru datele de intrare și ieșire folosind Pydantic. Astfel, ne putem folosi de funcționalitățile de casting, validare, și generare de schema [46] puse la dispoziție de bibliotecă pentru a realiza scopul produsului.

4.3 Arhitectura aplicației

4.3.1 Microservicii versus monolit

Prima întrebare cu care ne-am confruntat în pasul de design de arhitectură a fost dacă alegem să optăm pentru o arhitectură monolitica[62] sau una bazată pe microservicii[38], aceste două moduri de organizare fiind cele mai folosite în prezent. Ambele moduri vin cu avantajele și dezavantajele proprii. La nivel teoretic, monolitul oferă avantaje precum simplificarea dezvoltării, timpi mult mai scăzuți de comunicare între componente, respectiv ușurință la deployment și testare pentru produs. În schimb, microserviciile oferă oportunități de scalare, sunt arhitecturi mult mai flexibile și adaptabile, iar în cazul lucrului în echipă, într-o organizație, paralelizarea lucrului între mai multe echipe aduce cu ea o potențială îmbunătățire a vitezei de dezvoltare.

Răspunsul la aceasta întrebare este în general dificil [2], fiind avantaje și dezavantaje de ambele părți, neexistând un mod de organizare obiectiv mai bun decât celălalt. În mod normal, specificul aplicației face ca avantajele unui mod de organizare să se potrivească mai bine decât celălalt. Acest scenariu a avut loc și în situația noastră. Prin natura aplicației, la nivelul grafurilor de execuție, un sistem bazat pe microservicii aduce avantaje care sunt esențiale în îndeplinirea misiunii produsului. Posibilitatea de scalare orizontală a rețelelor de noduri generate de către utilizatori, cât și izolarea codului oferit de către aceștia, sunt factorii decisivi ce ne-au convins în a alege o arhitectură bazată pe microservicii. Totuși, am ales ca centrul de comandă, în implementarea sa curentă, sa fie scris sub forma unui singur serviciu mai mare, ce îndeplinește multiple scopuri. Acest fapt ne-a permis să ne concentrăm în principal pe demonstrarea conceptului aplicației. Centrul de comandă va putea fi desprins în mai multe servicii fără mare dificultate, într-o etapa ulterioară, pentru a beneficia la rândul sau de scalarea orizontală. Astfel, pentru această aplicație, putem afirma ca am ales o abordare bazată pe microservicii cu dimensiuni variabile, foarte comună în industrie. [53]

4.3.2 Privire de ansamblu

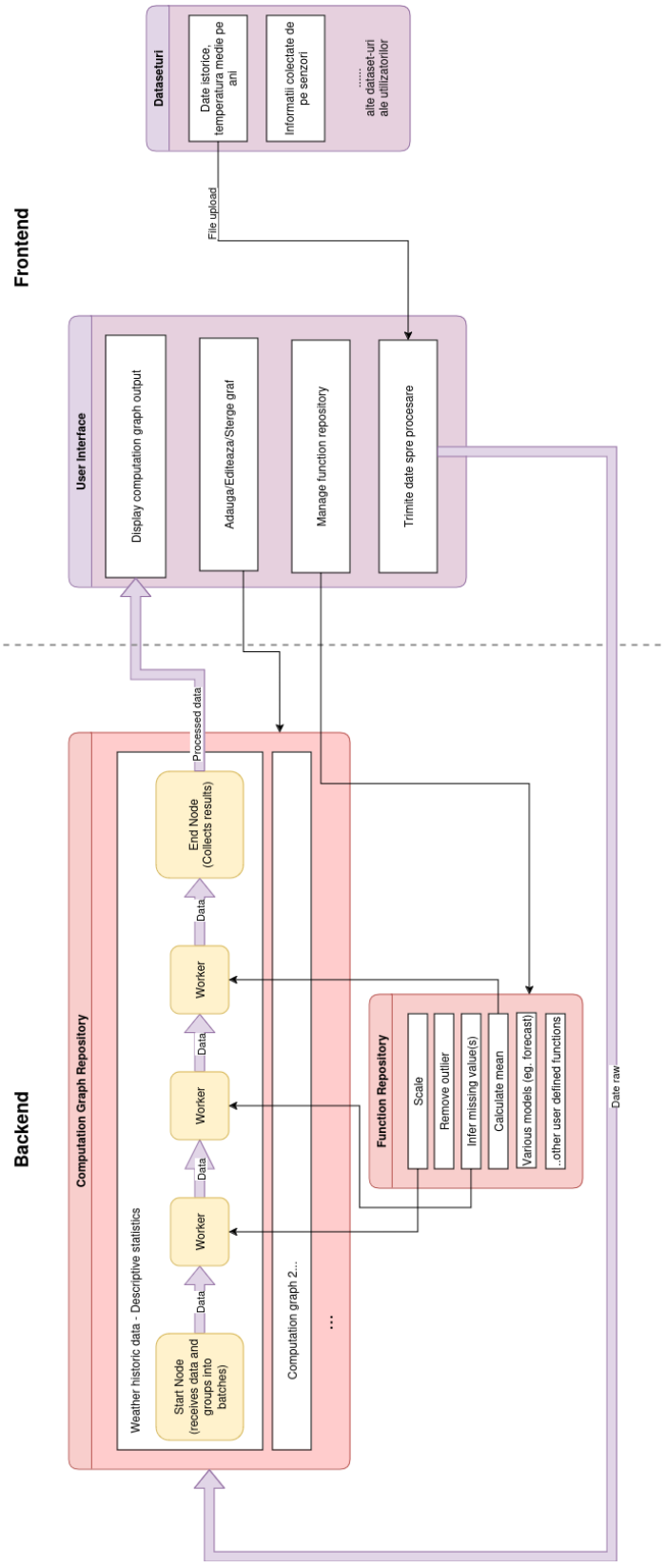

Figura 4.1: Privire de ansamblu asupra arhitecturii bazate pe microservicii din perspectivă funcțională și de flux de date

4.3.3 Comunicarea dintre componente

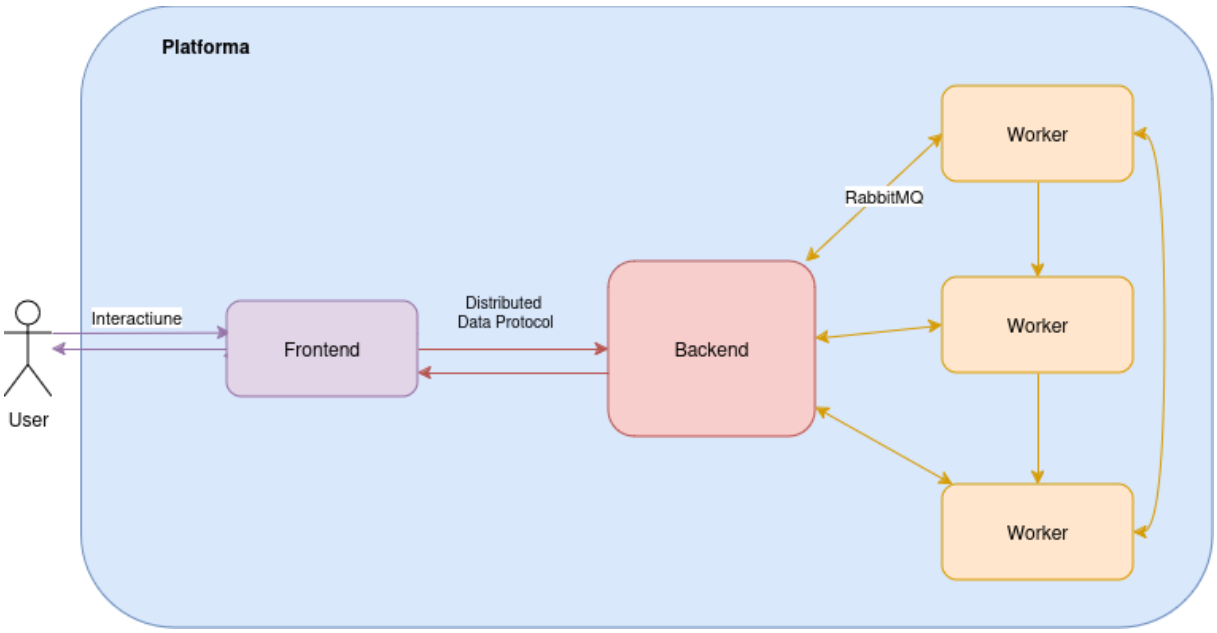

Figura 4.2: Diagrama care ilustrează comunicarea dintre straturile aplicației cât și canalele alese. Utilizatorul interacționează cu platforma prin intermediul frontend-ului. Frontend-ul și backend-ul sunt într-o continuă comunicare prin intermediul protocolului DDP[28]. Totodată, comunicarea dintre backend și workeri, respectiv dintre workeri, se face prin intermediul cozilor de RabbitMQ.

