



UNIVERSITATEA DIN BUCUREŞTI



FACULTATEA
DE
MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA
BAZE DE DATE ȘI TEHNOLOGII SOFTWARE

LUCRARE DE DISERTAȚIE

COORDONATOR ȘTIINȚIFIC

Lect. dr. Sorina-Nicoleta Preduț

ABSOLVENT

Răzvan-Cristian Mocică

BUCUREŞTI

2025



UNIVERSITATEA DIN BUCUREŞTI

FACULTATEA
DE
MATEMATICĂ ŞI INFORMATICĂ



SPECIALIZAREA
BAZE DE DATE ŞI TEHNOLOGII SOFTWARE

Platformă Web Pe Cinci Niveluri

COORDONATOR ȘTIINȚIFIC

Lect. dr. Sorina-Nicoleta Preduț

ABSOLVENT

Răzvan-Cristian Mocică

BUCUREŞTI

2025

REZUMAT

Domeniul dezvoltării web joacă un rol esențial în ecosistemul digital modern, însă integrarea coerentă a tuturor componentelor necesare pentru construcția unei aplicații complete rămâne o provocare continuă. Lucrarea de față propune o arhitectură modulară, structurată pe cinci niveluri funcționale: *AI*, *Front-end*, *Back-end*, *Bază de date* și *Deployment*, cu scopul de a demonstra cum pot fi dezvoltate aplicații scalabile, reactive și extensibile în contextul unei infrastructuri distribuite.

Pentru fiecare nivel au fost alese tehnologii actuale, dar bine consolidate, fiind urmărită aplicarea unor principii moderne precum reactivitatea, asincronia, containerizarea și orchestrarea automată. Arhitectura este susținută de o implementare practică completă, care reflectă fidel concepțele teoretice propuse și aduce contribuții originale semnificative. Dintre acestea se remarcă: gestionarea avansată a cache-ului în mai multe straturi, un sistem de căutare semantică bazat pe embedding vectorial, un mecanism de notificări în timp real, precum și integrarea de componente *AI* pentru generarea automată de conținut și interogare naturală. Aceste soluții au fost dezvoltate specific pentru nevoile aplicației, în absența unor implementări standard care să răspundă cerințelor arhitecturii propuse.

Contribuția lucrării constă în validarea unui model arhitectural și conceptual complet, adaptabil și extensibil, care poate fi aplicat în diferite contexte de dezvoltare software, nu doar în cel web abordat în această lucrare, și care poate servi drept model de bună practică în proiectarea și implementarea sistemelor distribuite moderne, fiind susținut de implementări proprii relevante ce consolidează aplicabilitatea practică a soluției.

ABSTRACT

The web development domain plays an essential role in the modern digital ecosystem, yet the coherent integration of all components required to build a complete application remains an ongoing challenge. This paper proposes a modular architecture, structured into five functional levels: *AI*, *Front-end*, *Back-end*, *Database* and *Deployment*, with the aim of demonstrating how scalable, reactive, and extensible applications can be developed within a distributed infrastructure.

For each level, current well-established technologies have been selected, with an emphasis on applying modern principles such as reactivity, asynchrony, containerization and automated orchestration. The architecture is supported by a fully developed practical implementation that faithfully reflects the theoretical concepts proposed and offers significant original contributions. Noteworthy among these are an advanced multi-layer caching strategy, a semantic search system based on vector embeddings, a real-time notification mechanism and the integration of *AI* components for automatic content generation and natural-language querying. In the absence of standard implementations capable of addressing the requirements of the proposed architecture, these solutions were developed specifically to meet the application's needs.

The contribution of the paper consists of the validation of a complete, adaptable and extensible architectural and conceptual model, which can be applied in different software development contexts, not only the web context addressed in this paper, and which can serve as a best-practice model in the design and implementation of modern distributed systems, being supported by relevant in-house implementations that consolidate the practical applicability of the solution.

CUPRINS

INTRODUCERE.....	9
CAPITOLUL 1. TEHNOLOGII UTILIZATE	10
1.1. Nivelul Back-end și Baze De Date.....	10
1.1.1. Microservicii Back-end.....	10
1.1.2. Comunicarea Între Servicii	11
1.1.3. Baza De Date Pentru Caching.....	12
1.1.4. Baza De Date Principală	12
1.1.5. Baza De Date Pentru Media.....	13
1.1.6. Plăți	14
1.2. Nivelul Front-end.....	14
1.3. Nivelul AI	15
1.4. Nivelul Deployment.....	15
1.4.1 Server	15
1.4.2. CI/CD (Continuous Integration / Continuous Delivery).....	16
CAPITOLUL 2. ARHITECTURA APLICAȚIEI	16
CAPITOLUL 3. IMPLEMENTAREA	18
3.1 Caching	18
3.1.1. Caching Back-end	18
3.1.2. Cache și Request-uri Front-end	20
3.1.3. Caching Imagini În Ingress	21
3.2. Căutare Semantică În Baza De Date	22
3.3. Microserviciile Python.....	23
3.4. AI În NextJS	24
3.4.1. Chat Bot	24
3.4.2. Alte Utilizări AI	25
3.5. Internaționalizare	26
3.6. Autentificarea Utilizatorilor.....	26
3.7. Arhivare	27
3.8. Comunicarea Între Microserviciile Din Back-end	28
3.8.1. Comunicarea HTTP	29
3.8.2. Comunicarea Prin Intermediul RabbitMQ	30
3.9. Blocări Distribuite În Redis	31

3.10. Testare.....	31
CAPITOLUL 4. SCENARII DE UTILIZARE	32
4.1. Vizitator	32
4.2. Utilizator Autentificat	33
4.3. Utilizator Cu Drepturi De Postare.....	34
4.4. Utilizator De Tip Admin.....	36
CAPITOLUL 5. CONCLUZII.....	37
BIBLIOGRAFIE.....	39
ANEXE.....	42
Definiții	42
Completați	51
Figuri.....	63
Secvențe De Cod.....	77

LISTĂ DE FIGURI

Figura 1.1. Rulare pipeline din Github Actions.....	16
Figura 2.1. Diagrama deployment-ului în cloud-ul privat	17
Figura 3.1. Căutare semantică în lista de planuri.....	23
Figura 3.2. Moderare text în aplicație.....	24
Figura 3.3. Interacțiune cu chatbot-ul cu extragere de informații din baza de date.....	25
Figura 3.4. Căutare cu AI în corpul elementelor	25
Figura 3.5. Generare titlu cu AI.....	26
Figura 3.6. Live chat cu atenționare când partenerul scrie.	30
Figura 4.1. Diagrama scenarii de utilizare pentru vizitator.	33
Figura 4.2. Pagina de comenzi pentru utilizator.	33
Figura 4.3. Kanban utilizator.....	34
Figura 4.4. Diagrama scenarii de utilizare pentru utilizatorul autentificat.	34
Figura 4.5. Pop-up notificări.....	35
Figura 4.7. Diagrama scenarii de utilizare pentru utilizatorul cu drepturi de postare	35
Figura 4.8. Grafic de statistici comenzi în funcție de țări.....	36
Figura 4.9. Top utilizatori cu drepturi de postare	36
Figura 4.10. Diagrama de utilizare pentru utilizatorul cu drepturi de administrare	37
Figura Anexe-Definiții 1. Diagrama AMQP	42
Figura Anexe-Definiții 2. Arhitectura GridFS.....	43
Figura Anexe-Definiții 3. Chunking în GridFS	43
Figura Anexe-Definiții 4. Exemplu de JWT	50
Figura Anexe-Completări 1. Rezultat comparație	51
Figura Anexe-Completări 3. Rezultat comparație	52
Figura Anexe-Completări 4. Rezultat comparație	52
Figura Anexe-Completări 5. Rezultat comparație	52
Figura Anexe-Completări 6. Rezultat comparație	53
Figura Anexe-Completări 7. Rezultat comparație	53
Figura Anexe-Completări 8. Rezultat comparație	53
Figura Anexe-Completări 9. Rezultat comparație	54
Figura Anexe-Completări 10. Rezultat comparație	54
Figura Anexe-Completări 11. Dashboard Grafana pentru un microserviciu de Spring	55
Figura Anexe-Completări 12. Dashboard Grafana pentru un nod din cluster	55

Figura Anexe-Completări 13. Dashboard Grafana pentru operatorul de CNPG	55
Figura Anexe-Completări 14. Servicii în Prometheus	56
Figura Anexe-Completări 15. Log-uri în Loki pentru un serviciu.....	56
Figura Anexe-Completări 16. Cerere urmărită în Tempo.....	56
Figura Anexe-Completări 17. Statistici în Umami pentru front-end sumar relativ	57
Figura Anexe-Completări 18. Statistici în Umami pentru front-end sumar general.....	58
Figura Anexe-Completări 19. Plan de execuție pentru o cerere care utilizează index GIN	61
Figura Anexe-Figuri 1. Diagrama de flux privind accesarea cache-ului în back-end	63
Figura Anexe-Figuri 2. Diagrama de flux privind invalidarea cache-ului în back-end.....	63
Figura Anexe-Figuri 3. Diagrama de flux privind trimiterea de notificări pentru invalidarea cache-ului în back-end	63
Figura Anexe-Figuri 4. Diagrama de flux privind mecanismul de caching în front-end	64
Figura Anexe-Figuri 5. Diagrama de flux privind mecanismul de căutare semantică în aplicație	64
Figura Anexe-Figuri 6. Diagrama de interacțiune a utilizatorului cu chat bot-ul în aplicație	65
Figura Anexe-Figuri 7. Diagrama de flux privind mecanismul de arhivare în aplicație	65
Figura Anexe-Figuri 8. Diagrama generală de comunicare între microserviciile din back-end	66
Figura Anexe-Figuri 9. Rezultatele acoperirii codului de către teste în modulul de bază.....	66
Figura Anexe-Figuri 10. Rezultatele acoperirii codului de către teste în modulul de caching	67
Figura Anexe-Figuri 11. Rezultatele acoperirii codului de către teste în modulul de căutare semantică	67
Figura Anexe-Figuri 12. Dashboard sumar pentru administrator	67
Figura Anexe-Figuri 13. Top utilizatori pentru administrator	68
Figura Anexe-Figuri 14. Top utilizatori cu drepturi de postare pentru administrator	68
Figura Anexe-Figuri 15. Creare e-mail din partea companiei	69
Figura Anexe-Figuri 16. Vânzări lunare pentru administrator-partea I.....	70
Figura Anexe-Figuri 17. Vânzări lunare pentru administrator-partea a II-a.....	71
Figura Anexe-Figuri 18. Vânzări lunare pentru administrator-partea a III-a	71
Figura Anexe-Figuri 19. Arhivare planuri pentru administrator	72
Figura Anexe-Figuri 20. Formular de creare rețetă-partea I.....	72
Figura Anexe-Figuri 21. Formular de creare rețetă-partea a II-a.....	73

Figura Anexe-Figuri 22. Formular de creare rețetă-partea a III-a	73
Figura Anexe-Figuri 23. Pagina pentru administrarea postărilor-partea I.....	74
Figura Anexe-Figuri 24. Pagina pentru administrarea postărilor-partea a II-a.....	74
Figura Anexe-Figuri 25. Căutare în site	75
Figura Anexe-Figuri 26. Calendarul zilelor urmărite-partea I.....	75
Figura Anexe-Figuri 27. Calendarul zilelor urmărite-partea a II-a.....	75
Figura Anexe-Figuri 28. Exemplu internaționalizare site limba engleză	76
Figura Anexe-Figuri 29. Exemplu internaționalizare site limba română	76

SECVENȚE DE COD

Extras Cod-Sursă Anexe-Completări 1. Gestionarea tabelei associative în cazul relațiilor one-to-many și many-to-many	60
Extras Cod-Sursă Anexe-Completări 2. Crearea unui index GIN pentru un array.....	61
Extras Cod-Sursă 1. Salvarea în cache-urile din back-end pentru un publisher cu mai mult de un element	78
Extras Cod-Sursă 2. Deduplicare de request-uri în front-end	81
Extras Cod-Sursă 3. Controlul ciclului de viață al listenere-lor RabbitMQ în serviciul de arhivare	84
Extras Cod-Sursă 4. Blocări distribuite în Redis	85
Extras Cod-Sursă 5. Test salvare publisher cu mai mult de un element în cache-ul din back-end când valoarea nu este prezentă în niciun cache.....	87
Extras Cod-Sursă 6. Interogarea bazei de date și procesarea stream-ului binar pentru un request de tip Range.....	88
Extras Cod-Sursă 7. Crearea pipeline-ului de difuzie folosind HuggingFace	89

INTRODUCERE

Domeniul web este unul vast și foarte popular întrucât este agnostic de device și platformă de operare, poate fi livrat sub forma *SaaS* (Software as a Service), este ușor de folosit pentru consumator, dar oferă și o gamă variată de tipuri de platforme pentru acesta. În prezent, majoritatea aplicațiilor web continuă să fie construite pe arhitecturi blocante, folosind framework-uri tradiționale precum *Spring MVC* în back-end și diverse biblioteci de tip *SPA* (Single Page Application) în front-end. Cu toate acestea, pe măsură ce cerințele legate de scalabilitate, concurență și eficiență în procesarea *I/O* devin tot mai importante, abordările reactive câștigă teren. În acest context, tehnologiile precum *Spring WebFlux* pentru back-end-ul reactiv și *NextJS* pentru front-end-ul modern permit dezvoltarea unor aplicații mai rapide și mai scalabile. Deși este considerat inovator în comparație cu soluțiile blocante dominante, acest model este doar adoptat parțial. De asemenea, în ceea ce privește infrastructura, se observă o tendință accentuată a dezvoltatorilor și companiilor de a evita implementarea și gestionarea unor soluții *Kubernetes* locale, preferând să utilizeze servicii de cloud gestionate, chiar și în cazurile în care cerințele aplicației nu justifică costurile ridicate ale acestora. Această dependență de cloud, adesea nejustificată, duce la risipă de resurse financiare, în timp ce soluții locale, eficiente și accesibile, precum *Microk8s*, ar putea oferi aceleași funcționalități pentru nevoile curente de dezvoltare la un cost redus.

Luând în considerare cele prezentate anterior, lucrarea de față are rolul de a crea o platformă web completă și modernă, care integrează și aderă la cele mai recente tendințe și standarde din domeniul web, folosind tehnologii actuale și paradigme noi, dar bine stabilite, evitând capcana de a urmări fiecare nou framework apărut.

Pentru realizarea acesteia, am structurat logica aplicației în cinci niveluri principale: *AI*, *Front-end*, *Back-end*, *Bază de date* și *Deployment*. Lucrarea urmărește să evidențieze și să dezvolte fluxuri complexe de integrare, atât în cadrul fiecărui nivel, cât și între niveluri, punând accent pe coerență, scalabilitate și eliminarea de „boilerplate”, scopul acestoria depășind o simplă implementare tehnică.

Pentru a ilustra conceptele prezentate, este necesară și o implementare concretă. Astfel, am ales să dezvolt un site wellness axat pe nutriție, având ca scop principal comercializarea de planuri nutriționale și integrarea tuturor funcționalităților specifice unui site de e-commerce. În acest domeniu, pe lângă numeroasele resurse video disponibile pe platforme precum *YouTube*, există și două aplicații relevante care pot fi considerate competitoare: *RPSstrength* și *MacroFactor*, ambele dezvoltate de figuri cunoscute din

industria fitness-ului. *RPStrength* se concentrează în principal pe planuri de antrenament, dar are și o mică secțiune pentru nutriție, în timp ce *MacroFactor* funcționează ca o aplicație mobilă pentru urmărirea caloriilor și a progresului nutrițional, oferind totodată postări și sugestii alimentare, într-un format mai limitat.

Aplicația de față se diferențiază printr-un avantaj esențial: diversitatea conținutului generat de comunitate. Permitând utilizatorilor acreditați să posteze rețete și informații proprii, aceasta extinde semnificativ spectrul de expertiză și perspectivele disponibile, în contrast cu aplicațiile concurente, unde conținutul este strict controlat de echipa de dezvoltare sau de fondatori. Totuși, un dezavantaj important rămâne lipsa notorietății, întrucât aplicațiile concurente beneficiază deja de o bază solidă de utilizatori și de recunoaștere prin intermediul creatorilor lor cunoscuți. Un element distinctiv care poate atrage utilizatori noi este integrarea de componente *AI*, care adaugă funcționalități moderne și interactive, fiind un aspect din ce în ce mai atractiv în rândul publicului interesat de tehnologie și personalizare.

Prezenta lucrare este structurată în patru capitole, urmate de concluzii și bibliografie. În primul capitol sunt prezentate tehnologiile utilizate, alături de argumentele care au stat la baza alegerilor făcute. Capitolul al doilea este dedicat descrierii arhitecturii aplicației, evidențiind complexitatea unui sistem distribuit. În capitolul al treilea este detaliată implementarea propriu-zisă, cu accent pe aspecte și concepte tehnice mai complexe și elementele considerate de interes. Ultimul capitol ilustrează scenariile de utilizare din cadrul aplicației demonstrative, evidențiind modul în care funcționalitățile sunt puse în practică din perspectiva utilizatorului.

CAPITOLUL 1. TEHNOLOGII UTILIZATE

1.1. Nivelul Back-end și Baze De Date

1.1.1. Microservicii Back-end

Aplicația este una web, iar aceasta se va baza pe API-uri de tip *REST* ([pentru detalii, Anexe, Definiții, REST](#)) în back-end, mai exact pe microservicii de acest tip. Am ales să implementez majoritatea acestora în *Spring Boot*, mai exact *Spring WebFlux* împreună cu paradigma programării reactive ([pentru detalii, Anexe, Definiții, Programare Reactivă](#)) și funcționale ([pentru detalii, Anexe, Definiții, Programare Funcțională](#)). Restul microserviciilor vor fi implementate în *Python* folosind *Flask* și au ca rol doar facilitarea interogării unor modele de *AI* ce rulează local.

Am optat pentru *WebFlux* în loc de clasicul *MVC* întrucât acesta este mai performant în cazul unui high-throughput, dar și scalează mai „logaritmic” în cazul traficului ridicat ([pentru detalii, Anexe, Completări, Comparatie Paradigme](#)). [50]

În perioada recentă, în limbajul *Java* au apărut *Virtual Threads*. Acestea sunt o caracteristică introdusă în *Java 21* prin *Project Loom*, care permite crearea unui număr foarte mare de fire de execuție ușoare, gestionate de *JVM*, nu direct de sistemul de operare. Acestea sunt ideale pentru aplicații cu multe operațiuni *I/O* blocante, cum ar fi serverele web sau aplicațiile care fac multe apeluri la baze de date, dar și în scenarii care necesită gestionarea unui număr mare de fire concurente, pentru rularea eficientă a mii sau milioane de sarcini simultane și ușoare din punct de vedere computațional. Acestea însă nu sunt recomandate pentru sarcini intensive pe *CPU*.

Așadar, thread-urile virtuale sunt excelente pentru aplicații *I/O*-intensive care necesită gestionarea unui număr mare de sarcini concurente, care sunt limitate de către *I/O*, nu de către aplicație în sine. [20] [36]

Întrucât pentru *Virtual Threads* scrierea codului este tot sincronă, similară cu cea din *Spring MVC*, o migrare la acestea este mult mai facilă decât una la *WebFlux*, în care codul este de tip funcțional. De aceea, de multe ori se ignoră *WebFlux*, însă acesta este încă mai rapid și mai eficient decât thread-urile virtuale. [36]

În contextul dezvoltării aplicațiilor moderne, în special a celor *I/O*-intensive, o soluție performantă și flexibilă constă în combinarea modelului reactiv oferit de *Spring WebFlux* cu thread-urile virtuale, pentru a gestiona eficient atât operațiunile non-blocante, cât și cele blocante. *WebFlux* asigură un flux de date asincron și scalabil în situațiile în care toate componentele sistemului suportă programarea reactivă. Totuși, în cazurile în care anumite biblioteci sau surse de date nu oferă suport reactiv și funcționează doar în mod blocant, firele virtuale pot fi utilizate ca un mecanism de integrare/adaptare. Acestea permit rularea codului sincron într-un mod eficient, fără a suprasatura resursele sistemului, păstrând astfel avantajele de performanță ale arhitecturii reactive, fără a compromite simplitatea sau compatibilitatea cu tehnologii mai vechi. Așadar, în aplicația de față am optat pentru combinarea celor două, nu alegerea oarbă doar a uneia dintre acestea.

1.1.2. Comunicarea Între Servicii

Comunicarea între microserviciile back-end-ului se realizează atât folosind *WebClient* din *Spring*, care acceptă și *HTTP* (Hypertext Transfer Protocol) reactiv, cât și *RabbitMQ* pentru a realiza comunicări asincrone atât prin cozi de mesaje, cât și prin *RPC* (Remote

Procedure Call) ([pentru detalii, Anexe, Definiții, RPC](#)). *RabbitMQ* este o alegere naturală în *Spring* deoarece el aderă la protocolul *AMQP* (Advanced Message Queuing Protocol), care este un standard ce permite interoperabilitate în schimbul de mesaje între sisteme, indiferent de furnizorul broker-ului, astfel aplicațiile care folosesc acest standard sunt complet agnoscibile de tipul de broker utilizat, atât timp cât implementează corespunzător standardul ([pentru detalii, Anexe, Definiții, AMQP](#)).