4.3.4 Baza de date

Baza de date utilizată în cadrul aplicației este o instanță de MongoDB, aflată în containerul ei propriu, care conține colecțiile pentru:

- biblioteca de funcții
- reprezentările grafurilor computaționale
- utilizatori din platformă
- nodurile de execuție active
- caching pentru datele din pipeline-uri

Deși în implementarea curentă, ea este centralizată, implementarea nu ține cont de acest lucru, fiind foarte ușor ca într-o iterație viitoare a produsului, baza de date să fie la rândul său distribuită sau sharded[40].

Pentru a interacționa cu baza de date, am implementat un data layer folosind paradigmele specifice Meteor.js, Publications [31], care determină pentru un endpoint de tip

GET, care dintre documentele dintre colecția cerută să fie returnate, funcționând că o filtrare, respectiv Methods, care sunt funcții ce pot fi apelate din frontend prin Remote Procedure Call [63], și care pot declanșa atât operațiuni asupra bazei de date, cât și operațiuni de deployment și control al rețelei de noduri de execuție.

Capitolul 5

Implementarea aplicației

În urma completării etapei de design arhitectural, am obținut cele necesare pentru a putea începe dezvoltarea efectivă a platformei. Platforma a fost dezvoltată în decursul a șase luni, plecând de la elementele de sine stătătoare, care interacționează cu un număr restrâns de alte elemente (cum ar fi reprezentarea grafică, biblioteca de funcții sau nodul de execuție de sine stătător) și extinzându-ne spre componente cu dependințe ridicate (procesul de deployment, comunicarea între servicii).

În continuare, vom discuta detaliile de implementare ale aplicației, punând accentul pe provocările tehnice interesante apărute pe parcurs. Vom detalia procesul de soluționare, de la cercetare, la implementare, la eventualele compromisuri care au fost necesare, după caz.

5.1 Organizarea mediului de dezvoltare

Am început dezvoltarea prin configurarea unui mediu care să permită dezvoltarea tuturor serviciilor care compun aplicația.

5.1.1 Sistem de versionare

Am folosit git[54] ca sistem de versionare, iar ca locație pentru repository-ul remote, GitHub[24]. Fiind un proiect bazat pe microservicii, una dintre deciziile care a trebuit făcută la acest pas a fost alegerea dintre o structură monorepo, în care toate microserviciile aparțin unui singur repository, sau multirepo, în care fiecare microserviciu este ținut într-un repository separat. Ambele metodologii vin cu un set de avantaje și dezavantaje. În timp ce un monorepo ușurează procesul de dezvoltare, pot apărea probleme la timpii de build, respectiv la managementul dependințelor. În schimb, separarea în mai multe repository-uri vine cu un bonus de modularizare. [43]

În cazul nostru, am ales o abordare care este centrată pe ideea de monorepository. Din perspectiva de componente, toate microserviciile din aplicație se afla în același repository

remote pe GitHub. Totuși, în situația noastră, având de a face cu cod generat de către utilizator și care este introdus apoi în platformă, prin intermediul bibliotecii de funcții, am identificat nevoia de a izola funcțiile generate de utilizatori într-un repository separat. Așadar, structura curenta se bazează pe două repository-uri, un monorepo conținând toate serviciile și structura care aparține platformei, și un alt monorepo care conține exclusiv funcțiile generate de utilizatori.

5.2 Biblioteca de funcții

Prima componentă importantă care a fost implementată a fost biblioteca de funcții. Conform design-ului de produs, aceasta bibliotecă oferă utilizatorilor acces, din intermediul platformei, atât la funcțiile scrise de ei, cât și de la alții, formând astfel un ecosistem de lucru complet.

Biblioteca de funcții este un repository de GitHub care este integrat cu restul platformei prin folosirea de Access Tokens, via GitHub API. Aceasta conține funcții adăugate de către utilizatori, care pot fi mai apoi încărcate în interfața grafică și conectate, formându-se fluxuri de date. La deployment, structura de graf formată se transformă într-o rețea de noduri de execuție ce rulează codul dat de utilizatori, transferul de date de la un nod la altul efectuându-se prin cozi de mesaje stabilite dinamic.

5.2.1 Despre funcții

O *funcție* în contextul nostru este un fișier Python ce conține cod, variabile, și o metoda ca punct de start, denumită în continuare entry-point, care va fi rulată atunci când graful este deployed, iar nodul de execuție care are funcția atașată primește input-ul. Pentru a fi validă, deci utilizabilă în cadrul platformei, metoda cu rol de entry-point trebuie să respecte șablonul dat la generare. În plus, numele fișierului și numele funcției entry-point coincid. În rest, codul poate lua orice structura dorită de utilizator, atât timp cât este tot în același fișier, restricție ce nu ar trebui să fie problematica, dată fiind filosofia de modularitate pe care se bazează produsul.

```python
from pydantic import BaseModel

class Input(BaseModel):
    # add your input parameters here, with types

class Output(BaseModel):
    # add your output parameters here, with types

def function_name(input:Input) -> Output:
    # add your function logic here
```

Figura 5.1: Fragment de cod care are rolul de șablon de start pentru utilizatorii care doresc să adauge funcțiile proprii în platformă

5.2.2 Implementare

În implementarea curentă, utilizatorii pot interacționa cu acest repository prin intermediul GitHub Codespaces, nefiind nevoie să își seteze propriul environment local [23]. Astfel, îndeplinim condiția de dezvoltare exclusiv în browser, stabilită în etapa de produs, păstrând totodată familiaritatea unui mediu de dezvoltare consacrat, și oferind posibilități de rulare, testare, și debugging al funcției, înainte de urcarea ei în platformă.

Pentru a-și adaugă funcția nou-scrisă în bibliotecă, și a folosi-o în platformă, ei trebuie să deschidă un pull request care adaugă funcțiile lor în repository, respectând formatul-șablon dat. La trimiterea unui pull request, un administrator al platformei este notificat prin e-mail. Acesta poate aproba funcțiile spre adăugare în codebase. Validarea de către o persoană de încredere adaugă un pas în plus de securitate, fiind vorba de cod nou introdus în ecosistem.

Cum filosofia de produs din spatele funcțiilor este bazată pe modularitate, din fericire, această operațiune nu ar trebui să se întâmple foarte des. Cum toți utilizatorii au acces la biblioteca de funcții, există șanse foarte bune ca o bună parte, dacă nu chiar toate funcțiile dorite să existe deja în bibliotecă. Dacă un pas nu există, el poate fi scris și introdus în platformă, iar data viitoare când va fi nevoie de el, el va putea fi folosit direct, fără rescriere.

Figura 5.2: Captura de ecran care demonstrează funcționalitatea de integrare cu Github Codespaces a bibliotecii, pentru dezvoltarea exclusiv remote a funcțiilor care pot fi folosite în aplicație. În urma dezvoltării de către utilizatori a funcțiilor în acest mediu interactiv, ele pot fi încărcate în platformă, care le va genera o reprezentare grafică, le va permite conectarea cu alte funcții, și încărcarea automată în cloud la pasul de deployment

În backend, codul este accesat și preluat în aplicație folosindu-se GitHub API citegithub-api. Un access token [25] care permite citirea din repository-ul care conține biblioteca de funcții a fost introdus în platformă, astfel încât codul să poată fi accesat din platformă.

5.2.3 Adăugarea în platformă

Integrarea în platforma a funcțiilor scrise de utilizatori este on-demand, și făcută în doi pași. După ce codul a fost aprobat și adăugat în repository-ul bibliotecii de funcții, ea trebuie adăugată și în interfață. Acest pas este foarte simplu, făcându-se din frontend. Figura **??** ilustrează interfața care permite acest lucru. În urma adăugării, funcția va fi vizibilă în interfața din platformă a bibliotecii, iar prin pasul de drag and drop, poate fi adăugată în suprafața de lucru și conectată cu alte funcții.