În realitate, comunicarea dintre back-end și front-end nu este doar unidirecțională. Pentru a permite o legătură bidirecțională, adiacent celei unidirecționale de tip *HTTP*, am folosit protocolul *WebSocket* ([pentru detalii, Anexe, Definiții, WebSocket](#)). Întrucât o comunicare de acest tip este complexă și necesită precizie ridicată, în aplicația prezentată a fost implementată astfel:

- Folosind protocolul *STOMP* (Simple/Streaming Text Oriented Message Protocol), care este un standard ce permite clienților din conexiuni bidirectionale să comunice prin intermediul unui broker, anume *RabbitMQ* în cazul de față. Aceasta din urmă este responsabil de a centraliza mesajele și de a asigura respectarea statefulness-ului (istoricului). [\[34\]](#)
- Folosind *WebSocket* nativ, dar cu *Redis Channels* pentru a asigura curățarea automată a mesajelor și o scalabilitate superioară.

1.1.3. Baza De Date Pentru Caching

Deoarece orice back-end care se dorește a fi la un standard înalt are nevoie de un mecanism de caching, în aplicația de față, pe lângă un prim „layer in memory” în fiecare instanță a unui microserviciu reactiv, există și o bază de date de caching distribuit, anume *Redis* (Remote Dictionary Server), mai specific fork-ul *KeyDB*. *Redis* este o bază de date in-memory, extrem de rapidă, ce funcționează pe structura „cheie:valoare” și este utilizată preponderent ca sistem de cache. [\[49\]](#) [\[22\]](#)

Am optat pentru *KeyDB*, deoarece un dezavantaj major al *Redis* este caracteristica de single-threaded, procesând o singură comandă la un moment dat, ceea ce limitează scalabilitatea. *KeyDB* este un înlocuitor compatibil cu *Redis* (drop-in replacement), dar cu suport multithread, permitând utilizarea mai eficientă a procesoarelor moderne. Pe lângă suportul pentru mai multe thread-uri, oferă optimizări suplimentare, cu performanțe I/O de până la trei ori mai rapide, chiar și pe un singur thread. În plus, există și un conector care acceptă paradigma reactivă pentru acest tip de bază de date. [\[22\]](#) [\[3\]](#)

1.1.4. Baza De Date Principală

Pentru baza de date principală a aplicației am optat pentru *PostgreSQL*, deoarece,

având o arhitectură bazată pe microservicii, denormalizările, mai ales a relațiilor *one-to-many* și *many-to-many*, sunt inevitabile și chiar uneori recomandate pentru a păstra independența microserviciilor ([pentru detalii, Anexe, Completări, Denormalizare În Postgre](#)), iar cum unul dintre principalele sale avantaje este suportul excepțional pentru date complexe și de tip *array*, fiind mult mai facil lucrul cu acestea decât în alte RDBMS-uri precum *Oracle* sau *MySQL*, alegerea sa a fost naturală. [\[2\]](#) [\[4\]](#)

Am optat pentru *Postgre*, pe de o parte, deoarece este unul dintre puținele baze de date care oferă un conector respectând paradigma reactivă, i.e. implementând protocolul *R2DBC* de conectare. Pe de altă parte, un alt avantaj major al acestei baze de date este extensia *PgVector*, care permite salvări și căutări de similaritate pentru vectori numerici. Aceasta din urmă este foarte importantă, încrănat în cadrul aplicației prezentate, și nu numai, nu mai este necesară încă o bază de date specializată precum *ElasticSearch* pentru a avea căutări semantice puternice, ci este de ajuns să căutăm direct în *Postgre*, făcând în prealabil embedding-ul necesar *query-ului* de interogare. Nu am ales *MongoDB* deoarece, deși în cadrul platformei sunt prezente denormalizări, structura elementelor este bine definită, iar astfel datele sunt structurate, ceea ce înseamnă că o bază de date *SQL* este o alegere bună, dar nu neapărat singura potrivită. [\[48\]](#)

În plus, folosind operatorul *CNPG* (Cloud Native Postgre) în *Kubernetes* (k8s) pentru rularea bazei de date, unul dintre avantajele importante ale acestuia este integrarea nativă cu *PgBouncer*, un „connection pooler” performant pentru *PostgreSQL*. Acesta gestionează eficient conexiunile la bază, prin reutilizarea și limitarea celor active simultan, aspect esențial într-o arhitectură cu multiple microservicii. Astfel, *PgBouncer* contribuie la îmbunătățirea performanței și scalabilității aplicației, reducând încărcarea serverului de baze de date și prevenind blocajele generate de conexiuni excesive. De asemenea, operatorul suportă nativ și replicarea bazei, fiind utilizată în aplicația demonstrativă, asigurând disponibilitate crescută și reziliență. [\[24\]](#)

1.1.5. Baza De Date Pentru Media

Deoarece, platforma va permite și încărcarea de imagini și videoclipuri, și am dorit pe de o parte o arhitectură self-contained, iar pe de altă parte am dorit să evit costurile ridicate ale unui serviciu *SaaS* de gestionare a media, precum *Cloudinary*, am optat pentru folosirea *MongoDB GridFS* ([pentru detalii, Anexe, Definiții, GridFS](#)). O caracteristică importantă a *MongoDB* este că acceptă în mod nativ conectivitate reactivă, fiind conceput inițial pentru programarea asincronă, în special în ecosistemul *JavaScript*. Astfel, există un conector

compatibil cu modelul reactiv *Spring WebFlux*, care permite integrarea ușoară cu arhitectura aplicației. În plus, pentru a eficientiza stocarea și manipularea fișierelor binare în contextul *GridFS*, am utilizat un *Helm* chart cu suport pentru *sharding*, ceea ce permite scalarea și distribuirea eficientă a datelor media în instanțele *MongoDB*.

1.1.6. Plăți

Pentru a permite plăți online în cadrul aplicației, am ales integrarea cu *Stripe* deoarece este un serviciu cu renume în domeniu, care permite rularea anumitor părți local, astfel diminuând dependența de alte servicii, oferind totodată simplitatea implementării.

1.2. Nivelul Front-end

Pentru nivelul front-end am ales framework-ul *React*, mai exact „supra-framework-ul” *NextJS*. Aceasta este un framework open-source care simplifică configurațiile și mai ales permite integrare facilă și cu *React Server Components*, i.e. cod ce rulează și pe server, nu doar pe client, ceea ce este foarte facil când se dorește integrarea unor pipeline-uri de *AI*, precum cele din aplicația de față, întrucât acestea nu necesită resurse computaționale ridicate și astfel se pot păstra, pe de o parte, dimensiunile reduse ale codului de *JS* pe client, iar pe de altă parte, securitatea aplicațiilor care se conectează la serviciile de *AI*. De asemenea, *NextJS* permite și crearea unei aplicații full-stack, eliminând necesitatea unui back-end separat, dar acest tip de aplicații nu este foarte scalabil și nu este recomandat în principal pentru platforme dinamice și care se doresc a fi puternic scalabile. Acest framework este facil, întrucât oferă integrare rapidă cu diverse necesități din domeniul web, precum: routing avansat al unei aplicații single page, randare concurrentă, optimizarea încărcării elementelor statice precum fonturi, imagini, integrare *TypeScript* pentru dezvoltatori, diverse framework-uri de *CSS* precum cel folosit de față, anume *Tailwind*. [\[45\]](#) [\[44\]](#)

Datorită flexibilității lui *NextJS*, tot în cadrul acestuia, pentru a facilita integrarea cu anumite modele și pipeline-uri de *AI*, am optat pentru librăria *LangChainJS*. Aceasta permite integrarea facilă a *LLM*-urilor (Large Language Model) în aplicații de *JS* și conectarea la diverse baze de date vectoriale, facilitând caching al răspunsurilor pentru a îmbunătăți performanțele și acuratețea generărilor modelelor alese.

Nu în ultimul rând, pentru a oferi o imagine clară și detaliată asupra statisticilor de vizitare și a modului în care utilizatorii interacționează cu site-ul, aplicația front-end a fost integrată cu platforma *Umami Analytics*. Aceasta este o alternativă pentru *Google Analytics*, open-source, care poate fi rulată local, care permite monitorizarea eficientă a traficului și

comportamentului utilizatorilor într-un mod anonimizat, fără utilizarea cookie-urilor externe, respectând astfel principiile de confidențialitate și protecție a datelor personale.

1.3. Nivelul AI

Deoarece în orice aplicație modernă *AI*-ul este imperativ, am decis să utilizez mai multe modele și paradigmă pentru a satisface aceste tendințe, folosind diverse librării și tool-uri precum *Ollama* și *HuggingFace*.

Un prim instrument este *Ollama*, care facilitează rularea direct pe mașina locală a modelor *LLM* și nu numai, eliminând necesitatea serviciilor cloud. Acesta oferă un mediu izolat care include toate componentele necesare pentru desfășurarea modelor *AI*, cum ar fi greutățile modelului, fișierele de configurare, dependențele necesare și configurarea automată a modelor încât să ruleze pe multiple *GPU*-uri. [\[47\]](#)

Am optat pentru *Ollama* ca mediu de rulare pentru modele *AI*, întrucât pune la dispoziție și un *API* de tip *REST* pentru a interoga aceste modele. Astfel, în cadrul aplicației de față vom rula în acest environment un model de embeddings, anume *BGE-M3* cu cuantizarea *fp16* pentru a genera embedding-uri de căutare semantică, și un model *LLM*, anume *PHI4-14b* cu cuantizarea *q4_K_M* pentru diferite necesități din aplicație.

Tot ca un tool *AI*, sau mai mult o librărie/framework, este *HuggingFace*. Acesta permite rularea mai multor modele de *AI* în *Python* și, mai ales, oferă template-uri deja preconfigurate pentru diverse task-uri prestabilite pentru *NLP*, *Computer Vision* etc. [\[43\]](#)

În cadrul aplicației de față, pe lângă microserviciile din *Spring Boot*, am implementat încă alte trei microservicii care au ca scop crearea unui mediu de interogare pentru trei tipuri de pipeline-uri din *HuggingFace*, anume: *Diffusion*, pentru generare de imagini, *Time Series Prediction*, pentru predicții temporale și *Text Classification*, pentru moderare de conținut. Aceste trei noi servicii au la bază framework-ul *Flask*, întrucât este un framework matur și minimal de *Python* pentru crearea de *API*-uri.

1.4. Nivelul Deployment

1.4.1 Server

Pentru deployment am optat pentru crearea unui cloud privat folosind două calculatoare mai vechi, de aproximativ zece ani. Acest cloud a fost implementat folosind *Kubernetes* (k8s), care este o platformă open-source dezvoltată inițial de *Google*, destinată automatizării implementării, scalării și gestionării aplicațiilor containerizate. [\[29\]](#)

Denumirile calculatoarelor care alcătuiesc cluster-ul sunt *ServerManager* și *ServerWorker*, având următoarele specificații:

- *ServerManager*: i7 6700k, 48GB RAM și două GTX 1070, având Ubuntu 22.04.5LTS
- *ServerWorker*: i7 6700k, 48GB RAM și GTX 970, având Ubuntu 22.04.5LTS

Deoarece instalarea și configurarea de la zero a unui environment de k8s este foarte anevoieasă, am optat pentru folosirea *Microk8s*, care oferă o distribuție completă de k8s, fiind „production ready”, realizată de către *Ubuntu*. Diferența dintre *Microk8s* și alte distribuții locale de k8s, precum *Minikube* sau *Kind*, constă în faptul că acesta este conceput pentru uz în medii de producție, nu doar pentru testare sau dezvoltare locală. Pentru clusterul de față, *ServerManager* este managerul, adică cel care ghidează dependențele necesare și nodurile/nodul worker, iar *ServerWorker* este nodul worker. Un alt avantaj major al *Microk8s* este suportul nativ pentru addon-uri, operatori și *Helm* chart-uri, care au venit în completare pentru a realiza un cloud privat cât mai realist. [\[25\] \(pentru detalii, Anexe, Completări, Principalele Elemente Suplimentare În Cluster\)](#)

1.4.2. CI/CD (Continuous Integration / Continuous Delivery)

Pentru source control am ales *GitHub*, întrucât este cea mai populară platformă de acest fel, iar pentru realizarea unui *CI/CD* care rulează automat testele și crearea imaginilor de *Docker* am ales *GitHub Actions*, deoarece se integrează imediat în flow-ul de source control. Un pipeline pentru un microserviciu de *Spring* se poate observa în [Figura 1.1.](#):



Figura 1.1. Rulare pipeline din Github Actions.

CAPITOLUL 2. ARHITECTURA APLICAȚIEI

Aplicația urmează o arhitectură de tip microservicii containerizată, orchestrată cu ajutorul *MicroK8s*, fiind concepută pentru a asigura scalabilitate, securitate și observabilitate ridicată. Front-end-ul furnizează interfață utilizatorului și comunică cu back-end-ul prin intermediul unor gateway-uri, care gestionează traficul și accesul într-un mod controlat și securizat. În plus, codul client include automat un script de colectare a statisticilor anonime,

care transmite date către *Umami*, oferind o perspectivă generală asupra utilizării aplicației și comportamentului vizitatorilor.

Back-end-ul este alcătuit dintr-un set de microservicii autonome, fiecare ideal cu propria bază de date, și capabilități de comunicare inter-servicii, fie direct prin *HTTP*, fie prin intermediul unui sistem de mesagerie asincronă. Pentru componenta de procesare *AI*, sunt utilizate servicii dedicate cu suport *GPU*, gestionate prin *GPU Operator*, printre care se numără instanțe *Ollama* și microservicii dezvoltate în *Python*, bazate pe modele din ecosistemul *HuggingFace*.

Certificarea este automatizată, iar monitorizarea completă se realizează prin sisteme integrate de logare și metriki. Întregul sistem rulează containerizat, cu suport pentru scalare automată și întreținere facilă, adaptat cerințelor moderne de performanță și disponibilitate.

Structura aplicației la nivel de servere este ilustrată în [Figura 2.1.](#):

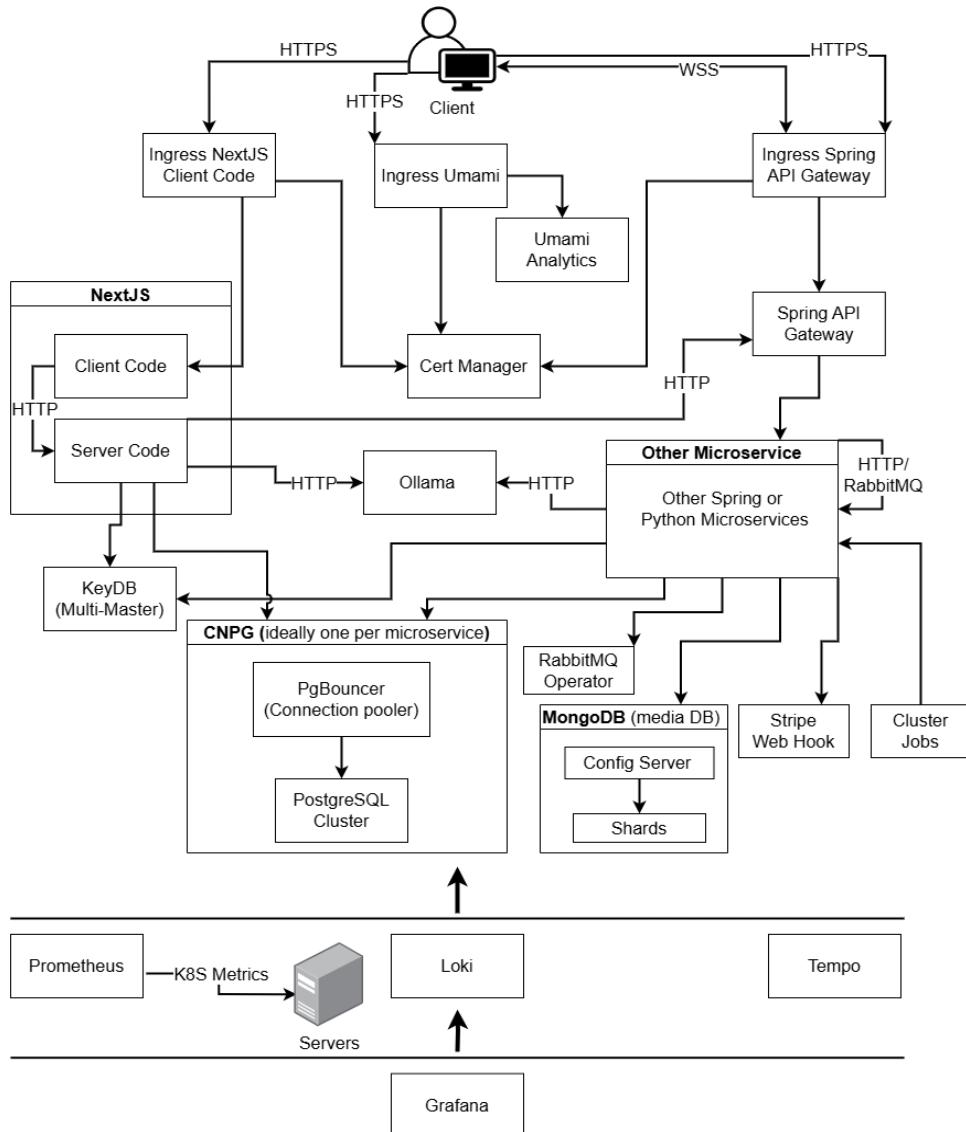


Figura 2.1. Diagrama deployment-ului în cloud-ul privat.

CAPITOLUL 3. IMPLEMENTAREA

3.1 Caching

Cum orice platformă actuală trebuie să răspundă rapid cerințelor utilizatorilor și mai ales cerințelor unui număr considerabil de utilizatori, caching-ul ([pentru detalii, Anexe, Definiții, Caching](#)) este o metodă consacrată de a mări viteza de răspuns a unei aplicații. În cadrul aplicației concrete, care respectă arhitectura propusă, am fost nevoit să implementez aproape de la zero mecanisme complexe de caching atât în back-end, cât și în front-end, pentru a satisface standardele actuale ale domeniului. Implementarea manuală a fost aleasă în detrimentul a ceva deja existent, întrucât nu am vrut să mă dezic de la paradigmile reactive și ale streaming-ului de date.

3.1.1. Caching Back-end

În back-end am ales o arhitectură de cache cu două niveluri, adică unul în memoria fiecărei instanțe ale unui microserviciu, iar cel de-al doilea distribuit între instanțe prin *KeyDB*. Am optat pentru această arhitectură întrucât, oricât de rapidă ar fi o bază de date in-memory, nu trebuie supraestimată viteza, deoarece, ca în orice bază de date, un număr mare de conexiuni și interogări poate duce la degradări masive ale performanței.

Conceptual, arhitectura caching-ului este după cum urmează:

- *Colecția principală*, cea care ține minte datele efective care se doresc a fi cache-uite, adică lista care survine în urma unui stream cu mai multe elemente sau elementul pentru un stream de lungime unu.
- *Colecția de chei de invalidare pentru fiecare ID / Reverse Cache Key*. Pentru a putea invalida la nevoie doar cât mai puțin din cache, pentru o cheie principală, la momentul când se dorește salvarea în colecția principală, am fost nevoit să definesc și modul de a obține ID-ul fiecărui element din stream-ul principal. Astfel, în această colecție de invalidare, pentru fiecare ID vom avea o mulțime de chei ale cache-ului principal, iar la invalidarea unui ID printr-un update/delete/altă operațiune, vom putea să exact ce chei ale cache-ului principal trebuie invalidate și, astfel, vom putea minimiza datele *stale* și eficientiza procesul prin invalidarea unei mulțimi minime, dar acoperitoare, de elemente din cache.

Spring Boot oferă deja o librărie de caching foarte potentă, însă aceasta este blocantă, iar singura opțiune pentru un stream de date este blocarea stream-ului până la ultimul element, după care salvarea în cache, iar apoi servirea tuturor datelor către client. Pentru a combatе

această limitare majoră și pentru a oferi o salvare transparentă față de client, în mecanismul de caching am optat pentru duplicarea printr-un „side effect” a stream-ului original fără a impune colectarea acestuia, iar doar după ce este terminat aplic logica de creare a elementelor necesare pentru cache.

Pentru a eficientiza salvarea în cache, am ales ca pentru fiecare stream să „trimit” acel proces secundar într-un virtual thread, întrucât acesta, preponderent, va aștepta ca stream-ul să fie terminat, deci putem spune că este, într-un fel, *I/O bound*, făcând folosirea acestor thread-uri virtuale optimă.

Inspirându-mă din librăria de caching pusă la dispoziție de *Spring*, am ales să creez propriile aspecte ([pentru detalii, Anexe, Definiții, Aspecte Si Spring AOP](#)) pentru salvarea și invalidarea cache-ului.

Deoarece sunt prezente două niveluri de caching, unul local per instanță și unul global per toate instanțele, complexitatea implementării este exponențial mai ridicată decât o simplă salvare într-un nivel global. Deci, pentru extragerea din cache se disting cazurile:

- *În niciun nivel nu este prezentă cheia cache-ului primar*: cache-ul primar este salvat atât local, cât și global, iar pentru ID-urile elementelor se creează/se adaugă la mulțimile de invalidare atât în cache-ul local, cât și global, cheia chache-ului primar.
- *Cheia cache-ului primar este găsită în local*: se servește cererea direct din cache-ul local, iar cel global nu este interogat.
- *Cache-ul local nu conține cheia primară, dar cel global o conține*: se adaugă doar la colecția primară locală datele din cache-ul global, după care datele sunt servite clientului.

Pentru o reprezentare vizuală, se poate consulta [Figura 1 din secțiunea Anexe, Figuri](#). De asemenea, pentru o secțiune de cod care gestionează salvarea unui publisher cu mai multe elemente, se poate consulta [Extras Cod-Sursă 1 din Anexe, Secvențe De Cod](#).

Invalidarea cache-ului pentru un ID este mai complexă, întrucât primul nivel este local și, astfel, se poate ajunge foarte ușor la inconsistențe. Așadar, pentru a minimiza datele *stale* fără a bloca execuția invalidării pentru instanța la care cererea care execută invalidarea a fost redirecționată, fiecare instanță va avea asociată o coadă anonimă care va fi conectată la un *Fanout Exchange* comun ([pentru detalii, Anexe, Definiții, Fanout Exchange](#)). Mesajul din coada anonimă va avea asociat și un ID unic al instanței care a propagat invalidarea, astfel minimizând invalidările inutile în cache-uri. Deci, pentru invalidare se disting următoarele:

- *Pipeline pentru invalidare în instanța în care cererea a ajuns*: pentru a realiza o invalidare cât se poate de exactă, se combină elementele din cache-ul local și global pentru

colecțiile de invalidare asociate ID-ului elementului invalidat, se invalidează cache-ul local și global și se vor trimite către celelalte instanțe două notificări prin coada de mesaje: prima cu cheile pentru cache-ul primar, iar a doua cu cheia (i.e. ID-ul) ce trebuie scoasă din cache-ul de invalidare. Ambele notificări conțin și identificatorul instanței care a trimis aceste notificări. Pentru o reprezentare vizuală, se poate consulta [Figura 2 din secțiunea Anexe, Figuri](#).

- *Primirea notificărilor de invalidare:* se verifică dacă identificatorul instanței care primește notificarea este cel al instanței care a trimis, dacă da, atunci notificarea este ignorată, dacă nu, conform tipului notificării, se invalidează cheile cache-ului corespunzător. Pentru o reprezentare vizuală, se poate consulta [Figura 3 din secțiunea Anexe, Figuri](#).

Această prezentare și logică de invalidare este doar implementarea de bază. În cadrul aplicației a mai fost extinsă prin multiple subimplementări care augmentează această bază pentru a răspunde necesităților specifice de invalidare și salvare: de exemplu, în aplicația demonstrativă, care include postări de tip blog, acestea pot fi aprobate sau neaprobată. Când o postare își schimbă statusul din aprobată în neaprobată, trebuie invalidate atât liste de aprobate pentru anumiți utilizatori, întrucât postarea nu mai face parte din ele, dar și cele de neaprobată, întrucât postarea acum face parte din acele liste.

3.1.2. Cache și Request-uri Front-end

Pentru front-end am fost nevoie, din nou, să implementez aproape de la zero logica de a transmite request-uri de tip *HTTP*, deoarece, folosind streaming și *Application/x-ndjson* ([pentru detalii, Anexe, Definiții, Application/x-ndjson](#)), nu există o librărie care să implementeze măcar o parte din cerințele de caching și deduplicare de request-uri pentru acest tip de date. Dar, pentru a avea un model conceptual matur, am decis să mă inspir din librăria *SWR* de la *Vercel*.

În teorie, programarea reactivă și paradigma streaming-ului sugerează să nu se blocheze primirea, ci acțiunile să se realizeze pentru fiecare element nou din stream, pe măsură ce acesta este consumat. Această soluție este destul de ineficientă într-un mod „pur” în front-end, unde, pentru o listă, am reactualizat *UI*-ul pentru fiecare element. Deci, am optat pentru o soluție de mijloc, cu un compromis rezonabil, adică batch-uirea elementelor, fără a bloca total stream-ul inițial, iar apoi elementele să fie transmise spre *UI* în batch-uri, diminuând numărul de rerandări.

Pentru caching am folosit un mecanism de tip *LRU* (Last Recently Used) cu un timeout pentru a păstra aceste batch-uri in memory. Partea interesantă la acest cache constă în faptul că este de tip optimist, distingându-se următoarele:

- Dacă cache-ul este gol pentru o cheie, atunci, în momentul în care se transmit batch-urile către *UI*, acestea vor fi salvate și în cache.
- Dacă cache-ul conține elemente pentru o cheie, atunci se transmite la client ce este în cache, request-ul este de asemenea transmis către server, iar pentru fiecare batch al noului request se verifică dacă este la fel cu cel din cache. Dacă este la fel, se ignoră, dacă este diferit, se schimbă în cache doar batch-ul respectiv și se transmite spre UI incremental fiecare batch schimbat. Această arhitectură este eficientă, încrucișat, în *React*, într-o listă, fiecare element trebuie să aibă o cheie, iar acea cheie este natural să fie ID-ul sau ceva derivat din element, astfel, vom reraunda doar părțile necesare, nu toate elementele asociate listei în *UI*. Pentru o reprezentare vizuală, se poate consulta [Figura 4 din secțiunea Anexe, Figuri](#).

Pentru a implementa deduplicare de request-uri ([pentru detalii, Anexe, Definiții, Deduplicare](#)), am folosit o arhitectură *TTL* (Time to Live Cache) cu un timeout de 1.050 ms. Neavând o simplă promisiune întoarsă ce conține o listă, la bază, funcția care face fetch adera la o arhitectură de callbacks pentru a putea gestiona batch-urile, însă pentru a salva în acest cache de deduplicare a trebuit să fac manual o conversie de la arhitectura de callbacks la cea de *AsyncIterator*. Această conversie a fost necesară încrucișat răspunsul unei cereri nu este o simplă promisiune cu un obiect *JSON*, ci este un stream de promisiuni de obiecte *JSON*, și astfel, în cache-ul de deduplicare salvez stream-ul de promisiuni în sine, adică un *AsyncGenerator*, producer-ul. Această arhitectură complexă este necesară, încrucișat am dorit ca pentru un request să fie transparentă deduplicarea, netrebuind modificări ale modului de cerere al datelor în funcție de prezența în cache a cheii. Pentru o secțiune de cod care gestionează deduplicarea se poate consulta [Extras Cod-Sursă 2 din Anexe, Secvențe De Cod](#).

3.1.3. Caching Imagini În Ingress

În cadrul aplicației demonstrative a fost implementat un microserviciu care are rolul de media service. *NextJS* are o componentă specială pentru imagini, fiind un *wrapper* peste cea nativă din *HTML*. Această componentă permite definirea unui *loader*, adică modelul procedurii de încarcare a imaginilor, și trimit automat către acesta lățimea și calitatea imaginii, în funcție de anumite proprietăți specificate în cod și *CSS*. Așadar, pentru a minimiza dimensiunea traficului către client, am ales să folosesc acest *loader*, astfel, în microserviciu se manipulează imaginea aşa încât să respecte request-ul și să se trimită către

client imaginea direct prelucrată, micșorându-se astfel cantitatea de date transferată. Această manipulare este destul de costisitoare și de durată, având în vedere specificațiile serverelor, aşadar am fost nevoie să recurg la anumite optimizări:

- Răspunsul care conține imaginea va avea asociate header-urile pentru cache-ul în browser ([pentru detalii, Anexe, Definiții, Browser Cache Headers](#)), întrucât imaginile sunt imutabile și statice prin natura lor.
- Pentru a minimiza resursele computaționale folosite, am recurs la a salva în cache-ul ingress-ului imaginile manipulate ([pentru detalii, Anexe, Definiții, NGINX Caching](#)) și astfel o cerere poate fi interceptată de către reverse-proxy, neajungând deloc la microserviciu.

Aceste mecanisme de caching nu sunt aplicate în cazul videoclipurilor, întrucât am putea foarte repede suprasatura atât stocarea clientului, cât și cea din *NGINX*. De asemenea, pentru videoclipuri nu se face o manipulare specifică și astfel, având în vedere că implementarea proprie suportă request-uri Range ([pentru detalii, Anexe, Definiții, Range Requests](#)), acestea sunt deja gestionate eficient de către browser, iar microserviciul doar citește din baza de date chunk-urile cerute, nu tot fișierul, datorită caracteristicilor GridFS ([pentru detalii, Anexe, Definiții, GridFS](#)). Pentru o secțiune de cod care gestionează cererile de tip Range pentru videoclipuri se poate consulta [Extras Cod-Sursă 6 din Anexe, Secvențe De Cod](#). Totodată, acest mecanism de cache este utilizat și pentru alte resurse statice din aplicație, cum ar fi anumite fragmente din codul generat de *NextJS*.

3.2. Căutare Semantică În Baza De Date

Pentru a crea o experiență de utilizator la standarde enterprise, orice aplicație web actuală ar trebui să aibă o modalitate de căutare de similaritate semantică, fie ea folosind strategii *fuzzy*, indecsi semantici precum cei din *ElasticSearch* sau chiar din *Postgre*. În implementarea de față, am optat pentru o abordare unică, sau cel puțin avangardistă. Unul dintre motivele pentru care am ales *Postgre* ca bază de date principală este extensia *PgVector*, aceasta oferă capabilități de căutări de similaritate pentru vectori de dimensiuni mari și acceptă și indecsi specifici, precum cel *HNSW* (Hierarchical Navigable Small World) folosit în implementarea de față ([pentru detalii, Anexe, Definiții, HNSW](#)). Pentru a crea acești vectori din siruri de caractere, am ales un model de creare de embedding-uri ([pentru detalii, Anexe, Definiții, Embedding](#)), anume *BGE-M3*, apoi salvez aceste embedding-uri în baza de date, indexându-le folosind *HNSW* pentru eficiență de căutare. Pentru a duce optimizarea căutărilor la nivelul următor, având în vedere că modelul de embedding întoarce vectori

aproape normalizați, am optat pentru similaritate folosind produs scalar ([pentru detalii, Anexe, Completări, Eficiență Produs Scalar](#)). De asemenea, având în vedere că serverele folosesc plăci video relativ învechite, optimizarea performanței devine esențială. Prin urmare, și pentru embedding-uri există un mecanism de cache, bazat pe *KeyDB*, întrucât generarea unui vector de embedding pentru un sir de caractere nu este un proces instantaneu.

De exemplu, în cazul postărilor din aplicație, la salvarea sau actualizarea titlului, acesta este procesat prin modelul de embedding, rezultând un vector care este apoi stocat și în baza de date. Pipeline-ul de filtrare a postărilor după titlu funcționează astfel:

- Interogarea este curățată și apoi transformată într-un vector de căutare. Acest vector este obținut fie din cache, fie generat de modelul de embedding, în cazul în care nu există deja în cache. După generare, vectorul este salvat în cache pentru utilizări viitoare, cu un timp de expirare.
- Se caută titlurile postărilor care au cea mai mare similaritate semantică cu vectorul de căutare. De asemenea, se definește un prag minim al similarității, pentru a filtra rezultatele care, deși au o anumită similaritate (specifică oricărei perechi de vectori într-un spațiu vectorial), nu ating un nivel considerat semnificativ pentru contextul semantic al căutării.

Pentru o reprezentare vizuală a procesului, se poate consulta [Figura 5. din secțiunea Anexe, Figuri](#). De asemenea, un exemplu concret din implementare este ilustrat în [Figura 3.1.:](#)

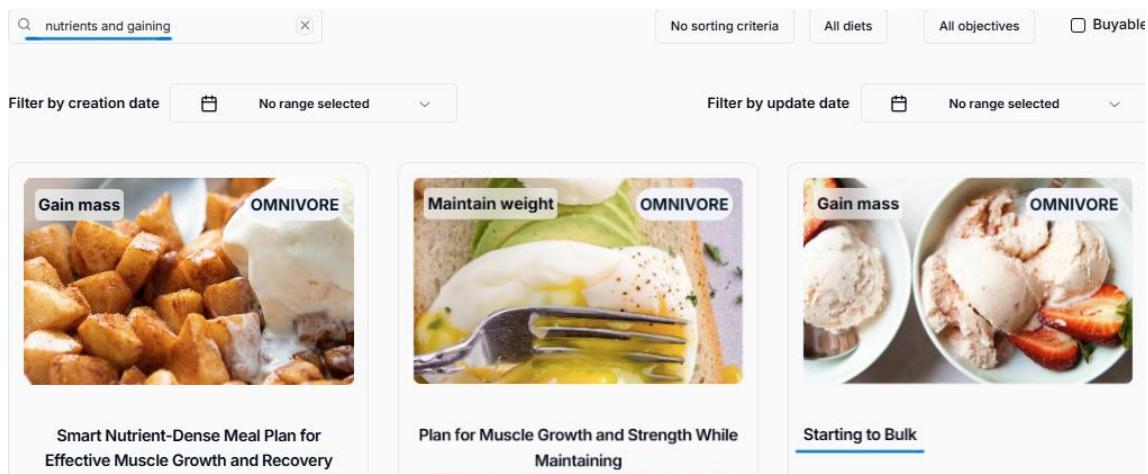


Figura 3.1. Căutare semantică în lista de planuri.

3.3. Microserviciile Python

Pentru a gestiona interogarea unor pipeline-uri de *AI* folosind *HuggingFace*, am optat pentru crearea de servicii adiacente în *Python*, folosind microframework-ul *Flask*. Ca orice alte servicii din cadrul aplicației, acestea sunt conectate la *Prometheus* pentru colectarea de metrii și la serviciile *Grafana*: *Loki* și *Tempo*.

Carcateristicile serviciilor sunt după cum urmează:

- Serviciul de generare de imagini: în cadrul acestuia este folosit un pipeline de „stable diffusion” pentru a permite generarea de imagini din aplicație după cerințele utilizatorilor, la crearea unor anumite elemente precum postări, rețete etc.
- Serviciul de predicție temporală: în cadrul acestuia este folosit un pipeline de „time series prediction” pentru a prezice vânzările viitoare în cadrul platformei.
- Serviciul de moderare lingvistică: asigură, pe de o parte, că anumite elemente precum unele prompturi la alte facilități *AI* sau comentariile la postările din cadrul aplicației sunt în limba engleză, iar pe de altă parte, că limbajul este decent și adekvat, nefiind vulgar sau toxic. Un exemplu concret pentru această moderare se poate observa în [Figura 3.2.](#):

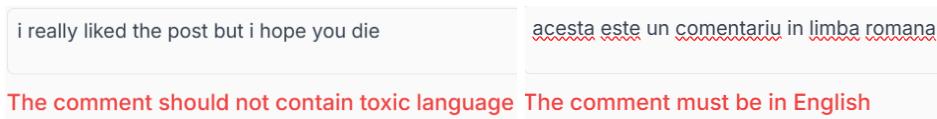


Figura 3.2. Moderare text în aplicație.

Pentru o secțiune de cod care creează pipeline-ul de difuzie, se poate consulta [Extras Cod-Sursă 7 din Anexe, Secvențe De Cod](#).

3.4. AI În NextJS

În cadrul aplicației *NextJS*, am integrat biblioteca *LangChainJS* pentru a crea mai multe funcționalități care valorifică modelele de inteligență artificială rulate local prin *Ollama*. Astfel, aplicația beneficiază de un cadru modular și extensibil pentru interacțiunea cu *LLM*-uri și modele de embedding, permitând orchestrarea ușoară a prompturilor, gestionarea contextului și integrarea surselor externe de date. [\[23\]](#)

3.4.1. Chat Bot

Pentru a oferi context modelului *LLM*, am făcut *scraping* la propriul site, iar apoi am salvat aceste informații în mod vectorial în *Postgre*. Astfel, modelul este capabil să răspundă întrebărilor care au legătură cu site-ul. De asemenea, în cadrul chat-ului se pot realiza și căutări pentru postări și planuri nutriționale.

Pe scurt, o interacțiune cu chat-ul respectă următorii pași:

- Chat-ul are la dispoziție istoricul recent al conversației și baza de date cu informațiile despre site.
- Întrebarea utilizatorului este trecută printr-un *multi-query retriever* care are ca scop de a crea mai multe query-uri asemănătoare cu cel al utilizatorului pentru a extrage informațiile

necesare din context într-un mod exhaustiv. Acest retriever folosește tot modelul *LLM* pentru a realiza generările de noi interogări.

- Contextul este comprimat folosind modelul de embedding pentru a elimina redundanțele și pentru a fi minimizat, încărcând pe plăci video învechite, sunt anumite limitări atât la memoria *VRAM*, cât și la capacitatea de procesare. (pentru detalii, [Anexe, Definiții, Contextul Unui LLM și Anexe, Completări, Limitări Plăci Video Vechi](#))

- Modelul decide dacă trebuie să facă căutări în back-end pentru postări sau planuri nutriționale.
- Modelul întoarce răspunsul sub formă de stream pentru o experiență de utilizare îmbunătățită.

Pentru o reprezentare vizuală, se poate consulta [Figura 6 din secțiunea Anexe, Figuri](#).

Un exemplu de interacțiune este ilustrat în [Figura 3.3.:](#)



Figura 3.3. Interacțiune cu chatbot-ul cu extragere de informații din baza de date.

3.4.2. Alte Utilizări AI

- *Căutări în corpul elementelor:* aplicația demonstrativă oferă posibilitatea de a efectua căutări în limbaj natural direct în conținutul elementelor. Această funcționalitate permite utilizatorilor să formuleze interogări expresive, apropiate de limbajul uman, pentru a regăsi rapid informații relevante, fără a fi necesare filtre rigide sau termeni exacți. Un exemplu este ilustrat în [Figura 3.4.:](#)

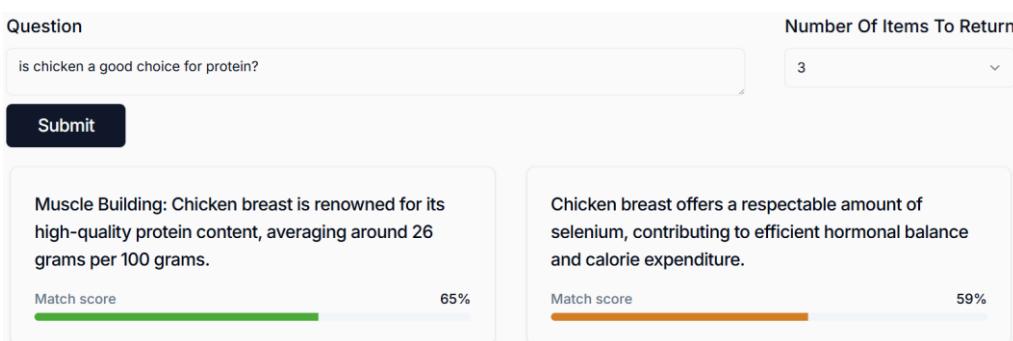


Figura 3.4. Căutare cu AI în corpul elementelor.

- Generarea de titluri și descrieri pentru elemente: Aplicația permite utilizarea unui model *LLM* pentru a asista utilizatorii în redactarea titlurilor sau descrierilor aferente elementelor, facilitând astfel procesul de creare a conținutului. Totuși, utilizatorul rămâne în

întregime responsabil pentru validarea și acceptarea conținutului generat, înainte ca acesta să fie salvat și utilizat în platformă. Un exemplu se poate observa în [Figura 3.5.](#):



Figura 3.5. Generare titlu cu AI.

3.5. Internaționalizare

Pentru a oferi utilizatorilor o experiență cât mai accesibilă și plăcută, front-end-ul aplicației a fost conceput cu suport pentru internaționalizare (i18n). În prezent, interfața este disponibilă în două limbi, română și engleză, însă arhitectura aleasă permite extinderea rapidă și facilă către orice altă limbă. Acest lucru a fost posibil prin integrarea unei biblioteci dedicate internaționalizării, care gestionează textele aplicației pe baza unor fișiere *JSON* separate pentru fiecare limbă. Astfel, adăugarea de noi traduceri presupune doar crearea unui nou fișier *JSON* corespunzător și completarea acestuia cu valorile necesare, fără a fi nevoie de modificări în logica aplicației. Această flexibilitate contribuie la scalabilitatea proiectului și la adaptarea sa pentru un public internațional. Un exemplu este ilustrat în cadrul figurilor [28](#) și [29](#) din secțiunea Anexe, Figuri.

3.6. Autentificarea Utilizatorilor

Pentru autentificarea utilizatorilor există două opțiuni:

- O primă opțiune este autentificarea cu e-mail și parolă locală. În cadrul acesteia, utilizatorii își pot crea un cont specific aplicației, folosind un e-mail unic și o parolă, care este criptată și stocată în siguranță în baza de date. Sistemul include, de asemenea, o funcționalitate completă de resetare a parolei, oferind utilizatorilor posibilitatea de a-și recupera accesul în cazul în care au uitat parola sau doresc să o actualizeze din motive de securitate.
- Aplicația oferă utilizatorilor posibilitatea de a se autentifica utilizând standardul OAuth2 ([pentru detalii, Anexe, Definiții, Oauth2](#)), integrând în acest scop provideri cunoscuți precum *Google* și *GitHub*. Astfel, utilizatorii pot opta pentru o autentificare rapidă și securizată, fără a fi necesar să creeze un cont nou, beneficiind de acces facil prin intermediul conturilor existente pe aceste platforme.

Pentru gestionarea acestor procese de autentificare și autorizare în back-end, aplicația utilizează un *API Gateway* central, care intermediază toate cererile venite de la client. Gateway-ul realizează autentificarea și autorizarea cererilor prin apeuri *HTTP* către microserviciul de utilizatori, iar apoi propagă cererile doar dacă acestea sunt valide, în funcție de ruta accesată și de politicile de acces configurate. Pentru transmiterea identității utilizatorului de către client la server, sistemul suportă atât token-uri *JWT* ([pentru detalii, Anexe, Definiții, JWT](#)), cât și cookie-uri securizate, alegerea fiind dinamică în funcție de contextul cererii. De exemplu, în cazul stabilirii unei conexiuni *WebSocket*, unde transmiterea de header-e personalizate este limitată, autentificarea se face eficient prin intermediul cookie-urilor, în timp ce pentru cererile *HTTP* clasice se utilizează preponderent *JWT* în header-ul *Authorization*. În plus, gateway-ul joacă un rol esențial în protejarea aplicației împotriva atacurilor de tip *CSRF* (Cross-Site Request Forgery), prin verificarea token-ului transmis de client. Astfel, acesta acționează ca un punct de control centralizat, contribuind semnificativ la securitatea, modularitatea și scalabilitatea arhitecturii generale.