La adăugarea în platformă, se cere completarea unor metadate, precum descrierea care să fie redată în platformă. Intern, se creează un nou document în colecția de Mongo care colectează funcțiile, ce conține aceste metadate. Apoi, se trimite un call GET către GitHub API pentru a obține codul din fișierul funcției care a fost tras în editor. Acest cod este luat din fișier și transformat în string. În această etapă, are loc, de asemenea, o prevalidare. Codul de Python al utilizatorului este rulat într-un container separat pe server, cu privilegii reduse, pentru a verifica corectitudinea lui și a ne asigura că nu exista erori. Dacă acest pas funcționează cu succes, vom genera o reprezentare de tip JSON schema a datelor de input și output, funcționalitate oferită de biblioteca Pydantic prin funcția `model_dump_schema()` [46], și o vom salva și pe aceea alături de codul dat de utilizator, în documentul din colecția Mongo.

```python
import json
from enum import Enum
from typing import Annotated, List

from pydantic import BaseModel, Field, StrictInt
from pydantic.config import ConfigDict


class Input(BaseModel):
    array: List[int]

class Output(BaseModel):
    ans: StrictInt

def process(input: Input) -> Output:
    return Output(ans=sum(input.array))
```

Figura 5.3: Code snippet care ilustrează o posibilă funcție ce poate fi adăugată de către utilizator în biblioteca de funcții. Această funcție primește un array de numere întregi și returnează suma lor

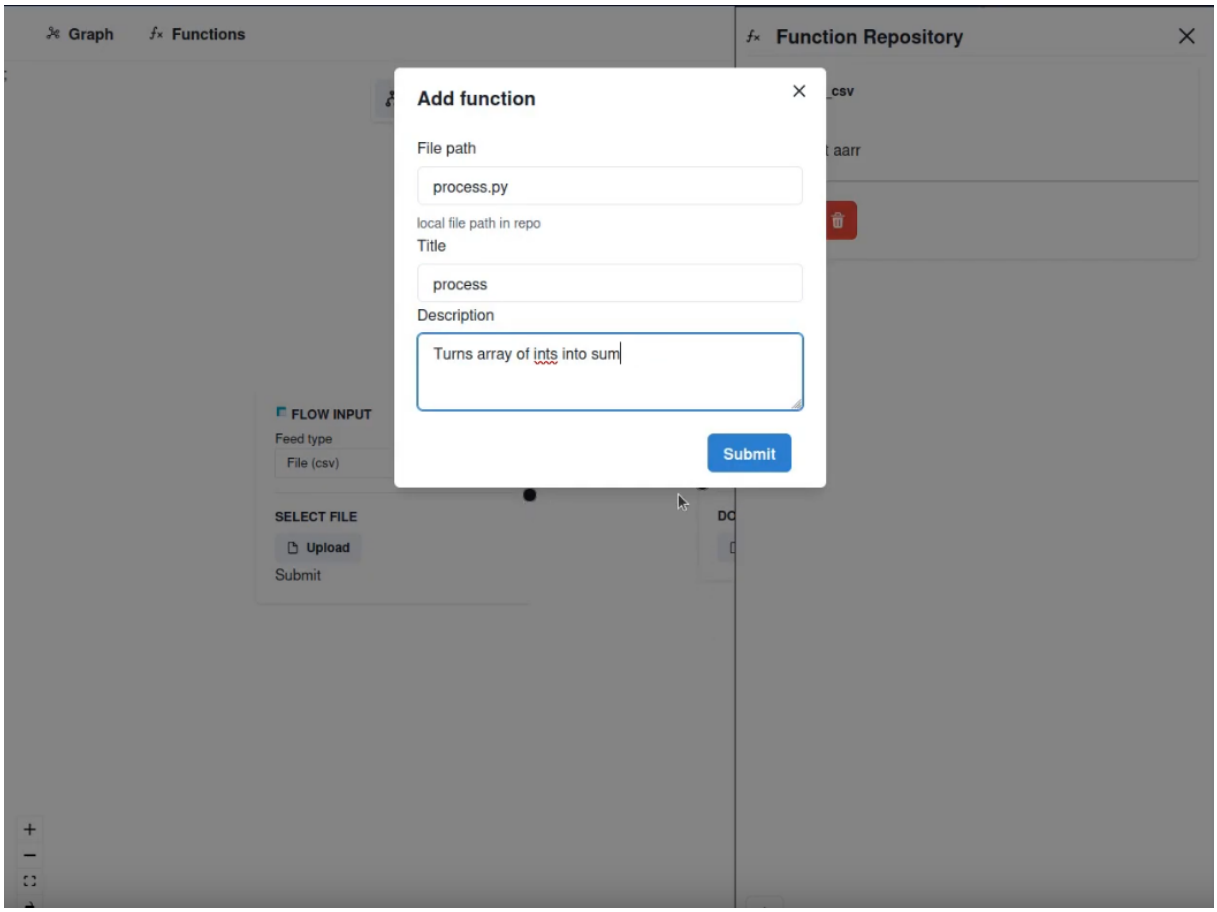

Figura 5.4: În urmă adăugării în bibliotecă, funcția poate fi încărcată apoi în platformă prin intermediul interfeței grafice. În urma acestei adăugări, ea poate fi trasă în editor-ul de lucru și conectată cu alte funcții pentru a obține un graf computațional.

5.3 Editorul de grafuri

5.3.1 Interfața grafică

Elementele de control din pagină (precum bara de stare, bara de unelte din partea dreaptă), au fost implementate cu Chakra UI [65], o bibliotecă de componente grafice modernă utilizată în crearea de interfețe estetice, care respectă principii moderne de UI/UX. Am dedicat majoritatea spațiului din ecran editor-ului de grafuri, implementat folosind biblioteca react-flow.

Editorul de grafuri oferă un sistem în care utilizatorului îi este înfățișată o suprafață de lucru cu care poate interacționa prin intermediul unor unelte puse la dispoziție prin bare aflate în jurul suprafeței. În cazul nostru, suprafața de lucru oferă în partea din dreapta acces la panoul bibliotecii de funcții, iar în partea de sus, oferă date despre statutul grafului (dacă este activ sau nu), respectiv posibilitatea de a controla acest lucru. Pentru a lucra cu ușurință, utilizatorul poate mări, micșora, sau se deplasa în suprafață folosind

mouse-ul. De asemenea, elementele constituente ale grafului pot fi mutate și rearanjate în plan după bunul plac. Punctele de conectare, desenate ca niște cercuri negre, aflate în partea stângă, respectiv dreaptă a nodurilor, corespunzând datelor de intrare, respectiv de ieșire din graf, pot fi conectate cu ajutorul mouse-ului, oferind o experiență de utilizare firească.

Prin editor, utilizatorul poate manipula structura grafului computațional, pregătindu-l pentru deployment. În fapt, în interfața grafică, există o conexiune între colecția din baza de date Mongo corespunzătoare grafurilor computaționale, și frontend, acest lucru fiind realizat prin biblioteca Minimongo[41]. În frontend, în Minimongo, pot fi accesate doar grafurile utilizatorului conectat în mod curent, filtrarea fiind făcută la nivel de server prin Meteor.js Publications [27]. Mai mult decât atât, folosind în interfață hookul de React useTracker [29], pus la dispoziție de Meteor.js, putem obține comunicare bidirecțională între frontend și backend. Astfel, modificările efectuate prin apelarea de metode Meteor, de pe server, prin RPC[63], care afectează elementele din baza de date, vor fi anunțate înapoi către front-end prin acest hook, obținându-se astfel interactivitate.

În situația interfeței noastre, fiecare modificare de pe ecranul utilizatorului este trimisă către backend prin Meteor Methods, iar rezultatul modificării revine către interfață prin Subscription. Pentru a ascunde latența și a obține o experiență de utilizare mai plăcută, în frontend este, de asemenea, calculat efectul modificării și afișat prezumptiv în interfață, efect denumit Optimistic UI[32].

5.3.2 Graful computațional

Graful computațional este structura de date care descrie secvența de pași prin care un set de date este transformat din forma sa originală, neprelucrată, într-o formă utilă analistului de date. Este un graf orientat aciclic[36] în care datele curg de la un nod de Start, ce poate primi prin interfață fișiere cu date în format CSV, la nodul de Stop, care expune către utilizator o modalitate de a descarcă rezultatele procesării sale. În procesul de prelucrare, datele sunt trecute printr-o înlănțuire de funcții, configurată de către utilizator, care le prelucrează în formă dorită.

Din punct de vedere al stocării datelor, graful computațional este memorat în baza de date exact în formatul folosit și la randare, de către react-flow, sub formă de document cu lista de noduri și lista de muchii.

5.3.3 Noduri

La tragerea prin drag and drop în interfață a unei funcții existente în biblioteca de funcții, în planul de lucru se va genera reprezentarea sa grafică, denumită în continuare nod. Acest nod conține parametrii de intrare și ieșire ai funcției, tipurile de date, respectiv punctele de conectare pe laturile stânga și dreapta. Punctele de conectare din stânga

corespund argumentelor de intrare, iar cele din dreapta corespund celor de ieșire. Un astfel de exemplu poate fi văzut în figura următoare:

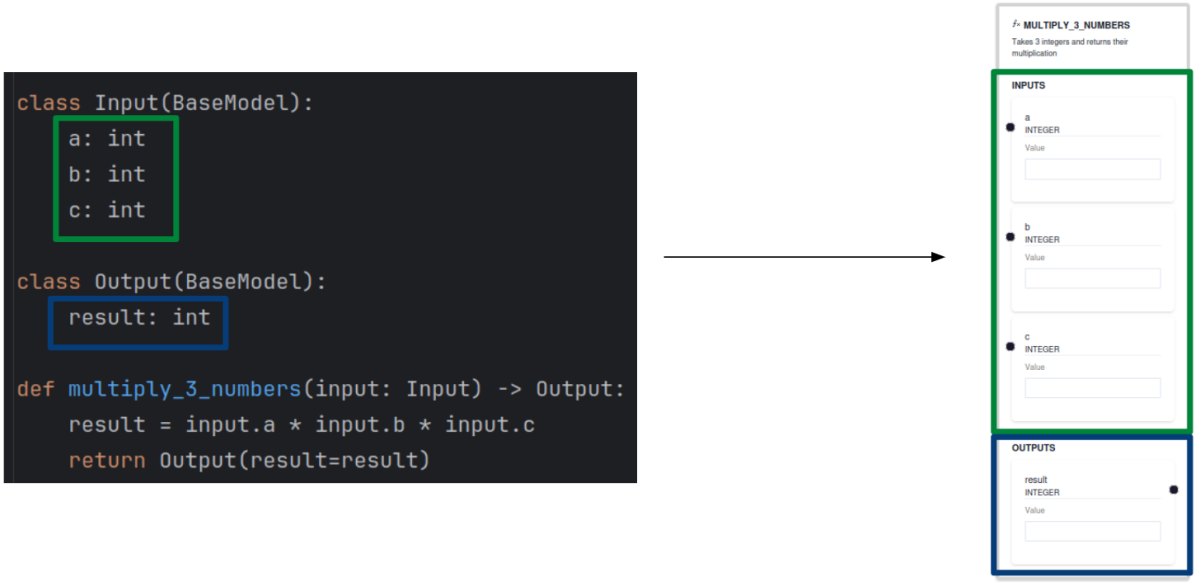

Figura 5.5: Corespondența dintre un exemplu de funcție scrisă de către utilizator și reprezentarea sa grafică generată. Argumentele de intrare sunt marcate cu culoarea verde, iar cele de ieșire cu albastru. Poate fi observată corespondența între tipurile de date, și faptul că numele parametrilor este de asemenea preluat.

Generarea reprezentării

Adăugarea unui nod în interfața grafică, prin drag and drop din biblioteca de funcții, ia documentul asociat din colecția Mongo și îl pasează motorului de randare grafic, bazat pe react-flow. Nodul afișează într-un card[64] numele, descrierea, cât și parametrii funcției. Pentru a putea genera corect elementele grafice ale parametrilor, se folosesc schemele JSON[46] corespunzătoare semnăturii funcției, care au fost generate la adăugarea funcției în biblioteca de funcții. Fiind un format standardizat, putem itera prin aceasta, și recupera numele parametrilor, cât și numele tipului lor de date, pe care le putem afișa în graf. Punctele de conectare generate sunt configurate astfel încât doar cele care corespund unui câmp din input-ul funcției să fie conectabile cu vreunul din output-ul unei alte funcții. Astfel, este îndeplinită filosofia de flux de date, care se propagă de la o funcție la altă.

5.3.4 Muchii

Muchiile sunt create prin conectarea a două puncte care corespund unor funcții diferite, unde cele două câmpuri conectate au același tip de date. În fapt, o muchie este un element din lista de muchii asociată grafului, și conține informații precum sursă și destinație (target).

Figura 5.6: Bara de stare din colțul dreapta-sus al aplicației care expune funcționalități de adăugare și management al grafurilor din sistem

Încercarea de a crea o muchie între două noduri din graf trece printr-o serie de validări:

- ca muchia să fie trasă de la un punct de conectare corespunzător unui câmp din datele de intrare ale unei funcții, cu un câmp din datele de ieșire ale altei funcții, pentru a păstra principiul de data flow,
- ca tipul de date al celor două elemente ce se dorește a fi unite să se potrivească perfect.

Dacă aceste condiții sunt îndeplinite, muchia este validă, deci este salvată în graful computațional.

5.3.5 Salvare și încărcare

Încărcarea unui graf în suprafața de lucru se face prin accesarea meniului din bara aflată în partea de sus a paginii. Aceasta oferă opțiuni pentru deschiderea, adăugarea, respectiv ștergerea unui graf din sistem, și funcționalitatea de logout pentru utilizator.

Salvarea grafului are loc automat, deoarece fiecare mișcare a utilizatorului în frontend este propagată și salvată în timp real în backend, mulțumită protocolului DDP[28].

5.4 Deployment în rețeaua de noduri de execuție

În urma manipulării interfeței de lucru din frontend, utilizatorul obține graful său de execuție. Prin apăsarea butonului de ”Play” din bara de stare, se poate începe procesul de deployment.

Procesul de deployment se folosește de entități numite Noduri de execuție, care sunt containere Docker care rulează câte un script Python compus din mai multe corutine [20] cu scopuri prestabilite. Există o entitate ce are rol de punct de control, numită în continuare ”Centrul de comandă”, un container Docker privilegiat [55], care urmărește starea nodurilor în rețea, poate porni sau opri nodurile, le poate da task-uri, și poate

scala numărul de noduri în funcție de nivelul de utilizare. În continuare, vom prezenta aceste entități, cum funcționează ele, cât și pașii pe care procesul de deployment ii execută pentru a trece de la o simplă reprezentare de tip graf, la o rețea funcțională de noduri și cozi, generată on-demand.

5.4.1 Nodul de execuție

Un nod de execuție este un container de Docker căruia îi poate fi atașată o funcție ce respectă formatul Bibliotecii de funcții. Pentru a își putea îndeplini toate funcțiile, el rulează trei corutine în paralel, implementate folosind asyincio [20]. Deși o implementare cu thread-uri ar fi fost preferabilă, Python nu suportă paralelismul datorită Global Interpreter Lock [68], așadar nu există niciun beneficiu peste implementarea folosind corutine.

Nodul de execuție comunică bidirecțional cu Centrul de comandă prin intermediul a două cozi comune, coada de Task-uri, respectiv coada de Status.

Un nod de execuție poate rula o singură funcție. Când este creat, el este considerat "disponibil". Acest lucru înseamnă că el poate primi o funcție. O dată ce primește o funcție, el devine "ocupat". În acest mod de operare, el începe să primească și să acumuleze date de input din rețea. Când nodul detectează că a primit input de la toți dependenții săi, rulează codul funcției care i-a fost asignat, obține rezultatele procesării, și trimite output-ul mai departe.

Acest proces este controlat de corutinele pe care le vom prezenta în continuare:

Corutina de Heartbeat

Un Heartbeat [61] este un semnal pe care un nod aflat într-o rețea îl transmite periodic pentru a anunța faptul că este funcțional. Toate nodurile din rețea transmit periodic, la un interval de 10 secunde, un astfel de mesaj, pe o coada comună, numită în continuare coada de Status. În acest mesaj este trimis totodată status-ul nodului (dacă este liber sau ocupat), pentru a ține evidența gradului de ocupare din rețea, respectiv timestamp-ul la care a fost trimis mesajul. Centrul de comandă consumă coada de Status, și ține evidența într-o colecție de Mongo, a gradului de ocupare din rețea.

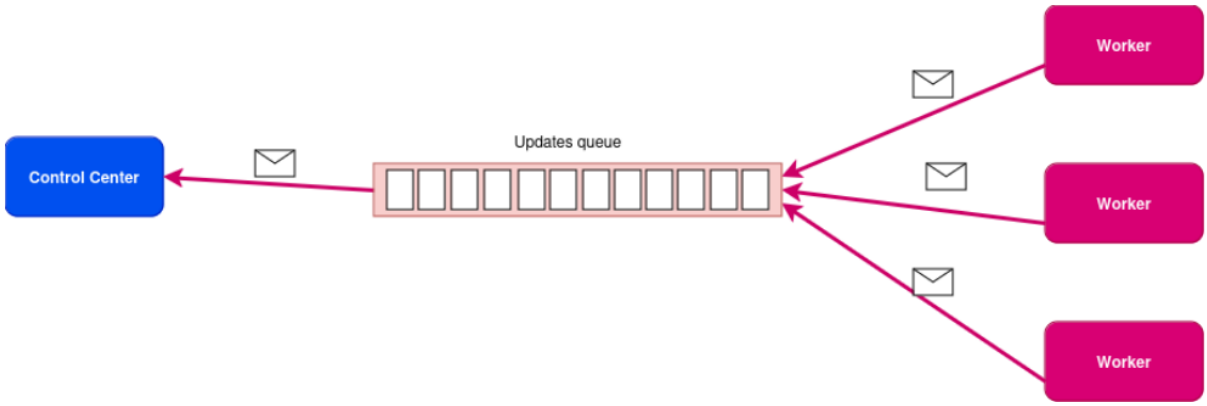

Figura 5.7: Nodurile de execuție trimit la fiecare 10 secunde, pe o coadă comună de Updates, status-ul lor. Centrul de comandă ține evidența nodurilor și a status-urilor. Dacă un nod nu mai trimite niciun mesaj timp de 30 de secunde, este considerat ca fiind blocat, și este șters din baza de date.

Corutina de Task

Aceasta corutină consumă din coada de RabbitMQ denumită Tasks. Coada este comună cu toate celelalte Noduri de execuție. În sistemul nostru, un Task poate fi interpretat ca o comandă de forma ”ia această reprezentare a unui Nod, conținând arcele de intrare, respectiv ieșire, și funcția ce trebuie rulată, și transformă-l într-un Nod de execuție în rețea”.[42]

Atunci când Centrul de comandă primește un graf pe care să-l transforme într-o rețea live, el pune câte un mesaj pe coada de task-uri pentru fiecare nod din reprezentare. Nodurile de execuție consumă de pe coada de task-uri aceste comenzi. Ele sunt distribuite de către RabbitMQ în pattern-ul Round Robin [69]. Dacă un nod de execuție primește un Task, dar el este deja ocupat, acesta refuză mesajul, iar RabbitMQ știe să îl trimită la următorul consumator, până când găsește unul disponibil.