3.7. Arhivare

În cadrul aplicației, pentru a păstra istoricul modificărilor asupra elementelor, fie că sunt actualizate sau șterse, a fost implementat un microserviciu dedicat gestionării acestor evenimente. La fiecare acțiune de tip update sau delete, aplicația publică un mesaj într-o coadă durabilă în *RabbitMQ*, asigurând astfel persistența și trasabilitatea evenimentelor.

Având în vedere că aceste cozi pot acumula un număr mare de mesaje într-un timp scurt, a fost necesară optimizarea modului de procesare. Astfel, în locul modelului tradițional *push-based*, adică modul normal de operare în *RabbitMQ*, am optat pentru o abordare *pull-based*, permitând un control mai fin asupra momentului și volumului de mesaje prelucrate în unitatea de timp. Pentru a susține acest comportament, am exploatat posibilitatea oferită de *Spring AMQP* de a configura și porni listeneri în mod programatic, permitând declanșarea manuală a prelucrării de mesaje, în funcție de nevoile aplicației și de capacitatea de procesare disponibilă la un anumit moment. Această abordare oferă atât flexibilitate, cât și scalabilitate în gestionarea fluxurilor de date.

Pentru a controla execuția acestui listener am recurs la următoarele:

- *Job recurrent de tip Cron temporizat*: un task programat care, pe baza unei expresii *Cron* predefinite și a unei durate de viață configurabile, pornește și oprește automat listener-ul asociat unei cozi, permitând procesarea periodică și controlată a mesajelor.

- *Pornire manuală temporizată*: listener-ul este activat manual pentru o perioadă de timp prestabilită, după care este dezactivat automat, oferind un control precis asupra ferestrei de procesare în funcție de nevoile operaționale.

Pentru o secțiune de cod care gestionează acest ciclu de viață al listener-ilor RabbitMQ se poate consulta [Extras Cod-Sursă 3 din Anexe, Secvențe De Cod](#).

Având în vedere că arhitectura actuală presupune acumularea mesajelor într-o coadă, urmată de procesarea lor într-un singur flux controlat, am optat pentru utilizarea unor listeneri de tip batch ([pentru detalii, Anexe, Definiții, Spring AMQP Batch RabbitMQ Listener](#)). Această alegere permite procesarea simultană a unui grup de mesaje, reducând astfel costurile de comunicare și îmbunătățind semnificativ eficiența generală a sistemului.

În cadrul aplicației de față, mesajele rezultate în urma procesării cozii sunt salvate pe disc sub forma de fișiere *JSON*, fiecare fișier fiind asociat momentului exact al procesării. Această abordare asigură o persistență clară și organizată a datelor procesate, facilitând atât auditarea, cât și eventuale analize ulterioare. Arhitectura este însă flexibilă și permite cu ușurință salvarea în alte formate (*CSV, Avro, Parquet* etc.) sau chiar direct într-o bază de date, fiind astfel potrivită pentru o viitoare integrare cu o bază de date de tip data warehouse, precum *Oracle* sau *HBase*, în scopuri de raportare sau analiză avansată.

Pentru o reprezentare vizuală, se poate consulta [Figura 7 din secțiunea Anexe, Figuri](#).

3.8. Comunicarea Între Microserviciile Din Back-end

În cadrul unei arhitecturi pe microservicii, fiecare componentă a aplicației este izolată și rulează independent, ceea ce impune o necesitate constantă de comunicare între servicii. Pentru a asigura o comunicare eficientă și scalabilă, microserviciile dezvoltate cu *Spring* utilizează frecvent *WebClient*, componentă a *Spring WebFlux*, care permite efectuarea de cereri *HTTP* într-un mod reactiv, adică asincron și non-blocant, optim pentru aplicații cu trafic intens și cerințe ridicate de performanță.

Adiacent apelurilor *HTTP*, serviciile colaborează și prin intermediul unui sistem de mesagerie asincronă, utilizând *RabbitMQ* atât pentru schimbul de mesaje prin cozi durabile, cât și pentru implementarea de mecanisme de tip Remote Procedure Call (RPC), atunci când este necesară o interacțiune sincronă sau asincronă, dar decuplată de rețeaua clasiceă *HTTP*. Această combinație de protocoale asigură flexibilitate, reziliență și adaptabilitate în funcție de nevoile concrete ale fiecărui flux de date și fiecărei situații concrete.

Pentru o reprezentare vizuală, se poate consulta [Figura 8 din secțiunea Anexe, Figuri](#).

3.8.1. Comunicarea HTTP

Spre deosebire de alte implementări blocante, *WebClient* permite un model de programare bazat pe fluxuri de date (*Mono*, *Flux* din *Project Reactor* în cazul de față), în care cererile către alte servicii pot fi lansate fără a bloca firele de execuție. Acest lucru este extrem de util în aplicații cu cerințe ridicate de concurență și scalabilitate, în care gestionarea eficientă a resurselor este critică.

Totuși, comunicarea într-o arhitectură distribuită este inevitabil expusă unor riscuri: servicii indisponibile, rețea instabilă, răspunsuri lente sau eronate. Pentru a face față acestor situații și a evita degradarea în lanț a întregului sistem, am integrat biblioteca *Resilience4j*, un set de instrumente destinat creșterii toleranței la erori. Această librărie oferă un set de mecanisme pentru creșterea stabilității aplicațiilor, inclusiv *Circuit Breaker*, pentru oprirea temporară a apelurilor către servicii instabile, *Retry*, pentru reîncercarea automată a cererilor eşuate și *Rate Limiter*, pentru limitarea numărului de cereri. [16]

Combinatia dintre *WebClient* și *Resilience4j* oferă un echilibru solid între performanță și reziliență, permitând realizarea de cereri rapide și non-blocante între microservicii, în timp ce adaugă un strat esențial de control și protecție împotriva erorilor și instabilității din ecosistem. Această abordare este esențială în arhitecturile distribuite moderne, unde fiabilitatea aplicației depinde nu doar de propriul cod, ci și de comportamentul și disponibilitatea serviciilor externe cu care interacționează. Așadar, această arhitectură, deși nu implementează un service *mesh* în sensul clasic, reproduce multe dintre caracteristicile acestuia, creând o rețea logică de microservicii rezilientă, asemănătoare unui *mesh* network gestionat la nivel de aplicație. De exemplu, o configurare pentru comunicare cu serviciul de media este:

```
resilience4j.circuitbreaker.instances.fileService.automaticTransitionFromOpenToHalfOpenEnabled=true
resilience4j.circuitbreaker.instances.fileService.failureRateThreshold=30
resilience4j.circuitbreaker.instances.fileService.minimumNumberOfCalls=40
resilience4j.circuitbreaker.instances.fileService.permittedNumberOfCallsInHalfOpenState=5
resilience4j.circuitbreaker.instances.fileService.registerHealthIndicator=true
resilience4j.circuitbreaker.instances.fileService.slidingWindowSize=100
resilience4j.circuitbreaker.instances.fileService.slidingWindowType=TIME_BASED
resilience4j.circuitbreaker.instances.fileService.slowCallDurationThreshold=15s
resilience4j.circuitbreaker.instances.fileService.slowCallRateThreshold=50
resilience4j.circuitbreaker.instances.fileService.waitDurationInOpenState=5s
resilience4j.ratelimiter.instances.fileService.limitForPeriod=100
resilience4j.ratelimiter.instances.fileService.limitRefreshPeriod=10s
resilience4j.ratelimiter.instances.fileService.timeoutDuration=500ms
resilience4j.retry.instances.fileService.enableExponentialBackoff=true
resilience4j.retry.instances.fileService.enableRandomizedWait=true
resilience4j.retry.instances.fileService.exponentialBackoffMultiplier=2
resilience4j.retry.instances.fileService.maxAttempts=5
resilience4j.retry.instances.fileService.waitDuration=100ms
```

3.8.2. Comunicarea Prin Intermediul RabbitMQ

În ceea ce privește comunicarea asincronă între microservicii, utilizarea *RabbitMQ* oferă un mecanism eficient și flexibil, atât prin cozi de mesaje, cât și prin apeluri de tip *RPC*. Transmiterea prin cozi permite decuplarea completă a serviciilor, oferind toleranță la erori și o procesare scalabilă a mesajelor. Modelul asincron este ideal pentru operațiuni de tip *fire-and-forget*, precum arhivarea elementelor, curățarea cache-ului sau trimiterea de notificări între servicii, unde nu este necesar un răspuns imediat. În schimb, modelul *RPC* este util atunci când este necesară o reacție rapidă, păstrând totodată avantajele infrastructurii de mesagerie. Această combinație susține un ecosistem robust, capabil să gestioneze atât astfel de fluxuri asincrone, cât și cereri punctuale cu răspuns sincron.

Pentru microserviciul dedicat majorității comunicării *WebSocket* dintre front-end și back-end, care este implementat în mod non-reactiv folosind modulul *WebSocket* din *Spring* cu suport pentru protocolul *STOMP*, am optat pentru utilizarea unui model de comunicare *RPC* prin *RabbitMQ* în interacțiunea cu microserviciul de utilizatori, care este construit în mod reactiv. Această alegere permite menținerea unei comunicări de bază asincrone între cele două servicii, asigurând integrarea fluentă a unui flux reactiv cu unul tradițional, fără a bloca resurse sau a compromite caracterul distribuit al arhitecturii. Astfel, utilizarea acestui model creează o punte eficientă de legătură între fluxul reactiv și cel tradițional.

De asemenea, aplicația demonstrativă include un chat live între utilizatori, implementat prin intermediul protocolului *STOMP*, care facilitează comunicarea în timp real între client și server. Același protocol este utilizat și pentru transmiterea unor multiple notificări în timp real către client, asigurând o experiență interactivă și dinamică. Pentru a spori securitatea, mesajele transmise prin chat sunt criptate la salvarea în baza de date folosind algoritmul *AES* (Advanced Encryption Standard), astfel încât, în eventualitatea unei breșe de securitate, conținutul acestora să nu poată fi accesat în mod direct. Un exemplu de interacțiune cu chat-ul este ilustrat în [Figura 3.6.](#):

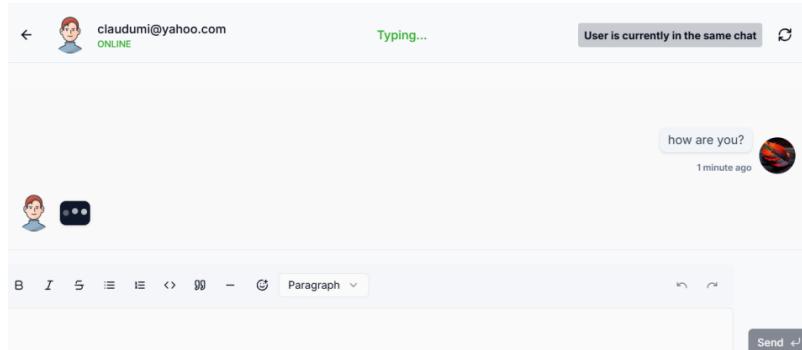


Figura 3.6. Live chat cu atenționare când partenerul scrie.

3.9. Blocări Distribuite În Redis

Aplicația demonstrativă reține numărul de vizualizări al fiecărei postări, însă, pentru a evita generarea unor operații de actualizare masivă și frecventă asupra bazei de date relationale, a fost implementat un *buffer* intermediar în *KeyDB*. Astfel, incrementările de vizualizări sunt înregistrate temporar în memoria *RAM* a cluster-ului, iar actualizarea efectivă în baza de date *SQL* se face periodic, reducând semnificativ stresul asupra sistemului principal de stocare.

Având în vedere natura distribuită a arhitecturii și posibilitatea existenței mai multor instanțe concurente ale aceluiași microserviciu, s-a impus necesitatea unui mecanism de *distributed locking*, pentru a asigura consistența operației de scriere. În acest scop, fiecare instanță încearcă să obțină un *lock* temporar în *KeyDB*, însă doar prima care setează cheia specifică are dreptul de a efectua actualizarea. Pentru siguranță, fiecare instanță atașează *lock*-ului o valoare unică (un *UUID* din *Java*), care este verificată înainte de eliberarea *lock*-ului distribuit, astfel încât doar instanța care l-a obținut inițial să îl poată revoca. *Lock*-ul este, de asemenea, însotit de un termen de expirare controlat, pentru a preveni blocajele în caz de eroare și a permite o revenire rapidă, cu minimă pierdere de informație. Acest mecanism asigură o actualizare sigură și coordonată într-un mediu concurrent, fără a compromite semnificativ integritatea datelor. Pentru o secțiune de cod care gestionează blocările distribuite se poate consulta [Extras Cod-Sursă 4 din Anexe, Secvențe De Cod](#).

3.10. Testare

Deoarece în cadrul back-end-urilor dezvoltate cu *Spring* logica aplicației este adesea complexă și bogată în fluxuri de decizie, se justifică pe deplin realizarea unor teste de calitate, menite să asigure funcționarea corectă și fiabilă a întregului sistem. În acest sens, am implementat un număr semnificativ de teste unitare și de integrare, utilizând instrumente specializate precum *JUnit5*, *Mockito*, *MockWebServer*, *Awaitility*, *TestContainers* și altele.

Dificultatea principală în scrierea acestor teste a fost dată de natura asincronă a aplicației, specifică arhitecturii reactive adoptate, care a impus o atenție sporită în gestionarea fluxurilor de date, sincronizarea etapelor de execuție și verificarea rezultatelor în contexte concurente și paralele. Acest aspect a necesitat adoptarea unor strategii și unele dedicate testării reactivității, astfel încât testele să rămână deterministe, relevante și robuste în fața comportamentului asincron al aplicației.

De asemenea, în cadrul aplicației, au fost implementate și teste comportamentale, în special pentru componenta de cache, unde nu este suficientă doar validarea valorilor stocate, ci este esențială și verificarea modului în care cache-ul este utilizat în contextul execuției. Astfel, testele nu urmăresc doar conținutul cache-ului, ci și dacă și când sunt apelate anumite metode, asigurându-se că logica de cache funcționează conform așteptărilor și contribuie eficient la optimizarea performanței aplicației. Această abordare ajută la validarea comportamentului general al sistemului, nu doar a rezultatelor punctuale, oferind o acoperire de testare mai profundă și mai relevantă.

Pentru o vizualizare a rezultatelor de acoperire pentru modulele de bază, se pot consulta figurile [9](#), [10](#) și [11](#) din secțiunea Anexe, Figuri. De asemenea, pentru un test care evidențiază comportamentul în cazul unui „cache miss” atât în cache-ul local, cât și în *KeyDB*, se poate consulta [Extras Cod-Sursă 5 din Anexe, Secvențe De Cod](#).

CAPITOLUL 4. SCENARII DE UTILIZARE

Aplicația demonstrativă are ca scop principal comercializarea planurilor nutriționale și publicarea de conținut relevant, precum postări și rețete, toate acestea fiind create exclusiv de către persoane autorizate, care au fost validate în prealabil de un administrator al platformei, pentru a garanta calitatea și credibilitatea informațiilor. În sprijinul acestui proces, atât utilizatorii cu drepturi de postare, cât și administratorii beneficiază de dashboard-uri specializate, adaptate rolului fiecărui, care le oferă instrumentele necesare pentru a gestiona eficient conținutul și a urmări performanțele în cadrul platformei.

4.1. Vizitator

Un vizitator este un utilizator care nu este autentificat în momentul accesării platformei și, prin urmare, beneficiază de acces restricționat. Acesta nu poate utiliza funcționalitățile principale ale aplicației, având acces doar la pagina principală, care conține numeroase elemente de tip *Call to Action*, menite să îl încurajeze să își creeze un cont. În plus, vizitatorii pot accesa separat un calculator de calorii și au posibilitatea de a interacționa cu un chatbot *AI* integrat. Această strategie urmărește în principal stimularea înregistrării, pentru a facilita implicarea activă și fidelizarea utilizatorilor. [Figura 4.1.](#) conține diagrama care evidențiază principalele scenarii pentru acest tip de utilizator:

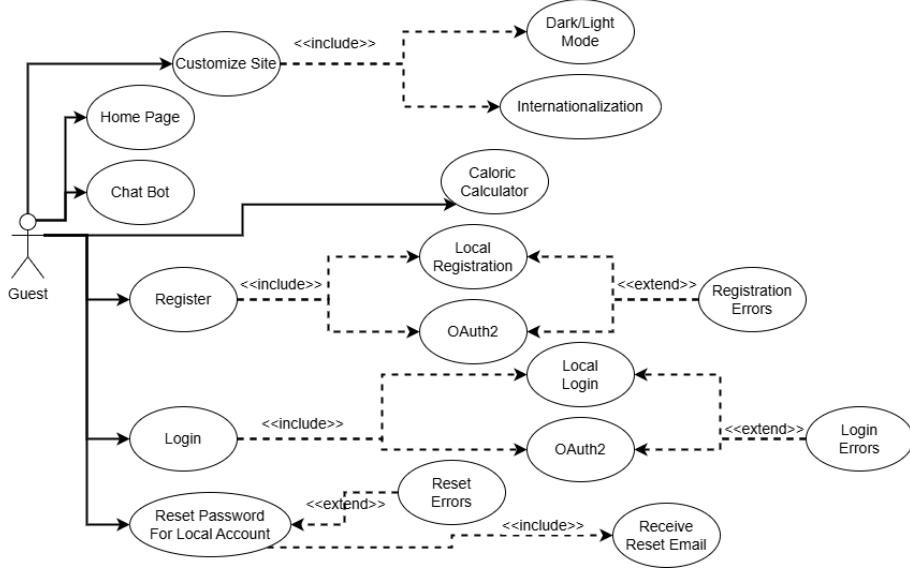


Figura 4.1. Diagrama scenarii de utilizare pentru vizitator.

4.2. Utilizator Autentificat

Un utilizator autentificat beneficiază de toate facilitățile oferite unui vizitator, având în plus acces complet la funcționalitățile principale ale platformei. Printre acestea se numără: achiziționarea de planuri alimentare, citirea și comentarea postărilor, exprimarea reacțiilor prin like sau dislike, utilizarea unui kanban personal pentru organizarea propriilor sarcini, un modul dedicat pentru monitorizarea zilelor în care a urmat un plan nutrițional, precum și pagini专特化 pentru gestionarea comenzi și vizualizarea planurilor achiziționate. De asemenea, platforma include un chat integrat cu alți utilizatori, care oferă notificări live, facilitând astfel comunicarea directă și în timp real între membri. Câteva exemple pentru interfață se pot observa în figurile [4.2.](#) și [4.3.](#):

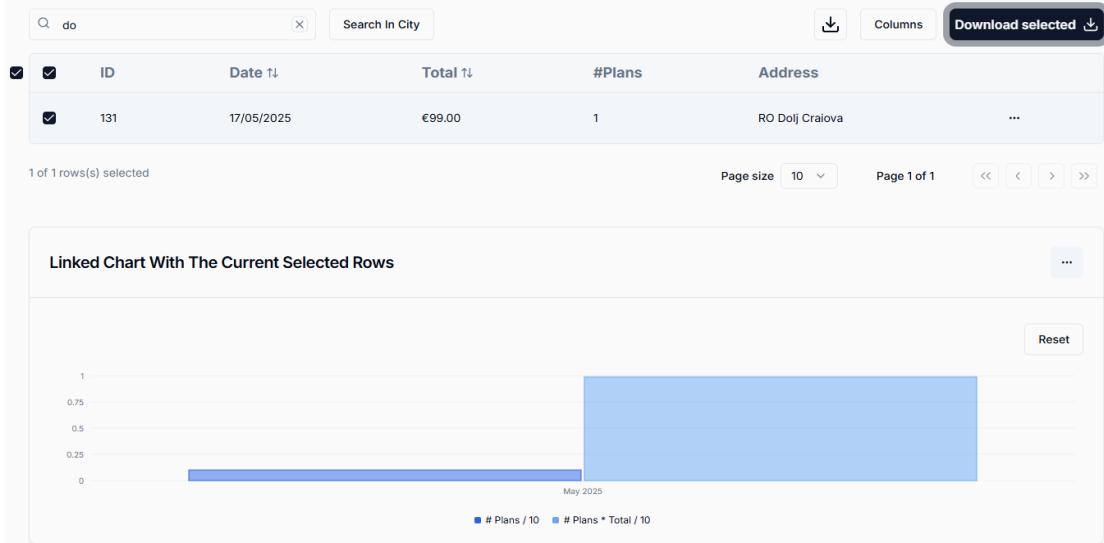


Figura 4.2. Pagina de comenzi pentru utilizator.

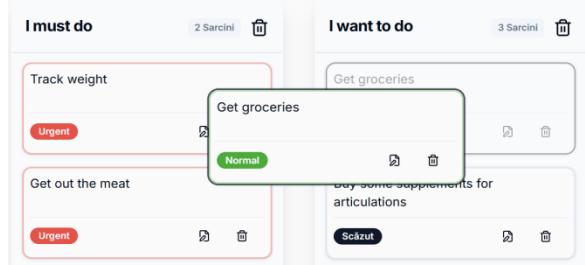


Figura 4.3. Kanban utilizator.