Atunci când un Nod de execuție primește un Task iar el nu mai are alt Task deja, el îl va prelua, transformându-se într-un nod activ. Acest procedeu are loc în trei pași [42]:

1. **Primește** funcția utilizatorului și adaug-o în environment-ul local apelând funcția exec() din Python cu un context gol local[21].

2. **Setează** dinamic, pentru nod, o coadă de input, și un topic de output, în RabbitMQ, pe exchange-ul[5] comun tuturor nodurilor.

3. **Pornește** corutina care consumă din coada de input, agregă conținutul, rulează funcția dată de utilizator, și publică rezultate pe topic-ul de output, prezentată în secțiunea următoare.

Implementarea din acest proof-of-concept a pasului de integrare a codului oferit de către utilizator prezintă o problemă de securitate evidentă, mitigată parțial de rularea în

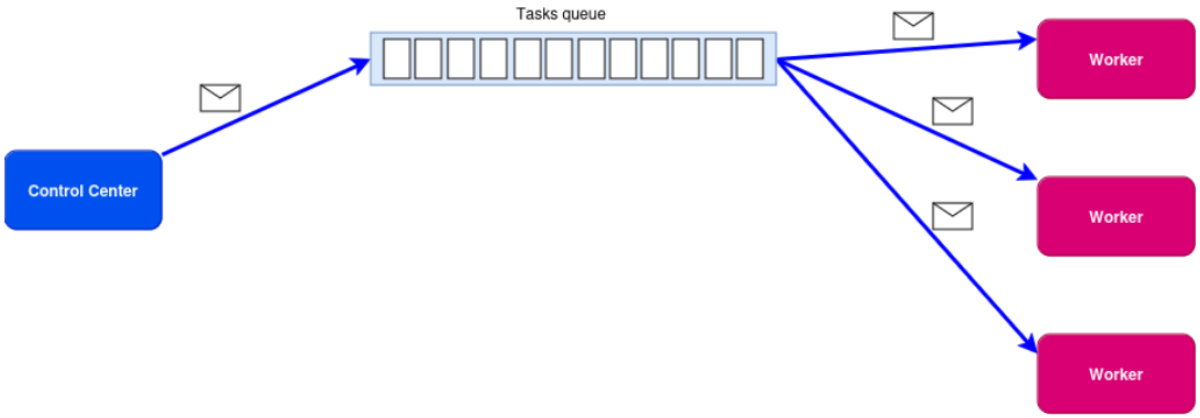

Figura 5.8: Atunci când un graf este deployed de către utilizator, pentru fiecare nod din graf se va transmite câte un mesaj pe coada de Tasks, conținând informațiile necesare pentru a putea fi instanțiat. Mesajele de pe coada de Tasks sunt consumate de către Workerii care sunt liberi (nu rulează deja o altă funcție). Mesajele sunt distribuite în pattern-ul Round Robin [69].

container. Există un număr de măsuri ce vor trebui luate pentru rularea în producție, cea mai importantă fiind rularea codului într-un proces separat, folosind RestrictedPython[22], un utilitar ce permite rularea codului de Python într-un Trusted Execution Environment [39], pentru a limita accesul codului la funcții ce ar putea periclita integritatea sistemului.

Corutina de Execuție

Această corutină, pornită în urma primirii unui Task, se ocupă de rularea efectivă a codului utilizatorului. Pentru a îndeplini acest obiectiv, ea consumă Input-ul primit pe coada sa, și îl acumulează într-un dicționar. Acumularea are loc deoarece o funcție poate depinde cu părți din output-ul mai multor alte funcții. Pentru a putea rula, este nevoie ca toate nodurile părinte ale nodului curent să fi terminat de procesat, și trimis output-ul lor, către funcția curentă. Deoarece procesarea are loc în batch-uri pentru a garanta coerența în procesare (toți parametrii de input să corespunda aceluiași batch), indexarea în dicționar este făcută, la rândul ei, în primul rand după batch_id, și mai apoi, după id-ul nodului părinte.

Rezultatele efective ale procesării nodurilor sunt cached, pentru a nu risca pierderea de date în cazul în care un nod trebuie repornit. Maparea din memorie ține doar evidența id-urilor de unde trebuie preluate datele.

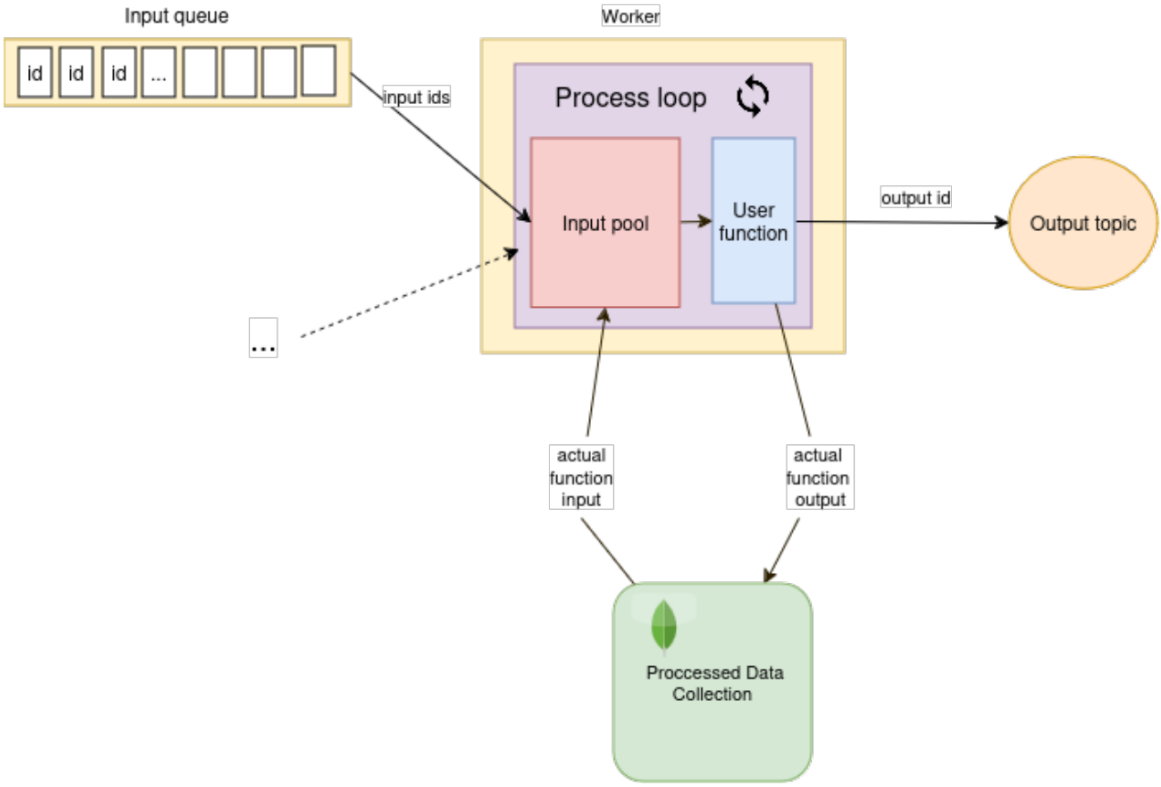

Figura 5.9: Diagrama care descrie procesul de prelucrare și acumulare a datelor de intrare în Worker.[42] În coada sa de Input, Worker-ul primește id-urile rezultatelor procesării funcțiilor de care worker-ul depinde. Ele sunt acumulate într-un dicționar care ține evidența batch-ului, și a numărului de inputuri primit. Când toate nodurile de care depinde workerul au trimis output-ul calculat de ele pentru acel batch, mesajele sunt "acknowledged" (pentru a dispărea din coada de RabbitMQ), datele efective sunt preluate din Mongo, funcția din Worker este rulată și rezultatul rulării este trimis pe topic-ul corespunzător. Dacă Worker-ul se blochează din orice motiv și trebuie repornit, datele nu sunt pierdute, iar la restartare, dicționarul se vă regenera cu input-ul neprocesat de pe coada sa.

Când maparea detectează că pentru un anume batch, toate funcțiile părinte au trimis datele, se rulează procesul de compunere a input-ului funcției curente, cu pașii următori:

1. Se obțin din colecția Mongo output-urile funcțiilor marcate în dicționarul de acumulare, pentru batch-ul curent.

2. Se creează un obiect nou tip Input, în care se va compune progresiv input-ul funcției.

3. Se iterează prin lista de arce ale nodurilor. Se iau în calcul doar arcele de intrare. Dacă există un arc de la nodul A la nodul B, înseamnă că o variabila din output-ul nodului A este folosită ca input în nodul B. Așadar, pentru a forma input-ul complet

al nodului B, vă trebui să luam valoarea din A și să o setăm în input-ul lui B, în variabila corespunzătoare.