Pentru mai multe exemple se pot consulta figurile [25](#), [26](#) și [27](#) din secțiunea Anexe, Figuri. De asemenea, [Figura 4.4.](#) conține diagrama care evidențiază câteva dintre principalele scenarii pentru acest tip de utilizator:

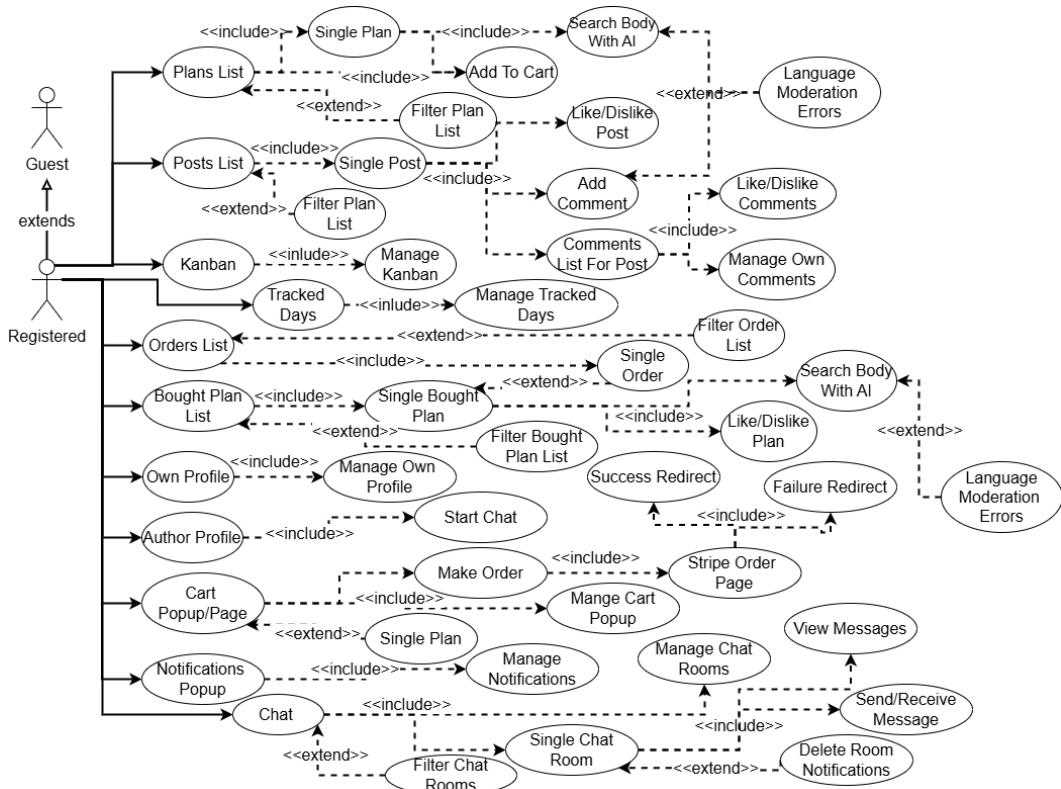


Figura 4.4. Diagrama scenarii de utilizare pentru utilizatorul autentificat.

4.3. Utilizator Cu Drepturi De Postare

Un utilizator cu drepturi de postare este un tip special de utilizator căruia administratorul i-a acordat permisiuni suplimentare pentru a contribui activ la conținutul platformei. Acesta beneficiază de toate funcționalitățile disponibile unui utilizator obișnuit, având în plus posibilitatea de a publica postări sau planuri alimentare. Totodată, are acces la un dashboard dedicat, unde poate consulta statistici detaliate privind performanța conținutului creat, precum și informații avansate despre vânzările realizate prin intermediul platformei. De asemenea, în procesul de creare a conținutului, acest utilizator poate apela la funcționalități

AI, atât pentru generarea de text, cât și pentru crearea de imagini, facilitând astfel un proces creativ mai rapid și eficient.

Câteva exemple pentru interfață se pot observa în figurile [4.5.](#) și [4.6.](#):

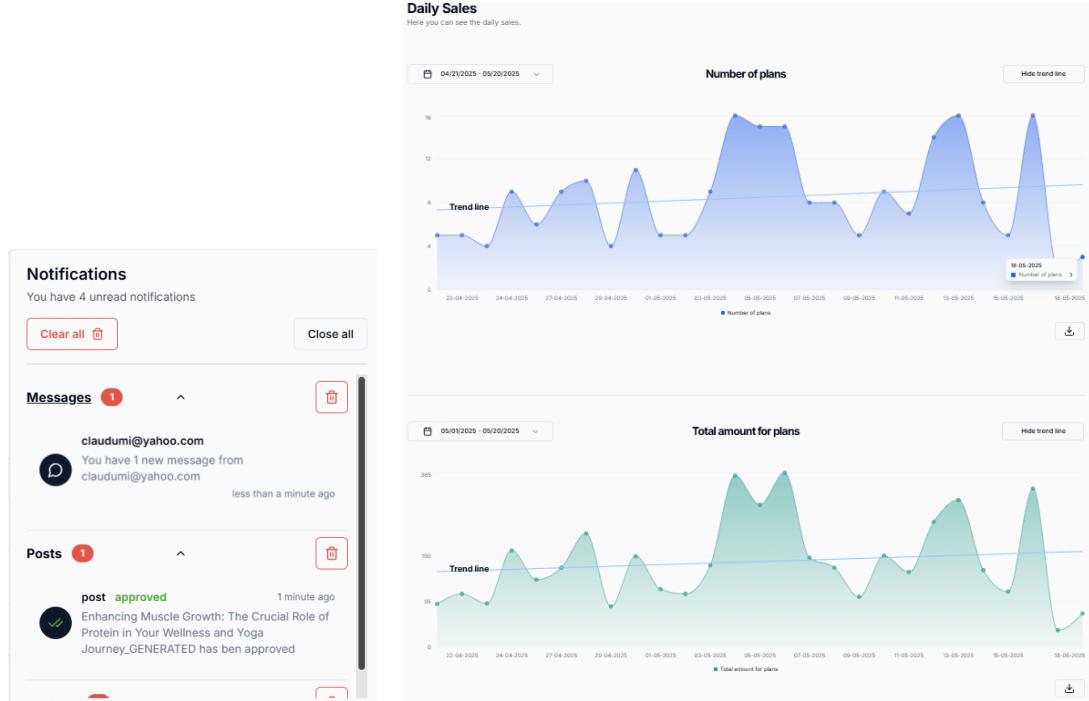


Figura 4.5. Pop-up notificări.

Figura 4.6. Pagina vânzărilor zilnice utilizator.

Pentru mai multe exemple se pot consulta figurile [20](#), [21](#), [22](#), [23](#) și [24](#) din secțiunea Anexe, Figuri. De asemenea, [Figura 4.7.](#) conține diagrama care evidențiază câteva dintre principalele scenarii pentru acest tip de utilizator:

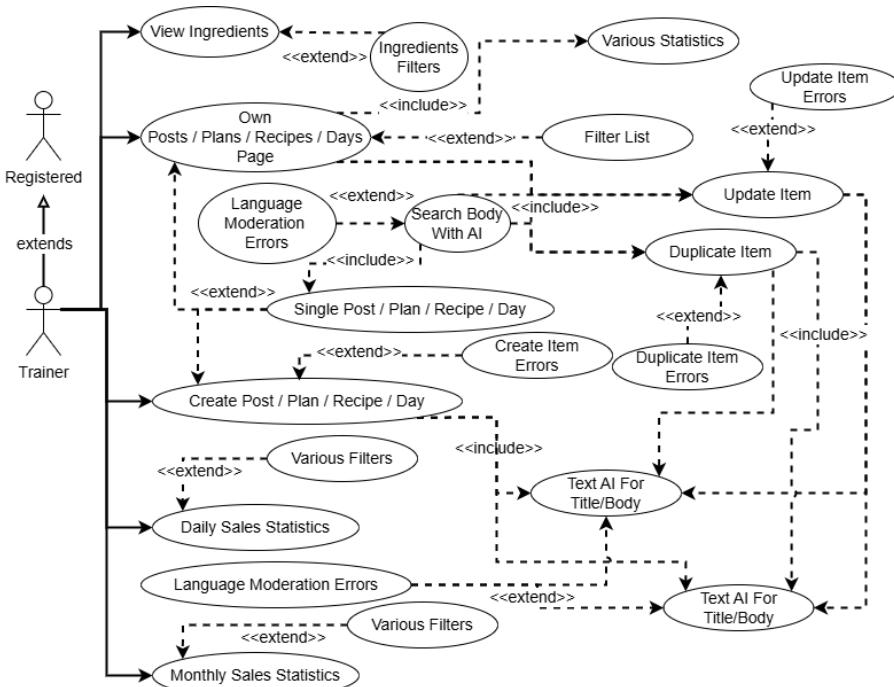


Figura 4.7. Diagrama scenarii de utilizare pentru utilizatorul cu drepturi de postare.

4.4. Utilizator De Tip Admin

Acest tip de utilizator deține drepturi depline în cadrul aplicației și are rolul de a gestiona elementele platformei, precum și de a decide care utilizatori primesc drepturi de postare. Pe lângă toate funcționalitățile disponibile unui utilizator cu drepturi de postare, administratorul beneficiază de acces extins la un dashboard suplimentar, care oferă statistici centralizate despre vânzări, conținut și activitatea generală din aplicație. În plus, acesta are acces la funcționalități precum gestionarea arhivării conținutului și trimiterea de e-mailuri în numele companiei, având astfel un control complet asupra operațiunilor administrative ale platformei. Câteva exemple pentru interfață se pot observa în figurile 4.8. și 4.9.:

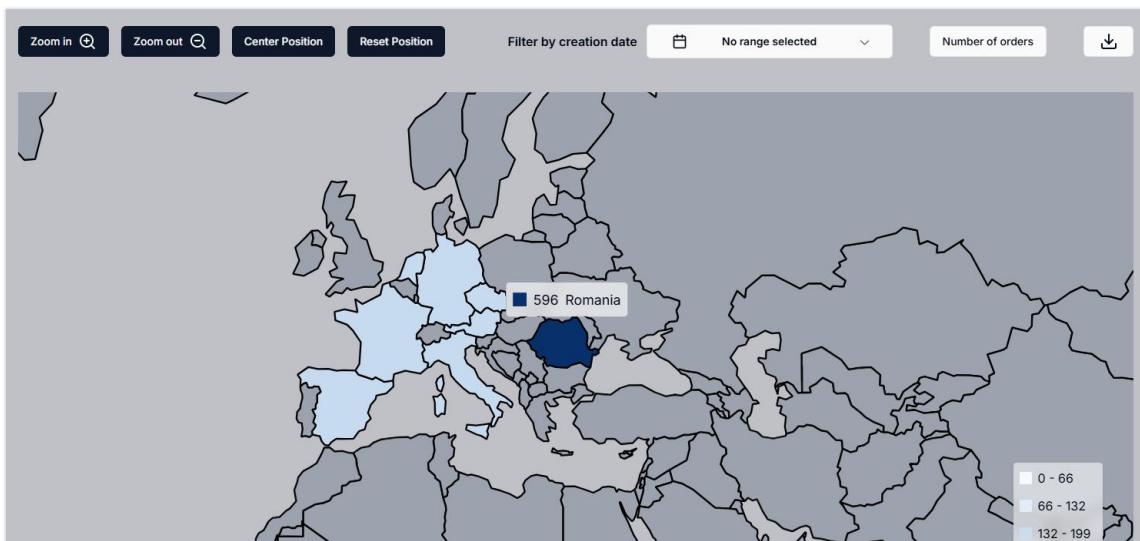


Figura 4.8. Grafic de statistici comenzi în funcție de țări.

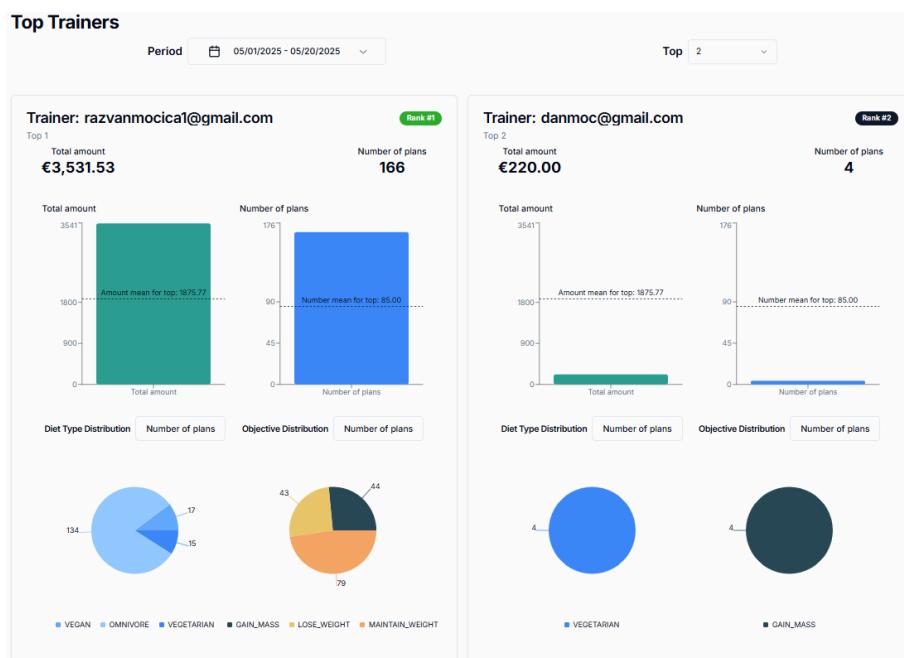


Figura 4.9. Top utilizatori cu drepturi de postare.

Pentru mai multe exemple se pot consulta figurile [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#) și [19](#) din secțiunea Anexe, Figuri. De asemenea, [Figura 4.10.](#) conține diagrama care evidențiază câteva dintre principalele scenarii de utilizare pentru acest tip de utilizator:



Figura 4.10. Diagrama de utilizare pentru utilizatorul cu drepturi de administrare.

CAPITOLUL 5. CONCLUZII

În concluzie, arhitectura prezentată, împreună cu implementarea sa efectivă, pot servi drept model de bună practică în domeniul dezvoltării web moderne. Aceasta integrează componente esențiale pentru realizarea unei aplicații *enterprise* robuste, scalabile și bine structurate, acoperind un spectru larg al unei dezvoltări complexe și complete, respectând totodată ceea ce este deja consacrat în domeniu. Totodată, tehnologiile utilizate nu sunt obligatorii, fiind alese în funcție de specificul aplicației, iar conceptele prezentate sunt agnostice din punct de vedere tehnologic, putând fi adaptate cu ușurință și în alte contexte, precum aplicații mobile sau desktop, datorită caracterului lor general și modular.

Dată fiind complexitatea atât a arhitecturii, cât și a implementării efective, s-au remarcat câteva obstacole majore care au necesitat soluții bine gândite și adaptate contextului, pentru a fi depășite cu succes. Printre acestea se remarcă:

- *Revenirea la origini*: Pentru a putea construi pipeline-uri complexe, precum cel de caching atât în front-end, cât și în back-end, a fost necesară o cercetare aprofundată a caracteristicilor de bază ale limbajelor și librăriilor utilizate. Acest demers a fost determinat de absența unor implementări existente care să răspundă cerințelor specifice ale arhitecturii.

Complexitatea acestor implementări a fost dată de necesitatea unei înțelegeri profunde a mecanismelor de funcționare interne și a abstractizării „low-level” ascunse, întrucât integrarea a trebuit realizată aproape de la zero, acestea fiind probleme specifice, pentru care nu există soluții complete și general valabile ușor de găsit sau aplicat direct.

- *Configurări*: Data fiind complexitatea arhitecturii, a fost necesară realizarea unor configurații variate pentru a asigura integrarea corectă a fiecărei componente. Pentru a reduce complexitatea și riscul de inconsistență, au fost aplicate mai multe strategii, printre care: utilizarea unui server de configurații în *Spring*, definirea de profile distincte pentru componente, recurgerea la template-uri predefinite, în special pentru deployment, precum și integrarea unui flux de *CI/CD*, care automatizează procesele de generare, publicare și testare.
- *Dependența de hardware*: Un aspect neașteptat a fost diferența semnificativă de performanță și comportamentele ascunse legate de asincronie, care pot fi mascate de un hardware performant. Trecerea la o configurație de server bazată pe echipamente mai vechi și mai puțin performante a evidențiat numeroase limitări, impunând optimizări suplimentare și o provizionare atentă a resurselor, pentru a menține stabilitatea și eficiența aplicației în condiții mai restrictive.

Atât arhitectura, cât și implementarea acesteia suportă anumite îmbunătățiri, atât din punct de vedere funcțional, cât și din cel al încapsulării logicii:

- *Integrarea unui warehouse*: În prezent, mecanismul de arhivare salvează elementele local, pe disc, în format JSON. Pentru a extinde considerabil utilitatea acestei funcționalități, se poate integra o bază de date de tip data warehouse, precum *Oracle* sau *HBase*, care să permită centralizarea, analiza și vizualizarea eficientă a datelor arhivate.
- *Extrapolarea în librării*: Având în vedere gradul de generalitate și natura specifică a problemelor rezolvate de anumite module din implementarea actuală, precum caching, serviciul de media și de căutare semantică în baza de date cu *AI*, realizarea cererilor în mod reactiv în front-end, aceste componente ar putea fi extinse și apoi extrapolate sub forma de librării open-source. O astfel de abordare ar putea sprijini comunitatea de dezvoltatori care se confruntă cu provocări similare și, totodată, ar crea oportunitatea de a beneficia de contribuții externe, îmbunătățind soluțiile prin feedback și colaborare activă.

În final, utilitatea și calitatea conceptelor prezentate în lucrare sunt susținute de implementarea practică, care reușește să integreze coerent toate componentele teoretice într-un ansamblu funcțional, relevant din punct de vedere tehnologic, astfel oferindu-i utilizatorului final o experiență care să satisfacă și cele mai exigente cerințe.

BIBLIOGRAFIE

- [1] Spring, *@RabbitListener with Batching*, <https://docs.spring.io/spring-amqp/reference/amqp/receiving-messages/batch.html> [accesat: 05.05.2025]
- [2] PostgreSQL, *About*, <https://www.postgresql.org/about/> [accesat: 12.04.2025]
- [3] KeyDB, *About KeyDB*, <https://docs.keydb.dev/docs/about> [accesat: 12.04.2025]
- [4] PostgreSQL, *Arrays*, <https://www.postgresql.org/docs/current/arrays.html> [accesat: 12.04.2025]
- [5] Spring, *Aspect Oriented Programming with Spring*, <https://docs.spring.io/spring-framework/reference/core/aop.html> [accesat: 17.05.2025]
- [6] MDN, *AsyncIterator*, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/AsyncIterator [accesat: 18.05.2025]
- [7] MDN, *Cache-Control header*, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Cache-Control> [accesat: 14.05.2025]
- [8] AWS, *Caching Overview*, <https://aws.amazon.com/caching/> [accesat: 18.05.2025]
- [9] MDN, *Callback function*, https://developer.mozilla.org/en-US/docs/Glossary/Callback_function [accesat: 18.05.2025]
- [10] RabbitMQ, *Consumer Acknowledgements and Publisher Confirms*, <https://www.rabbitmq.com/docs/confirms> [accesat: 05.05.2025]
- [11] SWR, *Deduplication*, <https://swr.vercel.app/docs/advanced/performance.en-US#deduplication> [accesat: 18.05.2025]
- [12] Lovisa Johansson, *FAQ: What is AMQP and why is it used in RabbitMQ?*, <https://www.cloudamqp.com/blog/what-is-amqp-and-why-is-it-used-in-rabbitmq.html> [accesat: 12.05.2025]
- [13] MDN, *for await...of*, <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for-await...of> [accesat: 18.05.2025]
- [14] PostgreSQL, *GIN Indexes*, <https://www.postgresql.org/docs/current/gin.html> [accesat: 15.04.2025]
- [15] MongoDB, *GridFS for Self-Managed Deployments*, <https://www.mongodb.com/docs/manual/core/gridfs/> [accesat: 19.04.2025]
- [16] Loredana Crusoveanu, *Guide to Resilience4j*, <https://www.baeldung.com/resilience4j> [accesat: 29.04.2025]
- [17] MDN, *HTTP range requests*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Range_requests [accesat: 06.05.2025]

- [18] TURING, *Introduction to Functional Programming*,
<https://www.turing.com/kb/introduction-to-functional-programming> [accesat: 12.04.2025]
- [19] JWT, *Introduction to JSON Web Tokens*, <https://jwt.io/introduction> [accesat: 18.05.2025]
- [20] Nil Seri, *Java's Virtual vs. Platform Threads and What's New in JDK 24*,
<https://senoritadeveloper.medium.com/javas-virtual-vs-platform-threads-and-whats-new-in-jdk-24-22de93f51a74> [accesat: 27.04.2025]
- [21] JWT, *JSON Web Token (JWT) Debugger*, <https://jwt.io/> [accesat: 18.05.2025]
- [22] Github, *KeyDB*, <https://github.com/Snapchat/KeyDB> [accesat: 12.04.2025]
- [23] DataStax, *LangChain JS: How to Get Started Faster & Easier*,
<https://www.datastax.com/guides/what-is-langchain-js> [accesat: 23.05.2025]
- [24] CloudNativePG, *Main features*, <https://cloudnative-pg.io/documentation/1.25/#main-features> [accesat: 24.05.2025]
- [25] Microk8s, *MicroK8s documentation - home*, <https://microk8s.io/docs> [accesat: 21.05.2025]
- [26] MuleSoft, *Newline Delimited JSON (ndjson) Format*,
<https://docs.mulesoft.com/dataweave/latest/dataweave-formats-ndjson> [accesat: 19.04.2025]
- [27] NGINX, *NGINX Content Caching*, <https://docs.nginx.com/admin-guide/content-cache/content-caching/> [accesat: 19.05.2025]
- [28] Nvidia, *NVIDIA Tensor Cores*, <https://www.nvidia.com/en-us/data-center/tensor-cores/> [accesat: 24.05.2025]
- [29] Kubernetes, *Overview*, <https://kubernetes.io/docs/concepts/overview/> [accesat: 19.05.2025]
- [30] Github, *pgvector*, <https://github.com/pgvector/pgvector> [accesat: 12.04.2025]
- [31] Baeldung, *RabbitMQ Message Dispatching with Spring AMQP*,
<https://www.baeldung.com/rabbitmq-spring-amqp> [accesat: 07.05.2025]
- [32] RabbitMQ, *RabbitMQ tutorial - Remote procedure call (RPC)*,
<https://www.rabbitmq.com/tutorials/tutorial-six-python> [accesat: 09.05.2025]
- [33] Ankit Rana, *Request Deduplication Pattern: Introduction*,
<https://www.linkedin.com/pulse/request-deduplication-pattern-introduction-ankit-rana/> [accesat: 23.05.2025]
- [34] STOMP, *STOMP Protocol Specification, Version 1.2*, <https://stomp.github.io/stomp-specification-1.2.html> [accesat: 26.04.2025]
- [35] Bonér, Jonas, Dave Farley, Roland Kuhn, și Martin Thompson. 2014, *The Reactive Manifesto*, <https://www.reactivemanifesto.org/> [accesat: 08.04.2025]

- [36] Sridharrajdevelopment, *Thread Per Request VS WebFlux VS VirtualThreads*,
<https://medium.com/@sridharrajdevelopment/thread-per-request-vs-virtualthreads-vs-webflux-33c9089d22fb> [accesat: 02.04.2025]
- [37] Lukas Fittl, *Understanding Postgres GIN Indexes: The Good and the Bad*,
<https://pganalyze.com/blog/gin-index> [accesat: 15.04.2025]
- [38] Tyler Mitchell, *What are Embedding Models? An Overview*,
<https://www.couchbase.com/blog/embedding-models/> [accesat: 21.04.2025]
- [39] Cloudflare, *What are embeddings in machine learning?*,
<https://www.cloudflare.com/learning/ai/what-are-embeddings/> [accesat: 21.04.2025]
- [40] DataStax, *What are Hierarchical Navigable Small Worlds (HNSW)?*,
<https://www.datastax.com/guides/hierarchical-navigable-small-worlds> [accesat: 23.04.2025]
- [41] Dave Bergmann, *What is a context window?*, <https://www.ibm.com/think/topics/context-window> [accesat: 21.05.2025]
- [42] RedHat, *What is a REST API?*, <https://www.redhat.com/en/topics/api/what-is-a-rest-api>
[accesat: 02.04.2025]
- [43] Miguel Rebelo, *What is Hugging Face?*, <https://zapier.com/blog/hugging-face/> [accesat: 21.05.2025]
- [44] Jakub Dakowicz, *What is Next.JS and Why Should You Use It in 2025?*,
<https://pagepro.co/blog/what-is-nextjs> [accesat: 27.04.2025]
- [45] NextJS, *What is NextJS?*, <https://nextjs.org/docs#what-is-nextjs> [accesat: 27.04.2025]
- [46] auth0, *What is OAuth 2.0?*, <https://auth0.com/intro-to-iam/what-is-oauth-2> [accesat: 28.04.2025]
- [47] Ariffud M., *What is Ollama? Understanding how it works, main features and models*,
<https://www.hostinger.com/tutorials/what-is-ollama> [accesat: 14.05.2025]
- [48] EDB, *What is pgvector, and How Can It Help Your Vector Database?*,
<https://www.enterprisedb.com/blog/what-is-pgvector> [accesat: 16.04.2025]
- [49] Aleksandar Pavlov, *What is Redis?*, <https://adevait.com/redis/what-is-redis> [accesat: 12.04.2025]
- [50] iamthatsoftwareguy, *When to Use Spring Boot WebFlux (And When Not To)*,
<https://medium.com/@iamthatsoftwareguy/as-someone-whos-been-working-with-spring-boot-for-the-last-8-years-i-ve-had-my-fair-share-of-3f90a063cd2c> [accesat: 07.04.2025]

ANEXE

Definiții

REST API (Representational State Transfer Application Interface)

Un *API REST* este o interfață de programare care respectă principiile arhitecturale *REST* și permite comunicarea între client și server prin intermediul protocolului *HTTP*. Acestea sunt interfețe simple și scalabile, bazate pe principii precum stateless (fără stare), cache-abilitate, interfață uniformă și arhitectură pe straturi, permitând interacțiuni standardizate și eficiente între client și server. Datorită flexibilității, sunt larg utilizate în dezvoltarea aplicațiilor web. [\[42\]](#)

Programare Reactivă

Programarea reactivă este o paradigmă care se bazează pe operațiuni asincrone, non-blocante și pe propagarea schimbărilor de date. Ea permite sistemelor să reacționeze în timp real la evenimente sau actualizări de date, oferind astfel scalabilitate și un răspuns rapid. Arhitectura reactivă, definită prin *Reactive Manifesto*, se bazează pe patru principii esențiale: responsivitate, reziliență, elasticitate și mesaje asincrone, oferind un cadru solid pentru construirea sistemelor moderne, scalabile și fiabile, capabile să se adapteze dinamic și să funcționeze eficient chiar și în condiții de stres sau eroare. [\[35\]](#)

AMQP (Advanced Message Queuing Protocol)

AMQP definește un set de capabilități pentru stocarea mesajelor și redirecționarea acestora, care trebuie să fie puse la dispoziție printr-o implementare de server compatibilă (i.e. un *broker*). De asemenea, este definită și o schemă de codificare a mesajelor.

Structura protocol se poate observa în [Figura Anexe-Definiții 1](#):

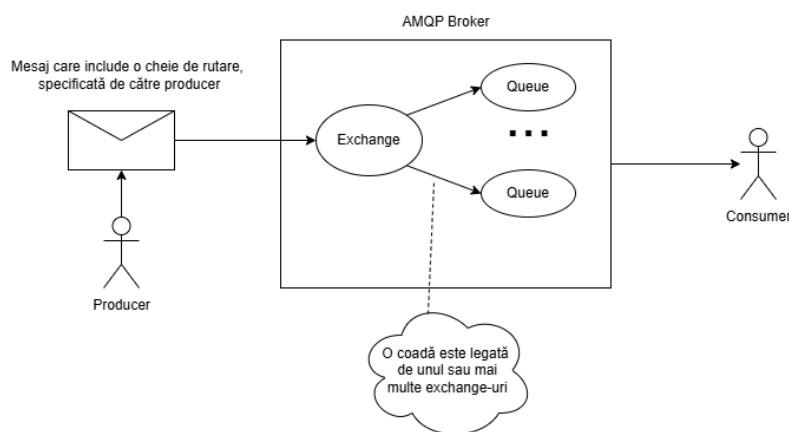


Figura Anexe-Definiții 1. Diagrama AMQP.

Un client, denumit producer, trimite un mesaj către un broker, mai exact către un exchange din cadrul acestuia. Exchange-ul are scopul de a distribui conținutul mesajului către una sau mai multe cozi, respectând anumite criterii. În final, mesajul este trimis către unul sau mai mulți clienți, denumiți consumatori. [12]

GridFS

GridFS este un mecanism de stocare a fișierelor mari în cadrul *MongoDB*, utilizat atunci când fișierele depășesc limita de *16MB* impusă pentru documentele individuale. Sistemul este alcătuit din două colecții principale:

„files”: reține metadatele fișierelor

„chunks”: reține în format binar conținutul unei bucăți al unui singur fișier.

Mecanismul este ilustrat în [Figura Anexe-Definiții 2](#):

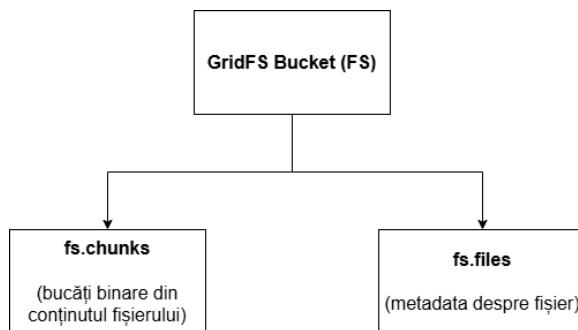


Figura Anexe-Definiții 2. Arhitectura GridFS.

Fiecare chunk al unui fișier este reținut într-un document separat. *Mongo* adaugă în mod automat fiecărui document un identificator unic cu numele *_id*, iar aceste două colecții vor fi legate printr-un „foreign key” *files_id* în *chunk*. De asemenea, sunt creați indecsăi atât pentru *_id* din ambele colecții, cât și pentru *files_id* și tuplul (*files_id, n*) din colecția chunks.

Se pot obține doar anumite părți ale unui fișier, nu tot fișierul deodată, astfel se eficientizează procesul de manipulare a datelor, încărcându-se în memorie minimum necesar, fiind în special util pentru videoclipuri care vor face request-uri de tip *Range*.

Chunking-ul se poate observa în [Figura Anexe-Definiții 3](#):

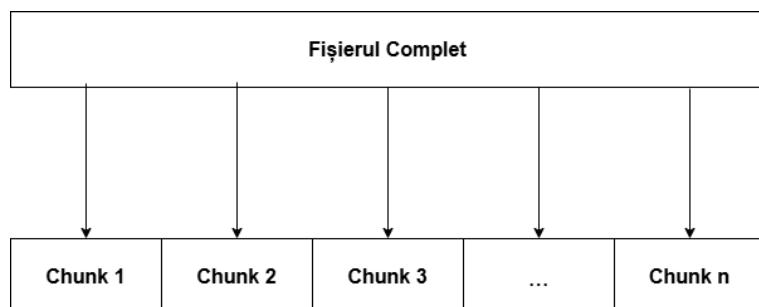


Figura Anexe-Definiții 3. Chunking în GridFS.

Fiecare chunk, mai puțin ultimul, va avea dimensiunea specificată, iar ultimul va conține restul fișierului. [\[15\]](#)

Application/x-ndjson

Application/x-ndjson este un format asemănător cu *Application/json*, mai exact, obiecte JSON, fiecare nou obiect JSON fiind delimitat de un „/n”. De exemplu, pentru o listă în JSON:

```
[ { "id": 1, "nume": "Razvan" },
  { "id": 2, "nume": "Andrei" },
  { "id": 3, "nume": "Mihai" },
  { "id": 4, "nume": "Alex" } ]
```

Aceasta va fi primită astfel:

```
{ "id": 1, "nume": "Razvan" }/n
{ "id": 2, "nume": "Andrei" }/n
{ "id": 3, "nume": "Mihai" }/n
{ "id": 4, "nume": "Alex" }/n
```

Folosind acest format, se aderă la paradigma reactivă, anume de a nu bloca sau aștepta pentru terminarea stream-ului. De asemenea, acest format este nativ în funcția *fetch* din JS, nefiind nevoie de alte librării adiacente. [\[26\]](#)

Programare Funcțională

Programarea funcțională este o paradigmă de programare bazată pe utilizarea funcțiilor pure. În acest stil, funcțiile nu au efecte secundare, iar datele sunt tratate ca imutabile. Programarea funcțională pune accent pe expresii și compozitia funcțiilor, în locul instrucțiunilor secvențiale, caracteristice programării imperative. Un mare avantaj al programării funcționale este ușurința în paralelizare. [\[18\]](#)

WebSocket

WebSocket este un protocol de comunicare bidirectională între server și client, în care oricare dintre participanți poate trimite mesaje. O conexiune de acest tip persistă atât timp cât oricare dintre părțile implicate nu renunță. Odată ce un participant îintrerupe conexiunea, în mod automat se îintrerupe și pentru cealaltă parte implicată. De asemenea, există și numeroase standarde pentru acest protocol, în scopul de a ușura sincronizarea mesajelor. [\[34\]](#)

RPC (Remote Procedure Call)

În modelul *RPC* (Remote Procedure Call), un client poate apela o funcție pe un server precum ar fi locală, dar de fapt apelul este transmis prin mesaje. Se creează o coadă pentru cereri și una pentru răspunsuri. Serverul ascultă cererile, le procesează și trimite rezultatul

înapoi clientului. *Spring AMQP* oferă automat sincronizarea mesajelor din cozile de *RPC*, astfel nefiind nevoie de o sincronizare manuală a răspunsului corect cu cererea. [\[32\]](#)

Caching

Caching-ul reprezintă o strategie fundamentală de optimizare a relației dintre timp și resurse în sistemele informatici. Esența caching-ului stă în anticiparea nevoilor viitoare pe baza experienței trecute, păstrarea în apropiere a ceea ce este probabil să fie din nou necesar. Este o formă de memorie practică, o încercare de a contracara natura lentă și repetitivă a procesului de calcul, apelând la redundanța controlată pentru a obține eficiență.

Prin caching se face un compromis între spațiu și timp: alocăm resurse pentru a reține temporar date, evitând astfel reluarea unor procese complexe sau costisitoare. Ideologic, caching-ul reflectă un principiu mai larg al tehnologiei, dorința de a minimiza efortul viitor prin investiții inteligente în prezent, de a valorifica trecerea timpului în favoarea unei reacții mai rapide, mai fluide. În general, dimensiunea unui cache este mult mai scăzută comparativ cu baza de date principală, este ținut în memoria *RAM* pentru a fi rapid accesat și nu trebuie supus normalizărilor bazelor de date primare/clasice. [\[8\]](#)

Aspecte și Spring AOP

Un aspect reprezintă o parte a aplicației care grupează o anumită logică ce nu aparține direct funcționalității de bază, dar care este totuși necesară în mai multe locuri, cum ar fi logarea, gestionarea tranzacțiilor sau verificarea securității. În loc să se scrie această logică repetat în fiecare metodă unde este necesară, se definește o singură dată, într-un aspect, iar aceasta va fi aplicată automat acolo unde este necesar. Aspectul stabilește ce anume trebuie să se întâmpile (de exemplu, salvarea unui mesaj într-un fișier de log) și în ce moment al execuției codului trebuie să se întâmpile acest lucru (de exemplu, înainte sau după rularea unei metode). Astfel, aspectele ajută la organizarea codului într-un mod mai clar și mai curat, separând preocupările generale de logica principală a aplicației. (Aspectul este asemănător în paradigmă unui decorator din alte limbi, sau chiar a unei funcții *wrapper* în anumite cazuri.)

Spring AOP (Aspect-Oriented Programming) este un modul din cadrul framework-ului Spring, care permite separarea logicii transversale (*cross-cutting concerns*) de logica principală a aplicației. Prin *AOP*, funcționalități precum logarea, securitate, tranzacții sau monitorizare pot fi implementate sub forma unor aspecte, aplicate automat în puncte cheie ale codului (numite *join points*), fără a modifica logica de bază.

Spring AOP funcționează pe baza *proxy-urilor* și este ideal pentru integrarea aspectelor în aplicații bazate pe *Spring*, folosind concepte precum *advice*, *pointcut* și *weaving*.

Este o implementare simplificată, care facilitează reutilizarea logicii. Un avantaj major al acestei implementări este ușurința cu care aspectele pot fi aplicate pe metode prin definirea unor adnotări proprii, astfel codul pentru care aspectul va fi aplicat nu trebuie să respecte multe standarde predefinite. [5]

Fanout Exchange

RabbitMQ Fanout Exchange trimitie fiecare mesaj primit către toate cozile conectate, indiferent de cheia de rutare, fiind ideal pentru broadcast (difuzare). Cozile anonime nedurabile sunt create temporar, fără un nume specificat (generate automat de către *Spring AMQP* în cazul de față), nu sunt persistente și se sterg automat la închiderea conexiunii. [31]

Deduplicare

Deduplicarea cererilor este un mecanism utilizat pentru a se asigura că o anumită cerere este procesată o singură dată, chiar dacă este trimisă de mai multe ori. Acest lucru este deosebit de util pentru metodele non-idempotente, în cazul în care reluarea unei cereri ar putea duce la duplicarea neintenționată a acțiunilor. Fiecarei cereri i se atribuie un identificator unic, adică o cheie de deduplicare a mesajelor, pentru a-i urmări unicitatea. Dacă sistemul detectează o cerere duplicată într-o fereastră de timp specificată, acesta confirmă cererea, dar nu o procesează din nou, prevenind astfel acțiunile duplicate. [33]

Pentru aplicațiile web, librării precum cea *SWR* utilizează un interval de deduplicare pentru a gestiona deduplicarea cererilor. În mod implicit, intervalul de deduplicare al *SWR* este setat la 2.000ms, dar se recomandă să fie configurat în funcție de cerințele specifice ale aplicațiilor. [11]

De asemenea, deduplicarea este utilă în momentul în care pe o pagină există elemente care conduc la aceleași request-uri adiacente în cascadă, de exemplu autorul unei postări.

Moduri De Asincronie

Un *callback* este o funcție transmisă ca argument altuii funcții, care este apelată ulterior, de obicei după finalizarea unei operațiuni asincrone. [9]

Un *AsyncIterator* este o interfață ce permite iterarea asupra unui flux de date asincron, oferind metode precum *next()* care returnează o promisiune ce rezolvă valoarea următoare. Este util pentru procesarea treptată a datelor care nu sunt disponibile simultan, cum ar fi stream-urile. [6]

Un *AsyncGenerator* este o funcție specială, declarată asincronă, care combină funcționalitatea generatoarelor cu asincronia. Aceasta permite emiterea de valori asincron,

folosind *await* în interiorul funcției și *yield* pentru a returna valorile succesiv. *AsyncGenerator*-ele implementează automat interfața *AsyncIterator*, fiind utile pentru definirea surselor de date asincrone iterabile, cum ar fi iterarea în cadrul unui stream care survine în urma uneia sau mai multor promisiuni. [13]

Trecerea de la *callbacks* la *AsyncIterator* și *AsyncGenerator* se face prin schimbarea modelului de gestionare a fluxurilor de date asincrone, de la un model bazat pe „push” (*callbacks*), la unul „pull” controlat de consumator.

- *Callbacks (Push)*: În modelul *callback*, producătorul de date „împinge” rezultatul către consumator imediat ce acesta devine disponibil, apelând o funcție *callback*. Controlul fluxului este la producător, iar consumatorul trebuie să fie pregătit să reacționeze oricând.
- *AsyncIterator (Pull)*: Trecerea către *AsyncIterator* deleagă consumatorul să ceară datele atunci când are nevoie de ele. Astfel, controlul fluxului este preluat de consumator.
- *AsyncGenerator*: *AsyncGenerator* extinde ideea anterioară, permitând să se creeze ușor surse de date asincrone, folosind *yield* pentru a emite valori și *await* pentru a aștepta evenimente. Astfel, se creează o legătură naturală între producător și consumator, ambii având un control mai echilibrat. Pentru o secțiune de cod care gestionează deduplicarea, folosind *AsyncGenerator*, se poate consulta [Extras Cod-Sursă 2 din Anexe, Secvențe de Cod](#).

Browser Cache Headers

Browser Cache-Control se referă la un set de headere-uri *HTTP* (precum *Cache-Control*, *Expires*, *ETag*, și *Last-Modified*) care indică browser-ului cum să gestioneze stocarea și reutilizarea resurselor. Prin aceste antete, serverul poate controla dacă o resursă poate fi stocată local, cât timp poate fi considerată validă și când trebuie revalidată cu serverul. Acest tip de caching este util în special în cazuri în care datele sunt imutabile și de dimensiuni mai mari, dar nu excesive, fiind perfect pentru imagini, de exemplu. [7]

NGINX Caching

NGINX caching este o metodă prin care serverul web *NGINX*, în cazul de față *ingress-ul* din k8s, poate păstra temporar copii ale răspunsurilor trimise către utilizatori, astfel încât la următoarele cereri, aceleași date să fie livrate mai rapid, fără a le genera din nou. Acest tip de cache ajută la îmbunătățirea vitezei site-ului și la reducerea încărcării pe server, fiind util în special pentru conținut care nu se schimbă frecvent, cum ar fi imaginile. Cache-ul este unul asemănător cu tipul *LRU*, astfel făcându-se o curățare automată a resurselor nefolosite recent. De asemenea, *NGINX* este capabil să folosească header-ele de cache *HTTP*

(pentru detalii, [Anexe](#), [Definiții](#), [Browser Cache Headers](#)) de la *upstream* pentru a salva datele, astfel făcând integrarea cu cache-ul mult mai facilă. [\[27\]](#)

Range Requests

Range Request este o funcționalitate a protocolului *HTTP* care permite browser-ului sau altor clienți să ceară doar o parte dintr-un fișier, nu întregul. Aceasta este utilă mai ales pentru fișiere mari, cum ar fi videoclipuri sau documente *PDF*, permitând încărcarea progresivă sau reluarea descărcărilor întrerupte. Serverul răspunde cu antetul *Content-Range*, indicând ce porțiune din fișier a fost trimisă și cu status-ul 206 indicând un răspuns parțial. [\[17\]](#)

Embedding

Modelul de embedding este un model de tip *encoder*, iar embedding-ul în sine este o tehnică utilizată în învățarea automată pentru a transforma datele discrete, cum ar fi cuvintele, în reprezentări numerice dense (vectori) într-un spațiu continuu. Aceste reprezentări păstrează relații semantice între entități, de exemplu, cuvintele cu semnificații similare vor avea vectori apropiati. Modelele de embedding sunt esențiale în procesarea limbajului natural (*NLP*), permitând algoritmilor să „înțeleagă” și să proceseze textul într-un mod numeric. Există numeroase modele de embedding preantrenate ce pot fi folosite, fapt ce elimină nevoia antrenării de la zero a unui astfel de model. [\[39\]](#) [\[38\]](#)

HNSW (Hierarchical Navigable Small World)

HNSW (Hierarchical Navigable Small World) este un algoritm eficient pentru căutarea aproximativă a celor mai apropiati vecini într-un spațiu vectorial. Acesta construiește o structură ierarhică de grafuri care permite navigarea rapidă între puncte, reducând semnificativ timpul de căutare în seturi mari de date vectoriale, păstrând în același timp o acuratețe ridicată. [\[40\]](#)

În contextul bazelor de date vectoriale, extensia *PgVector* pentru *PostgreSQL* permite stocarea și interogarea vectorilor de dimensiuni mari, cum ar fi cei de embedding, direct în baza de date relațională. *PgVector* are suport pentru algoritmul *HNSW*, oferind capabilități avansate de căutare semantică și similaritate. [\[30\]](#)

De exemplu, în implementarea prezentată pentru titlul unei postări, crearea unui index de tip *HNSW* folosind *PgVector*, cu 16 muchii între nodurile grafului „(m = 16)” și cu 64 de vecini considerați în timpul construirii „(ef_construction = 64)” utilizând produsul scalar pentru măsurarea distanței dintre vectori, se realizează astfel:

```
CREATE INDEX IF NOT EXISTS hnsw_post ON post_embedding USING hnsw(embedding vector_ip_ops) WITH(m = 16, ef_construction = 64);
```

Conextul Unui LLM

Contextul unui *LLM* reprezintă totalitatea informațiilor furnizate modelului sub formă de text înainte de a genera un răspuns; practic, este „memoria pe termen scurt” a modelului în timpul unei interacțiuni. Acest context include atât promptul utilizatorului, istoricul conversației, cât și eventuale informații relevante extrase dintr-o bază de date vectorială, fiind limitat de o dimensiune maximă exprimată în tokeni (unități textuale).