4. Alternativ, operația de mai sus poate fi văzută ca pe o copiere și redenumire a valorii din output-ul funcției A, care contribuie la input-ul funcției B.

5. Când am acumulat tot input-ul, putem apela funcția.

6. Rezultatul funcției (output-ul său) este publicat pe topic-ul nodului.

7. Nodurile de execuție care depind de output-ul funcției curente, au coada de intrare subscrisă la topic-ul de output al funcției de care depind, prin metoda fanout[4]. Astfel, datele se propagă prin sistem.

Pentru a putea păstra intactă structura Output-urilor rezultate prin rularea funcțiilor, și a îndeplini funcția de caching, output-urile sunt întâi convertite în dicționare și apoi sunt "pickled" (serializare prin transformare într-un stream de bytes) folosind biblioteca dill[48]. Operația inversă are loc înainte de extragerea câmpurilor și formarea input-ului nou.

5.4.2 Centrul de comandă

Centrul de comandă este entitatea ce coordonează rețeaua de noduri. Ea ține evidența nodurilor existente în prezent în rețea într-o colecție Mongo. Detaliile reținute de această colecție sunt statutul nodului (dacă este liber sau lucrează), respectiv starea nodului (funcțional/blocat). [42]

Centrul de comandă rulează într-un container Docker privilegiat, având astfel dreptul de a adăuga și elimina alte containere. Centrul de comandă nu interacționează în niciun moment cu funcțiile scrise de către utilizatori, pentru a preveni eventualele riscuri din punct de vedere al securității.

Entrypoint-ul din Centrul de comandă este o metodă RPC numită graph.golive() ce poate fi apelată din frontend cu graful care trebuie să fie deployed. Ea extrage din obiectul grafului informațiile necesare pentru deployment, eliminând alte informații folosite la randarea în pagină, apelează metoda de scalare dacă este cazul, și pune pe coada de task-uri, câte un mesaj pentru fiecare funcție din reprezentare, care trebuie legată la un nod de execuție.

```
1
2   'graph.golive'(graph) {
3       console.log("Going live....")
4       Meteor.call("graph.updateStatus", graph._id, "loading")
5
```

```
6         const availableMachines = Meteor.call("machines.
              getAvailableCount")
7
8         console.log("[Meteor] Available machines:", availableMachines
              )
9         // start more machines if needed
10        if (graph.data.nodes.length > availableMachines) {
11            console.log("[Meteor] Not enough machines available.
                  Scaling up!")
12            Meteor.call("machines.scaleup", graph.data.nodes.length)
13        }
14        /** Edges have a ton of info in react-flow, we only need
              these fields for the machines to work */
15        const formatSourceTargetEdge = (edge => {
16
17            const [sourceFunction, sourceTitle, sourceType] = edge.
                  sourceHandle.split('.')
18            const [targetFunction, targetTitle, targetType] = edge.
                  targetHandle.split('.')
19            return  {
20                sourceArgument: {
21                    nodeId: edge.source,
22                    functionId: sourceFunction,
23                    name: sourceTitle,
24                    datatype: sourceType
25                },
26                targetArgument: {
27                    nodeId: edge.target,
28                    functionId: targetFunction,
29                    name: targetTitle,
30                    datatype: targetType
31                }
32            }
33
34        })
35
36        /** queue nodes to be picked up by machines */
37        /** Extracts data necessary for machine to run from node,
              filters out rendering info */
38
39        for (let node of graph.data.nodes) {
```

```
            const nodeInfo = {
                graphId: graph._id,
                nodeId: node.id,
                functionId: node.data._id,
                userId: node.data.userId,
                code: node.data.code,
                name: node.data.name,
                inputEdges:graph.data.edges
                    .filter(edge => edge.target === node.id)
                    .map(edge => formatSourceTargetEdge(edge)),
                outputEdges:graph.data.edges
                    .filter(edge => edge.source === node.id)
                    .map(edge => formatSourceTargetEdge(edge))
            }
            Meteor.callAsync("machines.bindRequest", nodeInfo)
            // console.log("\n")
            Meteor.call("graph.updateStatus", graph._id, "online")

        }
    }
```

5.4.3 Strategia de scalare

În implementarea curentă, scalarea nodurilor are loc on-demand. Atunci când utilizatorul cere deployment-ul unui graf nou, centrul de comandă calculează numărul de noduri libere. Dacă numărul este insuficient, pornește un număr de noduri astfel încât să fie suficiente pentru graful cerut, plus un offset. Offset-ul este momentan fixat în mod arbitrar cu valoarea unei constante. O îmbunătățire a sistemului ar putea fi alegerea unei strategii de scalare bazată pe un offset variabil, în funcție de parametrii din sistem (număr de request-uri care au loc pe secundă, numărul mediu de noduri dintr-o rețea, etc.)

Nodurile sunt curățate periodic după o perioadă de inactivitate îndelungată.

În implementarea curentă, centrul de comandă este o parte din backend-ul scris în Meteor.js, dar ar putea fi decuplat de acesta într-o iterație viitoare.

5.4.4 Strategia de conectare dintre noduri

O problemă interesantă din punct de vedere tehnic este procesul prin care este făcută conectarea dintre noduri în mod dinamic, prin intermediul de cozi și topicuri de RabbitMQ [56]. Ceea ce este neobișnuit aici este faptul că entitățile de tip nod de execuție trebuie conectate între ele în mod arbitrar și configurabil.

Pentru a face acest lucru, este nevoie să detaliem puțin despre filosofia de funcționare a RabbitMQ.

RabbitMQ stabilește exchange-uri, componente care primesc mesaje și le rutează în funcție de strategia setată. Câteva exemple de strategii de rutare sunt Direct, Fanout, Header [5]. În cazul nostru, strategia de rutare folosita în aplicație este Topic [50]. Aceasta este cea mai flexibila dintre toate strategiile, putând fi configurată astfel încât să se comporte atât ca Fanout, cât și că Direct, prin intermediul topicurilor, numite și chei de rutare (routing keys)[**rmq-routing-keys**]. Spre deosebire de Direct exchange, unde mesajele cu un anumit routing key sunt direcționate tuturor cozilor care au un binding key identic atașat, exchange-urile tip Topic permit configurarea unor chei de rutare compuse din mai multe părți, separate prin caracterul punct, iar cozile pot fi configurate să aibă binding keys (chei de acceptare, ce funcționează precum niște măști), cu parametri speciali, precum *, ce funcționează precum un wildcard, sau #, ce permite acceptarea mesajelor cu una sau mai multe subchei. [**rmq-routing-keys**].

Pentru a genera sisteme de cozi de mesaje, chei de rutare, și chei de binding, care să fie configurabile dinamic, am impus următoarele principii în rețea:

1. Fiecare nod de execuție consumă mesaje de pe propria sa coadă unică. Coada este generată atunci când nodul primește o funcție.

2. Fiecare nod de execuție își publică output-ul, rezultatul rulării funcției sale, pe un topic unic, generat pentru funcția respectivă, ce respectă forma *[id-graf].[id-nod].[id-functie]*

3. Coada nodului de execuție este setată cu un număr de chei de rutare egal cu numărul nodurilor de care depinde pentru a își forma input-ul. Formatul cheilor de rutare este *[id-graf].[id-nod].[id-functie].#* Caracterul # semnifică faptul că RabbitMQ știe că mesajele trimise pe topicul *[id-graf].[id-nod].[id-functie]* să fie copiate și puse pe toate cozile care respectă pattern-ul. Astfel, rezultatul rulării unei funcții este trimis către toate nodurile care au nevoie de (măcar o parte din) rezultat pentru a rula la rândul lor.

4. Cozile sunt efemere, se generează la deployment, respectiv se șterg automat atunci când graful este coborât.

Aceste principii permit generarea rețelelor de noduri de execuție care comunică prin RabbitMQ.

Figura 5.11: Nodurile de Start și Stop au funcționalități speciale, nefiind generate de utilizator. Ele reprezintă capetele grafului computațional. Fișiere cu input în format CSV pot fi încărcate în nodul de Start, iar rezultatul procesării, acumulat, poate fi descărcat din nodul de Stop.

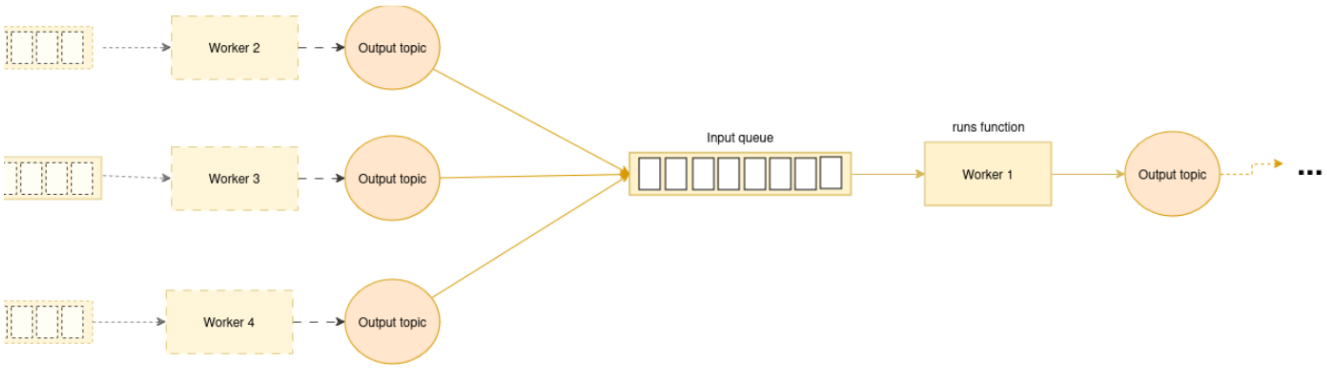

Figura 5.10: Fiecare Worker citește de pe coada proprie și trimite mesaje pe topic-ul său propriu. În pasul de deployment, atunci când nodurile sunt create, pentru un nod B care are nevoie de output-ul nodului A pentru a putea funcționa, (deci există muchie orientată de la A la B), pasul de deployment vă seta coada de intrare a nodului B să primească mesajele de pe topic-ul nodului A. Prin acest pas, se obține rețeaua de execuție. În practică, pe cozile de RabbitMQ se trimit doar id-uri, rezultatele procesărilor intermediare fiind salvate și încărcate din Mongo, pentru persistență în cazul în care un nod trebuie reîncărcat.

5.5 Încărcarea datelor în sistem

Odată ce graful a fost deployed, utilizatorul are opțiunea de a încărca fișiere CSV conținând datele sale prin intermediul interfeței grafice. Nodul de Start conține un câmp de Upload ce permite încărcarea acestor fișiere. Conținutul fișierelor este spart în linii, acestea fiind trimise prin graf. Rezultatul este acumulat și poate fi descărcat din nodul de Stop.

Capitolul 6

Concluzii

În această disertație, am construit un proof-of-concept pentru o platformă software-as-a-service, care răspunde nevoilor de scalabilitate, ușurință de utilizare, și reutilizare de cod, ale grupului țintă ales, data scientists/data analysts. Prin efectuarea unei analize de piață, în capitolul al doilea, am identificat o nevoie nesatisfăcută în cadrul grupului țintă: nevoia de a își putea scala proiectele cu ușurință, ideal, fără a fi nevoie de o echipă în plus care să construiască infrastructura necesară. Astfel, am început să compunem ideea și planul pentru o platformă care să ofere infrastructura necesară pentru crearea și întreținerea de fluxuri de date, permițând utilizatorilor să își aducă propriul cod (principiul "bring your own code"[52]), sau să folosească resursele oferite de către platformă, încapsulate sub formă de funcții inter-conectabile, în timp ce platforma furnizează infrastructura necesară.

Am demonstrat că este posibil să dezvoltăm o soluție care permite conectarea funcțiilor și crearea de pipeline-uri într-un mod facil și scalabil. Am ales această abordare pentru a permite utilizatorilor să își genereze infrastructura necesară pentru un proiect, direct din interfața grafică. Astfel utilizatorii se pot concentra pe dezvoltarea funcțiilor proprii, având la dispoziție beneficiile unei infrastructuri în cloud, fară să fie nevoie de ingineri software care să dezvolte infrastructura în paralel. Am arătat că o astfel de platformă poate funcționa și poate satisface nevoile utilizatorilor care doresc să creeze și să gestioneze fluxuri de date personalizate.

Pentru a arăta că ideea noastră este viabilă, am dezvoltat un proof-of-concept care arată cum pot fi conectate funcțiile într-un pipeline coerent, prin intermediul unei reprezentări grafice, denumită în contextul nostru graf computațional. În urma conectării, platforma noastră poate transforma această reprezentare într-o rețea de containere ce rulează codul utilizatorilor și comunică între ele. Această aplicație este un test preliminar al acestui concept și a confirmat că soluția propusă poate fi implementată cu succes. Aplicația arată nu doar posibilitatea tehnică, ci și cum poate simplifica procesul de procesare a unor volume mari de date, oferind utilizatorilor libertatea de a-și implementa propriile soluții într-un mediu robust și flexibil.

Pe lângă implementarea produsului, am stabilit și direcții pentru dezvoltarea unui

eventual plan de afaceri. Identificarea unei nișe care nu este complet acoperită de soluțiile existente pe piață oferă un avantaj competitiv și o oportunitate de a atrage o bază de clienți care au nevoie de flexibilitatea și puterea de procesare oferite de platforma noastră.

Am identificat, de asemenea, o serie de îmbunătățiri necesare pentru produsul nostru, pe parcursul implementării și testării acestuia. Printre aceste îmbunătățiri se numără creșterea securității datelor și îmbunătățirea experienței de utilizare printr-o interfață mai intuitivă și accesibilă. În eventualitatea în care vom continua cu dezvoltarea platformei, eventual în cadrul unul start-up, avem în vedere monitorizarea activă a tendințelor pieței și a cerințelor viitorilor clienți, pentru a adapta și îmbunătăți continuu platforma, maximizându-i astfel șansele de succes.

În concluzie, în cadrul acestei disertații, am demonstrat viabilitatea unei soluții low-code pentru crearea și gestionarea de flow-uri de date și am pus bazele unui potențial produs de tip IaaS. Am identificat o nișă în piață, care nu a mai fost explorată în acest fel, și am încercat să îi răspundem, prin efectuarea unui design de produs, iar mai apoi implementarea sa sub formă de proof-of-concept.

Bibliografie

[1] AltexSoft, „Apache Spark: Pros and Cons”, în *AltexSoft Blog* (2024), Accessed: 2024-08-12, URL: https://www.altexsoft.com/blog/apache-spark-pros-cons/.

[2] Atlassian, *Microservices vs Monolith*, Accesat: 2024-08-12, 2024, URL: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith.

[3] Adam SZ Belloum, Spiros Koulouzis, Tomasz Wiktorski și Andrea Manieri, „Bridging the demand and the offer in data science”, în *Concurrency and Computation: Practice and Experience* 31.17 (2019), e5200.

[4] CloudAMQP, „RabbitMQ Fanout Exchange Explained”, în *CloudAMQP Blog* (2024), Accesat: 2024-08-12, URL: https://www.cloudamqp.com/blog/rabbitmq-fanout-exchange-explained.html.

[5] CloudAMQP, „RabbitMQ for Beginners: Exchanges, Routing Keys, and Bindings”, în *CloudAMQP Blog* (2024), Accesat: 2024-08-12, URL: https://www.cloudamqp.com/blog/part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html.

[6] CloudAMQP, „Why Use RabbitMQ in a Microservice Architecture?”, în *CloudAMQP Blog* (2024), Accesat: 2024-08-12, URL: https://www.cloudamqp.com/blog/why-use-rabbitmq-in-a-microservice-architecture.html.

[7] AW Club, „R vs Python: Which is More Popular?”, în *AW Club Blog* (2024), Accesat: 2024-08-12, URL: https://aw.club/global/en/blog/r-vs-python.

[8] Code or No Code, „No-Code/Low-Code Data Science”, în *Code or No Code Blog* (2024), Accessed: 2024-08-12, URL: https://codeornocode.com/no-code/low-code-data-science/.

[9] CodeCut, „6 Common Mistakes to Avoid in Data Science Code”, în *CodeCut Blog* (2024), Accessed: 2024-08-12, URL: https://codecut.ai/6-common-mistakes-to-avoid-in-data-science-code/.

[10] DataCamp, „How to Overcome Challenges When Scaling Data Science Projects”, în *DataCamp Blog* (2024), Accessed: 2024-08-12, URL: https://www.datacamp.com/blog/how-to-overcome-challenges-when-scaling-data-science-projects.

[11] DataStax, „SQL vs NoSQL: Pros and Cons”, în *DataStax Blog* (2024), Accesat: 2024-08-12, URL: https://www.datastax.com/blog/sql-vs-nosql-pros-cons.

[12] Inc. Docker, *Docker Compose Documentation*, Accesat: 2024-08-12, 2024, URL: https://docs.docker.com/compose/.

[13] Inc. Docker, *Docker: Empowering Developers to Build, Share, and Run Applications*, Accesat: 2024-08-12, 2024, URL: https://www.docker.com/.

[14] Inc. Docker, *What is a Container?*, Accesat: 2024-08-12, 2024, URL: https://www.docker.com/resources/what-container/.

[15] Docker Hub, *Apache Datalab Docker Image*, https://hub.docker.com/r/apache/datalab, Accessed: 2024-08-12, 2024.

[16] Soham Dutta, *DevOps vs DataOps*, https://www.sprinkledata.com/blogs/devops-vs-dataops, Accesat: 2024-08-14.

[17] Exela Technologies, „The Role of Data Science in Business Decision Making”, în *Exela Technologies Blog* (2024), Accessed: 2024-08-12, URL: https://www.exelatech.com/blog/role-data-science-business-decision-making.