Procesarea contextului este costisitoare deoarece fiecare token adăugat crește volumul de calcule necesar pentru inferență. Astfel, contextul lung consumă semnificativ mai multă memorie *VRAM* și resurse de procesare, în special pe modele mari, unde timpul de răspuns și cerințele hardware cresc exponențial. [\[41\]](#)

OAuth2

OAuth2 este un protocol de autorizare standardizat, care permite unei aplicații să obțină acces delegat și securizat la resursele unui utilizator, găzduite pe un alt serviciu, fără a solicita parola acestuia. În loc să partajeze credențialele, utilizatorul autorizează aplicația să acționeze în numele său, obținând un token de acces, care este folosit ulterior pentru a accesa resursele dorite. *OAuth2* este utilizat pe scară largă în scenarii precum conectarea cu conturi de *Google* sau *GitHub*, și stă la baza unor soluții de autentificare avansate. În acest mod, utilizatorul nu este obligat să-și creeze un cont nou pentru anumite platforme și poate gestiona într-un mod globalizat autentificarea. [\[46\]](#)

Spring AMQP Batch RabbitMQ Listener

Batch container listener în *RabbitMQ* este un mecanism disponibil în cadrul *Spring AMQP*, care permite procesarea mai multor mesaje simultan într-o singură unitate logică, sub forma unui batch. În loc să primească fiecare mesaj către metodă individual, containerul colectează un set de mesaje și le procesează împreună, reducând astfel costul operațiunilor *I/O* și crescând eficiența în scenarii de trafic ridicat.

Acest model este util în aplicațiile care trebuie să prelucreze rapid cantități mari de mesaje, cum ar fi agregări, scrieri în baze de date sau procesări de fluxuri. Este important de menționat că listener-ul de tip batch poate afecta semantica de confirmare (*acknowledgment*), motiv pentru care trebuie configurat cu atenție, mai ales în contextul livrărilor garantate. Se poate opta pentru a confirma tot batch-ul deodată sau fiecare mesaj din batch, preferându-se prima variantă. [\[1\]](#) [\[10\]](#)

JWT

JSON Web Token (JWT) este un standard care definește o modalitate compactă și autonomă de transmitere securizată a informațiilor între părți sub forma unui obiect *JSON*. Aceste informații pot fi verificate și validate deoarece sunt semnate digital. JWT-urile pot fi semnate folosind un secret (cu algoritmul *HMAC*) sau o pereche cheie publică/privată folosind *RSA* sau *ECDSA*.

Autorizarea este cel mai comun scenariu pentru JWT. Odată ce utilizatorul este autentificat, fiecare solicitare ulterioară va include JWT, permitându-i utilizatorului să acceseze rutele, serviciile și resursele care sunt permise cu acel token.

Structura unui JWT este:

- *Header*: are două părți, tipul token-ului și algoritmul de semnare.
- *Payload*: conținutul efectiv al token-ului.
- *Semnătura*: confirmă identitatea celui care transmite token-ul și integritatea mesajului, fiind derivat din header și payload.

[19]

De exemplu, un JWT folosit în cadrul aplicației are forma prezentată în [Figura Anexe-Definiții 4](#):

The screenshot shows the jwt.io interface with the following sections:

- ENCODED VALUE:** A large text area containing a long base64 encoded string:

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJyYXp2YWStb2NpY2ExQGdtYwlsLmNvbSISInJvbGVzIjpbeyJhdXRob3JpdHkiOiJST0xFX0FETUlOIn1dLCJlbWVpbC16InJhenZhbmtvY2ljb21haWwvY29tLiwiCHJvdmlkZXIiOiJMT0NBTCiS1m1hdC16MTc0NjAxNzI2OCw1ZXhwIjoxNzQ5NjE3MjY4fQ.CuG9g1TnTKIXkOiyqTSurHw3WNvqTn0q84-bAOjb8I
```
- DECODED HEADER:** A JSON table showing the header:

```
{ "alg": "HS256" }
```
- DECODED PAYLOAD:** A JSON table showing the payload:

```
{ "sub": "razvanmocica1@gmail.com", "roles": [ { "authority": "ROLE_ADMIN" } ], "email": "razvanmocica1@gmail.com", "provider": "LOCAL", "iat": 1746817268, "exp": 1749617268 }
```
- JWT SIGNATURE VERIFICATION (OPTIONAL):** A section for entering a secret key, currently showing "Valid secret".

Figura Anexe-Definiții 4. Exemplu de JWT, folosind resursa [\[21\]](#).

Completări

Comparație Paradigme

Pentru a arăta diferențele de performanță am implementat și o aplicație de test care comunică doar cu o bază de date și face o mică procesare. Am implementat aplicația atât cu *virtual threads* cât și cu *WebFlux*, iar rezultatele confirmă că deși thread-urile virtuale se apropie ca performanță de programarea reactivă, aceasta din urmă este mult mai constantă și predictibilă și scalează mai „logaritmic” cu creșterea concurenței. De asemenea, cu cât se introduce un delay artificial (de câteva ms) în test (în mod normal acest delay este inevitabil uneori la interogări de baze de date, accesări de cache, sau efectiv întârzieri naturale ale utilizării masive unui server), *WebFlux* își crește și mai mult avantajul, având o schimbare mult mai mică, în timp ce schimbările în *virtual threads* sunt semnificative, ceea ce arată încă un avantaj al programării reactive, anume predictibilitatea. Acest avantaj se poate observa în cadrul figurilor Anexe-Completări [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#) și [10](#).

- Latență 0ms:
 - 5.000 cereri, 500 concurente
 - WebFlux

Percentage of the requests served within a certain time (ms)	
50%	173
66%	175
75%	176
80%	177
90%	180
95%	182
98%	182
99%	182
100%	196 (longest request)

Figura Anexe-Completări 1. Rezultat comparație pentru latență 0ms, 5.000 cereri, 500 concurente pentru WebFlux.

- Virtual Threads

Percentage of the requests served within a certain time (ms)	
50%	151
66%	165
75%	176
80%	181
90%	1080
95%	1243
98%	1392
99%	1629
100%	3163 (longest request)

Figura Anexe-Completări 2. Rezultat comparație pentru latență 0ms, 5.000 cereri, 500 concurente pentru Virtual Threads.

- 10.000 cereri, 1.000 concurente

- WebFlux

Percentage of the requests served within a certain time (ms)	
50%	366
66%	370
75%	375
80%	377
90%	392
95%	405
98%	407
99%	407
100%	409 (longest request)

Figura Anexe-Completări 3. Rezultat comparație pentru latență 0ms, 10.000 cereri, 1.000 concurente pentru WebFlux.

- Virtual Threads

Percentage of the requests served within a certain time (ms)	
50%	283
66%	358
75%	365
80%	368
90%	382
95%	1585
98%	3435
99%	3471
100%	7199 (longest request)

Figura Anexe-Completări 4. Rezultat comparație pentru latență 0ms, 10.000 cereri, 1.000 concurente pentru Virtual Threads.

- 20.000 cereri, 2.000 concurente

- WebFlux

Percentage of the requests served within a certain time (ms)	
50%	732
66%	738
75%	742
80%	744
90%	749
95%	755
98%	760
99%	763
100%	804 (longest request)

Figura Anexe-Completări 5. Rezultat comparație pentru latență 0ms, 20.000 cereri, 2.000 concurente pentru WebFlux.

- Virtual Threads

Percentage of the requests served within a certain time (ms)	
50%	677
66%	686
75%	689
80%	697
90%	1203
95%	2327
98%	3766
99%	5266
100%	8325 (longest request)

Figura Anexe-Completări 6. Rezultat comparație pentru latență 0ms, 20.000 cereri, 2.000 concurente pentru Virtual Threads.

- Latență 2ms
 - 20.000 cereri, 2.000 concurente
 - WebFlux

Percentage of the requests served within a certain time (ms)	
50%	760
66%	780
75%	785
80%	789
90%	795
95%	811
98%	824
99%	831
100%	839 (longest request)

Figura Anexe-Completări 7. Rezultat comparație pentru latență 2ms, 20.000 cereri, 2.000 concurente pentru WebFlux.

- Virtual Threads

Percentage of the requests served within a certain time (ms)	
50%	1008
66%	1019
75%	1022
80%	1026
90%	1044
95%	1050
98%	1054
99%	1056
100%	2076 (longest request)

Figura Anexe-Completări 8. Rezultat comparație pentru latență 2ms, 20.000 cereri, 2.000 concurente pentru Virtual Threads.

- Latență 5ms
 - 20.000 cereri, 2.000 concurente
 - WebFlux

Percentage of the requests served within a certain time (ms)	
50%	812
66%	824
75%	830
80%	834
90%	844
95%	853
98%	1762
99%	1854
100%	1957 (longest request)

Figura Anexe-Completări 9. Rezultat comparație pentru latență 5ms, 20.000 cereri, 2.000 concurente pentru WebFlux.

- Virtual Threads

Percentage of the requests served within a certain time (ms)	
50%	1634
66%	1644
75%	1652
80%	1656
90%	1672
95%	1683
98%	1689
99%	1692
100%	3331 (longest request)

Figura Anexe-Completări 10. Rezultat comparație pentru latență 5ms, 20.000 cereri, 2.000 concurente pentru Virtual Threads.

Principalele Elemente Suplimentare În Cluster

Componentele adiacente din cadrul clusterului de *Microk8s* al cloud-ului privat sunt:

- *Longhorn*: un sistem distribuit de stocare pentru *Kubernetes*, care oferă volume persistente și replicare automată a datelor, ideal pentru medii *cloud-native*.
- *Grafana*: o platformă open-source pentru vizualizarea datelor și monitorizare, permitând crearea de dashboard-uri dinamice pentru diverse surse de date. Exemple se pot observa în cadrul figurilor Anexe-Completări [11](#), [12](#) și [13](#):

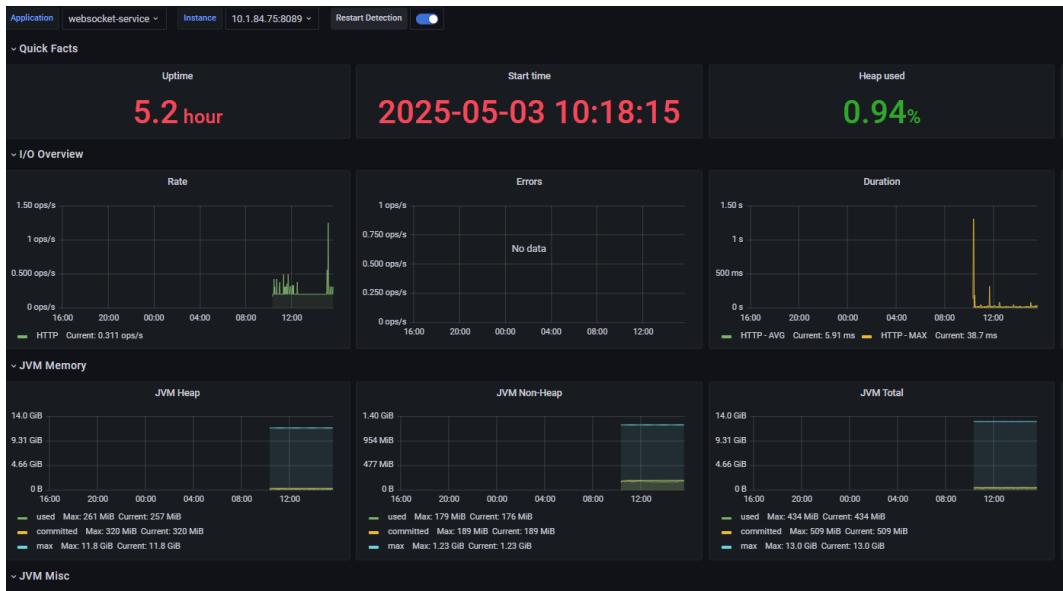


Figura Anexe-Completări 11. Dashboard Grafana pentru un microserviciu de Spring.



Figura Anexe-Completări 12. Dashboard Grafana pentru un nod din cluster.



Figura Anexe-Completări 13. Dashboard Grafana pentru operatorul de CNPG.

- *Prometheus*: un sistem de monitorizare specializat în colectarea metricilor din aplicații și infrastructură, folosit frecvent în ecosistemul *Kubernetes*, fiind ideal pentru vizualizare metricilor în *Grafana*. Astfel, fiecare componentă a arhitecturii prezentate interacționează cu acest serviciu. Un exemplu este ilustrat în [Figura Anexe-Completări 14](#):

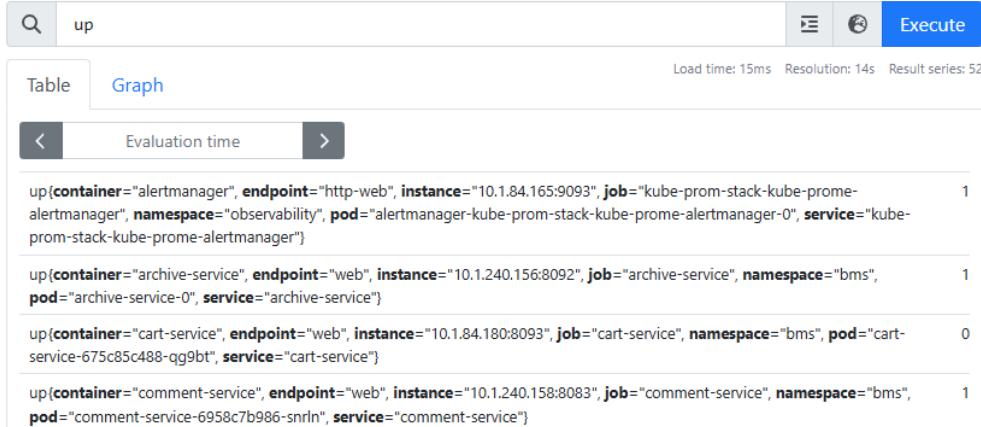


Figura Anexe-Completări 14. Servicii în Prometheus.

- *Loki*: o soluție de „log management” dezvoltată de *Grafana Labs*, optimizată pentru a colecta și indexa jurnalele într-un mod eficient. Un exemplu este ilustrat în [Figura Anexe-Completări 15](#):

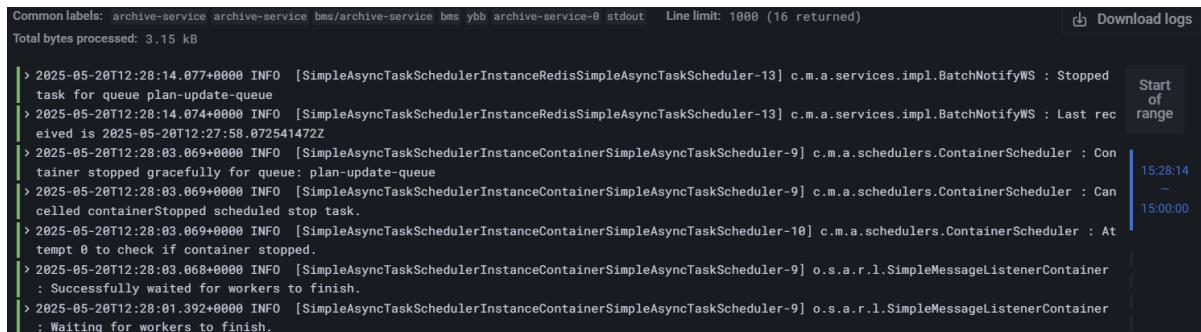


Figura Anexe-Completări 15. Log-uri în Loki pentru un serviciu.

- *Grafana Tempo*: soluție de „distributed tracing” dezvoltată de *Grafana Labs*, care permite urmărirea „end-to-end” a cererilor într-un sistem distribuit, integrându-se ușor cu *Grafana* pentru vizualizarea traseelor, similar cu *Zipkin*. Un exemplu este ilustrat în [Figura Anexe-Completări 16](#):

Trace ID	Trace name	Start time	Duration
682c7b2ee9e9d272bb0050aab4be4edf	gateway-service http get	2025-05-20 15:53:02	28 ms
682c7b2d595bec09558849c7ed48c13c	user-service http post /auth/validateToken	2025-05-20 15:53:01	5 ms
682c7b2d01ec05ffb159f4c25003c7be	user-service http post /auth/validateToken	2025-05-20 15:53:01	18 ms

Figura Anexe-Completări 16. Cerere urmărită în Tempo.

- *NGINX Ingress Controller*: gestionează traficul *HTTP* și *HTTPS* către aplicațiile dintr-un cluster *Kubernetes*, oferind balansare de sarcină și funcționalități avansate de rutare,

certificate *TLS*, caching pentru componente statice, *ModeSecurity* pentru protejarea serviciilor împotriva atacurilor, etc.

- *KEDA (Kubernetes Event-Driven Autoscaling)*: permite scalarea automată a aplicațiilor *Kubernetes* pe baza evenimentelor și metricilor externe, extinzând capacitatele native de scalare automată.
- *CNPG (Cloud Native Postgre)*: un operator *Kubernetes* open-source pentru gestionarea bazelor de date *PostgreSQL*, oferind provizionare automată, replicare și backup-uri în medii *cloud-native*. De asemenea, oferă și integrare automată cu *PgBouncer* pentru a gestiona în mod eficient conexiunile.
- *Bitnami Sharded MongoDB HelmChart*: un „helm chart” care permite *sharding* pentru baze de date *Mongo* și oferă o configurare facilă a acestui *shard*, fiind mai ales util în medii în care volumul de date stocat este ridicat și încodare acestora este consumatoare de resurse, precum stocarea de media.
- *KeyDB Multi-Master Set*: Pe lângă protocolul *Redis* și toate facilitățile principale ale acestuia, *KeyDB* permite și replicare „multi-master”, ceea ce înseamnă că fiecare nod al acestui environment este egal, făcând o comunicare „peer-to-peer”.
- *RabbitMQ Cluster Operator*: este un operator *Kubernetes* care automatizează gestionarea, scalarea și actualizarea clusterelor *RabbitMQ* în medii de k8s.
- *NVIDIA GPU Operator*: automatizează instalarea și gestionarea driverelor și componentelor necesare pentru utilizarea *GPU-urilor* în *Kubernetes*, facilitând rularea aplicațiilor care necesită aceste resurse precum modelele de *AI*.
- *Umami Analytics Helm Chart*: o alternativă open-source și anonimizată pentru *Google Analytics*. Exemple se pot observa în cadrul figurilor Anexe-Completări [17](#) și [18](#):

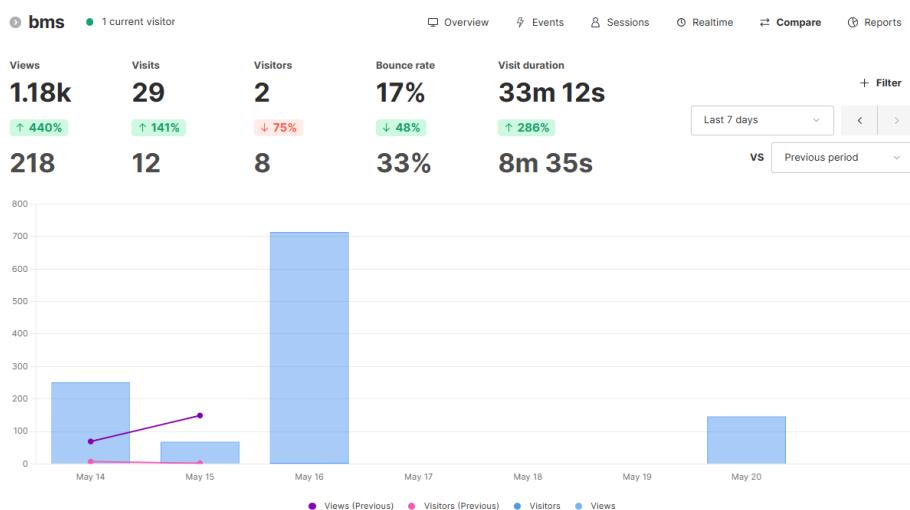


Figura Anexe-Completări 17. Statistici în Umami pentru front-end sumar relativ.

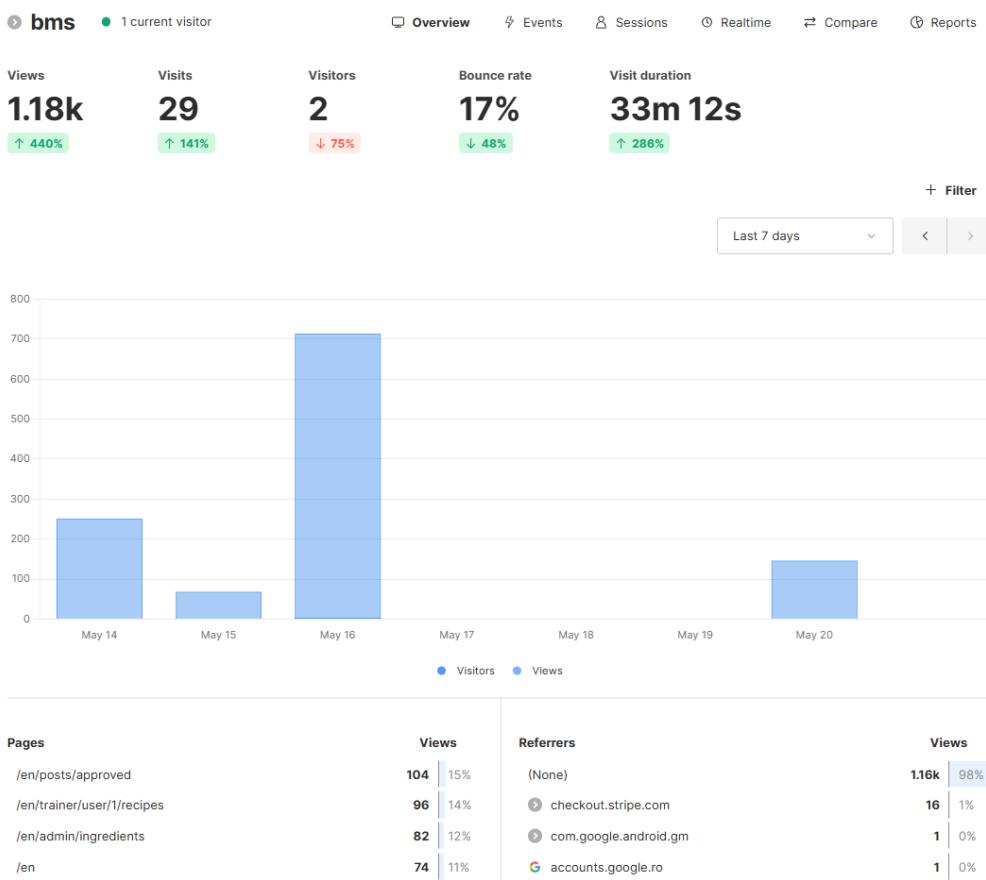


Figura Anexe-Completări 18. Statistici în Umami pentru front-end sumar general.

Denormalizare În Postgre

În cadrul implementării concrete, pentru a gestiona relațiile de tip *one-to-many* și *many-to-many*, am optat pentru o abordare denormalizată. Astfel, într-una dintre tabele, de regulă cea considerată dependentă într-un model normalizat, este păstrată o listă de ID-uri corespunzătoare entității „părinte”. În paralel, pentru a acoperi cazurile unde este necesară evidențierea a cărui părinte este în copil împreună cu multiplicitatea relației (1 în cazul *one-to-many* și ≥ 1 în cazul *many-to-many*), în microserviciul corespunzător este menținută și o tabelă „asociativă” suplimentară.

Această dublare intenționată a informației este justificată prin arhitectura specifică a aplicației, în care, de cele mai multe ori, interesele interogărilor sunt centrate pe identificarea rapidă a ID-urilor (folosite pentru a simula un „join distribuit” între microservicii sau pentru cereri viitoare din front-end), mai degrabă decât pe conținutul concret al listei. În plus, păstrarea listei în tabelele denormalizate aduce un beneficiu important în cazul relațiilor *many-to-many*: se conservă implicit ordinea elementelor, fără a fi necesară o sortare suplimentară.

Această strategie permite optimizarea interogărilor prin eliminarea unor *join*-uri adiționale, însă vine cu un cost crescut la nivel de memorie și implică necesitatea unui mecanism de consens între sursele de adevăr. Totodată, tabela asociativă rămâne esențială în scenariile unde conținutul exact al listei este relevant, cum ar fi, de exemplu, identificarea meselor în care apare o anumită rețetă.

În astfel de cazuri, utilizarea unui index de tip GIN (detaliat mai jos) nu este recomandată, întrucât cardinalitatea rețelilor din aplicație poate fi ridicată, ceea ce ar duce la o degradare a performanței indexului.

Pentru un exemplu de cod SQL pentru gestionarea consensului la actualizare și decrementarea multiplicității unui copil pentru un părinte se poate consulta [Extras Cod-Sursă Anexe-Completări 1](#).