[18] React Flow, *React Flow: An Open-Source Library for Building Node-Based UIs*, Accesat: 2024-08-12, 2024, URL: https://reactflow.dev/.

[19] React Flow, *xyflow: The Official Repository for React Flow*, Accesat: 2024-08-12, 2024, URL: https://github.com/xyflow/xyflow.

[20] Python Software Foundation, *Asynchronous Tasks and Coroutines*, Accesat: 2024-08-12, 2024, URL: https://docs.python.org/3/library/asyncio-task.html.

[21] Python Software Foundation, *The exec() Function*, Accesat: 2024-08-12, 2024, URL: https://docs.python.org/3/library/functions.html#exec.

[22] Zope Foundation, *RestrictedPython*, Accesat: 2024-08-12, 2024, URL: https://github.com/zopefoundation/RestrictedPython.

[23] Inc. GitHub, *GitHub Codespaces*, Accesat: 2024-08-12, 2024, URL: https://github.com/features/codespaces.

[24] Inc. GitHub, *GitHub: Where the World Builds Software*, Accesat: 2024-08-12, 2024, URL: https://github.com/.

[25] Inc. GitHub, *Managing Your Personal Access Tokens*, Accesat: 2024-08-12, 2024, URL: https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens.

[26] Ming Hui Goh, „What is Data Analysis Process?”, în *Medium* (2024), Accessed: 2024-08-12, URL: https://medium.com/@gohminghui88/what-is-data-analysis-process-84864779eb5.

[27] Meteor Development Group, *Data Loading: Publications*, Accesat: 2024-08-12, 2024, URL: https://guide.meteor.com/data-loading.

[28] Meteor Development Group, „Introducing DDP", în *Meteor Blog* (2023), Accesat: 2024-08-12, URL: https://blog.meteor.com/introducing-ddp-6b40c6aff27d.

[29] Meteor Development Group, „Introducing useTracker: React Hooks for Meteor", în *Meteor Blog* (2024), Accesat: 2024-08-12, URL: https://blog.meteor.com/introducing-usetracker-react-hooks-for-meteor-cb00c16d6222.

[30] Meteor Development Group, *Meteor: The JavaScript Application Platform*, Accesat: 2024-08-12, 2024, URL: https://www.meteor.com/.

[31] Meteor Development Group, *Meteor.js Publications API Documentation*, Accesat: 2024-08-12, 2024, URL: https://docs.meteor.com/api/pubsub.

[32] Meteor Development Group, „Optimistic UI with Meteor", în *Meteor Blog* (2024), Accesat: 2024-08-12, URL: https://blog.meteor.com/optimistic-ui-with-meteor-67b5a78c3fcf.

[33] IBM, *Data Science*, https://www.ibm.com/topics/data-science, Accessed: 2024-08-12, 2024.

[34] IBM, *Message Brokers*, Accesat: 2024-08-12, 2024, URL: https://www.ibm.com/topics/message-brokers.

[35] Kais, „JavaScript is Everywhere", în *DEV Community* (2024), Accesat: 2024-08-12, URL: https://dev.to/kais_blog/javascript-is-everywhere-5fo0.

[36] dbt Labs, *Directed Acyclic Graph (DAG)*, Accesat: 2024-08-12, 2024, URL: https://docs.getdbt.com/terms/dag.

[37] LinkedIn, *What Challenges Do Data Teams Face When Collaborating?*, https://www.linkedin.com/advice/1/what-challenges-do-data-teams-face-when-collaborating-lnnhe, Accessed: 2024-08-12, 2024.

[38] Microservices.io, *Microservices.io: Resources and Guidance for Microservices*, Accesat: 2024-08-12, 2024, URL: https://microservices.io/.

[39] Microsoft, *Trusted Execution Environment (TEE)*, Accesat: 2024-08-12, 2024, URL: https://learn.microsoft.com/en-us/azure/confidential-computing/trusted-execution-environment.

[40] MongoDB, *Sharding in MongoDB*, Accesat: 2024-08-12, 2024, URL: https://www.mongodb.com/docs/manual/sharding/.

[41] mWater, *Minimongo*, Accesat: 2024-08-12, 2024, URL: https://github.com/mWater/minimongo.

[42] Ciausu Nicoleta, *A Distributed, Online, Collaborative Platform for Data Analysis*, 2024.

[43] Kazim Ozkabada, *Choosing Between Monorepo and Multi-Repo Architectures in Software Development*, Accesat: 2024-08-12, 2024, URL: https://medium.com/@kazimozkabadayi/choosing-between-monorepo-and-multi-repo-architectures-in-software-development-5b9357334ed2.

[44] Posit, *RStudio Desktop: Download*, https://posit.co/download/rstudio-desktop/, Accessed: 2024-08-12, 2024.

[45] Project Jupyter, *Jupyter: Tools for Interactive Computing*, https://jupyter.org/, Accessed: 2024-08-12, 2024.

[46] Pydantic, *JSON Schema Generation*, Accesat: 2024-08-12, 2024, URL: https://docs.pydantic.dev/latest/concepts/json_schema/.

[47] Pydantic, *Pydantic Documentation*, Accesat: 2024-08-12, 2024, URL: https://docs.pydantic.dev/latest/.

[48] PyPI, *Dill: Serialization of Python Objects*, Accesat: 2024-08-12, 2024, URL: https://pypi.org/project/dill/.

[49] Quora, *Why is the Jupyter Notebook Not Suited for Larger Projects?*, https://www.quora.com/Why-is-the-Jupyter-Notebook-not-suited-for-larger-projects, Accessed: 2024-08-12, 2024.

[50] RabbitMQ, *RabbitMQ Tutorial: Tutorial Five - Topic Exchanges*, Accesat: 2024-08-12, 2024, URL: https://www.rabbitmq.com/tutorials/tutorial-five-python.

[51] *React*, https://react.dev/, Accesat: 2024-08-14.

[52] Resolve, „No-Code, Low-Code, and Bring-Your-Own-Code Platforms Explained", în *Resolve Blog* (2024), Accesat: 2024-08-12, URL: https://resolve.io/blog/no-code-low-code-bring-your-own-code-platforms-explained.

[53] Irakli Nadareishvili Ronnie Mitra, „Rightsizing Your Microservices: Finding Service Boundaries", în (2024), Accessed: 2024-08-12, URL: https://www.oreilly.com/library/view/microservices-up-and/9781492075448/ch04.html.

[54] Git SCM, *Git: Free and Open Source Distributed Version Control System*, Accesat: 2024-08-12, 2024, URL: https://git-scm.com/.

[55] Snyk, *Container Runs in Privileged Mode*, Accesat: 2024-08-12, 2024, URL: https://learn.snyk.io/lesson/container-runs-in-privileged-mode/.

[56] Pivotal Software, *RabbitMQ*, Accesat: 2024-08-12, 2024, URL: https://www.rabbitmq.com/.

[57] Cole Stryker, *What is a data pipeline?*, https://www.ibm.com/topics/data-pipeline, Accesat: 2024-08-14.

[58] VMware Tanzu, *Developer Experience*, Accesat: 2024-08-12, 2024, URL: https://tanzu.vmware.com/developer-experience.

[59] Petroc Taylor, *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025*, 2024.

[60] React Team, *Managing State*, Accesat: 2024-08-12, 2024, URL: https://react.dev/learn/managing-state.

[61] TechTarget, *Heartbeat*, Accesat: 2024-08-12, 2024, URL: https://www.techtarget.com/searchdatacenter/definition/Heartbeat.

[62] TechTarget, *Monolithic Architecture*, Accesat: 2024-08-12, 2024, URL: https://www.techtarget.com/whatis/definition/monolithic-architecture.

[63] TechTarget, *Remote Procedure Call (RPC)*, Accesat: 2024-08-12, 2024, URL: https://www.techtarget.com/searchapparchitecture/definition/Remote-Procedure-Call-RPC.

[64] Chakra UI, *Card Component Documentation*, Accesat: 2024-08-12, 2024, URL: https://v2.chakra-ui.com/docs/components/card.

[65] Chakra UI, *Chakra UI Documentation*, Accesat: 2024-08-12, 2024, URL: https://v2.chakra-ui.com/.

[66] Avinash Vagh, „React Ecosystem in 2024", în *DEV Community* (2024), Accesat: 2024-08-12, URL: https://dev.to/avinashvagh/react-ecosystem-in-2024-418k.

[67] Visier, „Overcoming the Challenges of Getting Clean Data", în *Visier Blog* (2024), Accessed: 2024-08-12, URL: https://www.visier.com/blog/overcoming-the-challenges-of-getting-clean-data/.

[68] Python Wiki, *Global Interpreter Lock (GIL)*, Accesat: 2024-08-12, 2024, URL: https://wiki.python.org/moin/GlobalInterpreterLock.

[69] Wikipedia, *Round-Robin Scheduling*, Accesat: 2024-08-12, 2024, URL: https://en.wikipedia.org/wiki/Round-robin_scheduling.