```
// Cheia primara a tablei asociative este formata din (master_id, child_id).

/*
 * SQL care realizeaza schimbarea copiilor pentru un parinte.
 * Noii copii sunt in lista newChildIds.
 */
private Pair<String, String> buildConsensusSql(String table) {
    /*
     * Adaugarea in tabela asociativa cu incrementarea
     * multiplicitatii la conflict.
     * Noii copii sunt in lista newChildIds.
     */
    String upsertTpl = """
        INSERT INTO %1$s (master_id, child_id, multiplicity)
        SELECT :masterId, x, cnt
        FROM (
            SELECT x, COUNT(*) AS cnt
            FROM UNNEST(CAST(:newChildIds AS BIGINT[])) AS x
            GROUP BY x
        ) AS batch
        ON CONFLICT (master_id, child_id) DO UPDATE
        SET multiplicity = EXCLUDED.multiplicity
    """;

    /*
     * Stergerea din tabela asociativa a inregistrarilor
     * care nu mai sunt in lista de copii
     */
    String pruneTpl = """
        DELETE FROM %1$s
        WHERE master_id = :masterId
        AND child_id NOT IN (
            SELECT DISTINCT x
            FROM UNNEST(CAST(:newChildIds AS BIGINT[])) AS x
        )
    """;

    return Pair.of(
        String.format(upsertTpl, table),
        String.format(pruneTpl, table));
}
```

```

    }

/*
 * Decrementarea multiplicitatii copilului childId pentru masterId si stergerea
 * lui daca multiplicitatea devine 0.
 */
private String buildRemoveChildSql(String table) {
    String tpl = """
        WITH decremented AS (
            UPDATE %1$s
            SET multiplicity = multiplicity - 1
            WHERE master_id = :masterId
                AND child_id = :childId
                AND multiplicity > 1
            RETURNING 1
        )
        DELETE FROM %1$s
        WHERE master_id = :masterId
            AND child_id = :childId
            AND NOT EXISTS (SELECT 1 FROM decremented)
        """;
    return String.format(tpl, table);
}

```

Extras Cod-Sursă Anexe-Completări 1. Gestionarea tabelei asociative în cazul relațiilor one-to-many și many-to-many.

În *PostgreSQL*, un GIN (Generalized Inverted Index) este o un tip de index specializat pentru tipurile de date compuse și cu valori multiple, precum *array*-uri, *tsvector* și câmpuri *json*. Spre deosebire de indecșii tradiționali *B-tree*, care stochează câte o intrare pentru fiecare rând, un index GIN inversează această abordare prin păstrarea unei intrări pentru fiecare element individual din conținut, permitând astfel căutări inverse rapide pentru operații de tip incluziune și intersecție, cu costul unui consum mai mare de memorie.[\[14\]](#) Aceasta face GIN deosebit de potrivit pentru modele de date denormalizate, frecvent întâlnite în arhitecturile bazate pe microservicii, unde relațiile *one-to-many* sunt adesea stocate ca *array*-uri, pentru a minimiza operațiile artificiale de *join* între servicii.

Pentru *array*-uri, GIN asigură performanțe de citire ridicate împreună cu operatorii precum „@>”, „&&” și „= ANY()”. Acești operatori se potrivesc direct formatului invers al indexului, evitând scanările complete ale tabelei și îmbunătățind semnificativ viteza de filtrare. Totuși, ca limitare importantă, indecșii GIN devin mai puțin eficienți atunci când volumul de date indexate este foarte mare (i.e. cardinalitatea cheilor), atât din cauza costurilor ridicate de întreținere la scriere, cât și din perspectiva consumului de memorie și latenței crescute la actualizare.[\[37\]](#)

De exemplu, în cadrul aplicației, o postare poate avea mai multe *tag*-uri, care au o cardinalitate totală mică. Pentru a permite căutarea eficientă a postărilor pe baza acestor

etichete, am optat pentru utilizarea unui index GIN aplicat pe un *array* de ID-uri de *tag-uri*, în locul unei tabele de nomenclator însotite de un *join*. Un exemplu este ilustrat în [Extras Cod-Sursă Anexe-Completări 2](#):

```

CREATE TABLE IF NOT EXISTS Post (
    id          SERIAL PRIMARY KEY,
    approved    BOOLEAN NOT NULL DEFAULT FALSE,
    body        TEXT    NOT NULL,
    title       TEXT    NOT NULL,
    user_likes  BIGINT[]      DEFAULT '{}',
    user_dislikes BIGINT[]    DEFAULT '{}',
    user_id     BIGINT NOT NULL,
    tags        TEXT[]       DEFAULT '{}',
    created_at  TIMESTAMP     DEFAULT CURRENT_TIMESTAMP,
    updated_at  TIMESTAMP     DEFAULT CURRENT_TIMESTAMP,
    images      TEXT[]       DEFAULT '{}'
);
-- Creare index GIN pentru array
CREATE INDEX IF NOT EXISTS idx_post_tags ON post USING gin (tags);

```

Extras Cod-Sursă Anexe-Completări 2. Crearea unui index GIN pentru un array.

Pentru index-ul *idx_post_tags* s-ar păstra în memorie o structură conceptuală de tipul:

Tag (din coloana <i>tags</i>)	Post IDs
#wellness	{3, 4, 5, 6}
#mentalhealth	{1, 3, 4, 10, 11}

Planul asociat cererii care filtrează tabelul *Post* după tag-uri folosind un index GIN se poate observa în [Figura Anexe-Completări 19](#) :



Figura Anexe-Completări 19. Plan de execuție pentru o cerere care utilizează index GIN.

Eficiență Produsului Scalar

Pentru vectorii aproape normalizați, cum sunt cei generați de modelele de embedding semantic, utilizarea produsului scalar pentru măsurarea similarității este mai eficientă atât computațional, cât și din punct de vedere al acurateței semantice.

În acest caz, cosinusul unghiului dintre doi vectori, o măsură standard de similaritate, se reduce la simplul produs scalar al acestora:

$$\cos(\theta) = (\vec{u} \cdot \vec{v}) / (\|\vec{u}\| \cdot \|\vec{v}\|) \text{ atunci când } \|\vec{u}\| \approx 1, \|\vec{v}\| \approx 1, \text{ deci } \cos(\theta) \approx \vec{u} \cdot \vec{v}$$

Produsul scalar este mai eficient de calculat decât distanțele *Euclidiene* sau alte metrici, deoarece presupune doar înmulțiri și adunări, fără operații suplimentare de normalizare sau rădăcini pătrate. De asemenea, indexul *HNSWM* din *PgVector* este optimizat pentru produsul scalar, mai ales când vectorii sunt aproape normalizați. [\[30\]](#)

Limitări Plăci Video Vechi

Plăcile grafice noi, în special cele dedicate *AI* (de exemplu, *NVIDIA A100*, *H100* sau *RTX 30xx*, *40xx*, *50xx*), dispun de optimizări hardware avansate pentru rularea modelelor *LLM*, cum ar fi „tensor cores”, memorie *VRAM* extinsă și suport pentru formate numerice eficiente (ex. *FP16*, *INT8*). Aceste optimizări permit procesarea mai rapidă și mai eficientă a contextului lung, reducând consumul de resurse.

În schimb, plăcile mai vechi (de exemplu, seria *GTX 9xx*, *10xx* sau *RTX 20xx*) nu beneficiază de aceste îmbunătățiri, având *VRAM* limitată și lipsind suportul complet pentru accelerarea specifică *AI*, ceea ce face dificilă rularea eficientă a *LLM*-urilor, mai ales când contextul este extins. [\[28\]](#)

Figuri

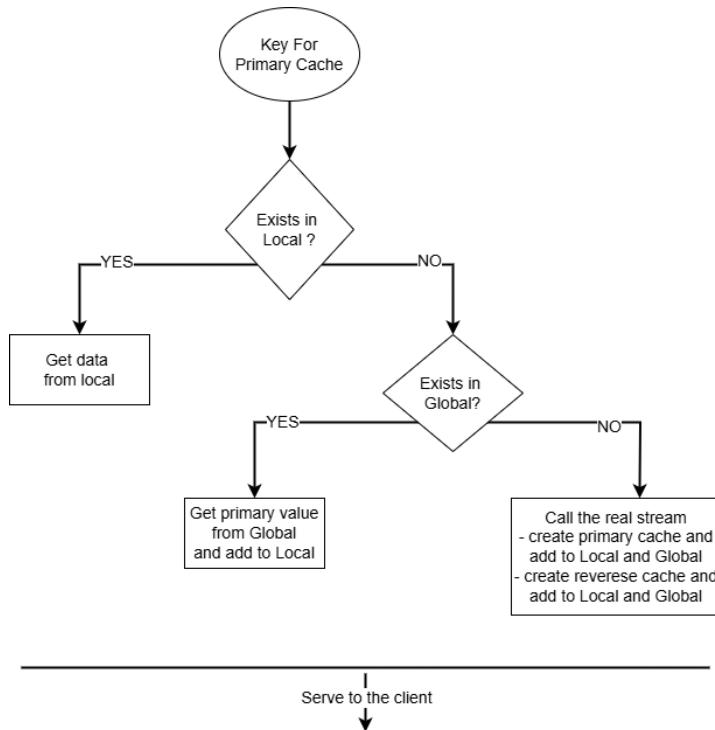


Figura Anexe-Figuri 1. Diagrama de flux privind accesarea cache-ului în back-end.

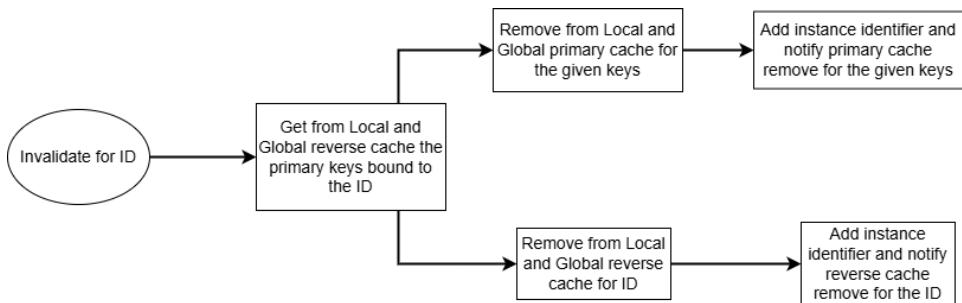


Figura Anexe-Figuri 2. Diagrama de flux privind invalidarea cache-ului în back-end.

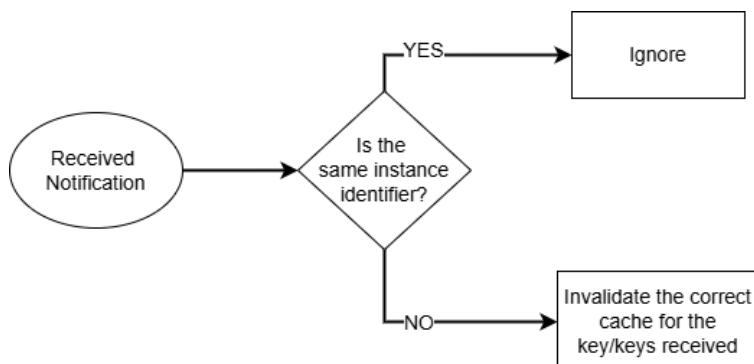


Figura Anexe-Figuri 3. Diagrama de flux privind trimiterea de notificări pentru invalidarea cache-ului în back-end.

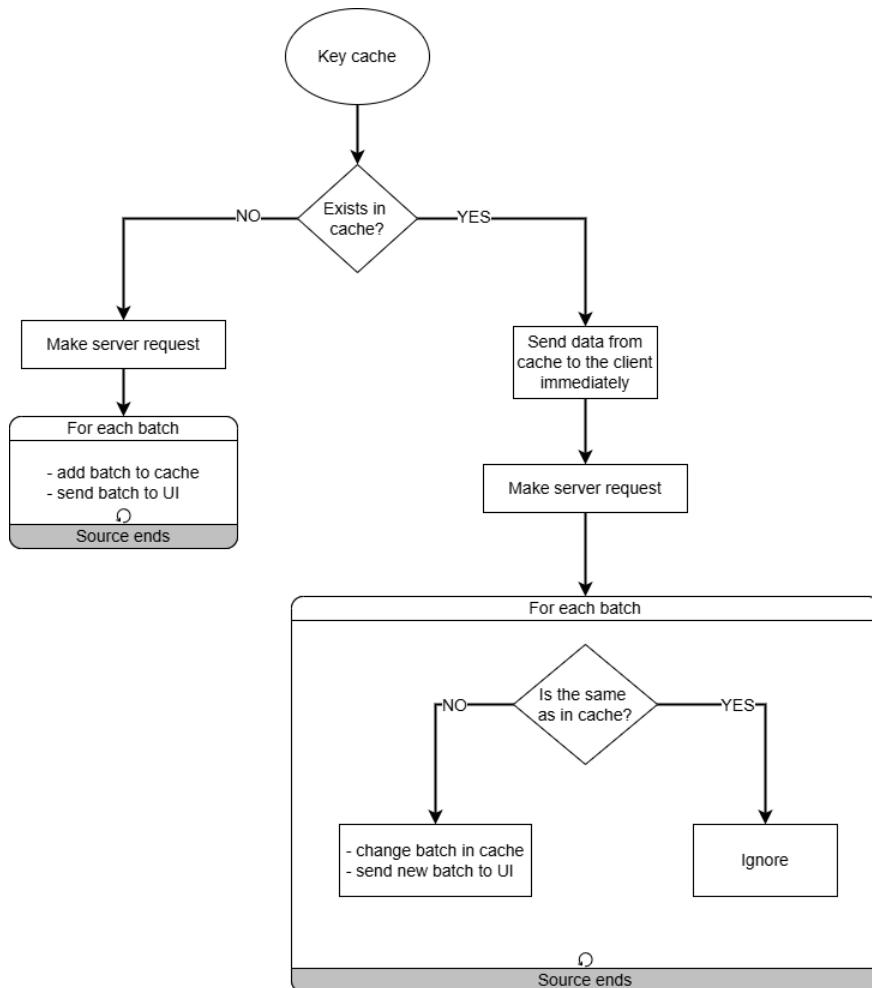


Figura Anexe-Figuri 4. Diagrama de flux privind mecanismul de caching în front-end.

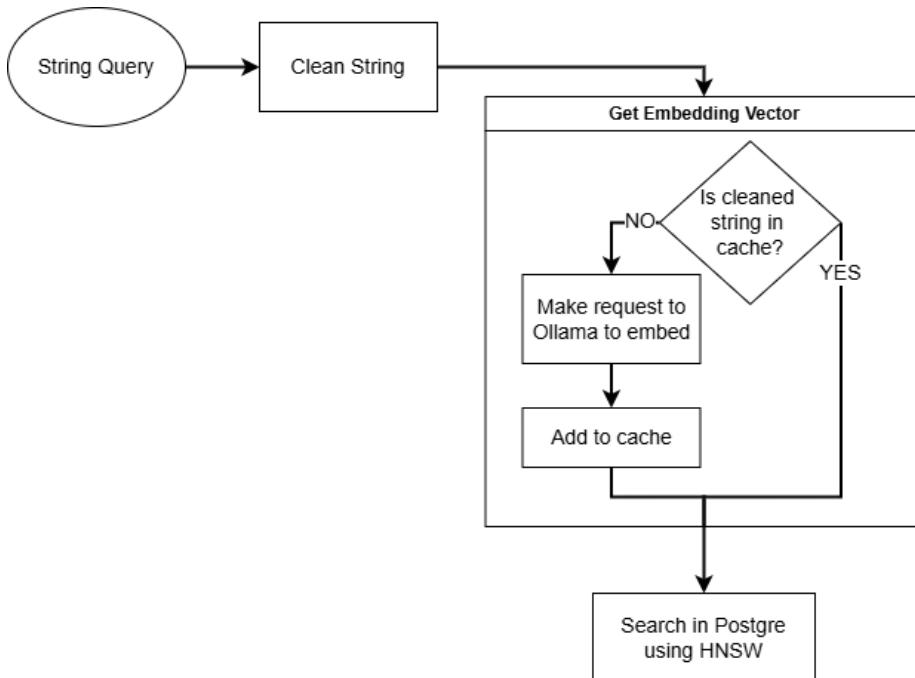


Figura Anexe-Figuri 5. Diagrama de flux privind mecanismul de căutare semantică în aplicație.

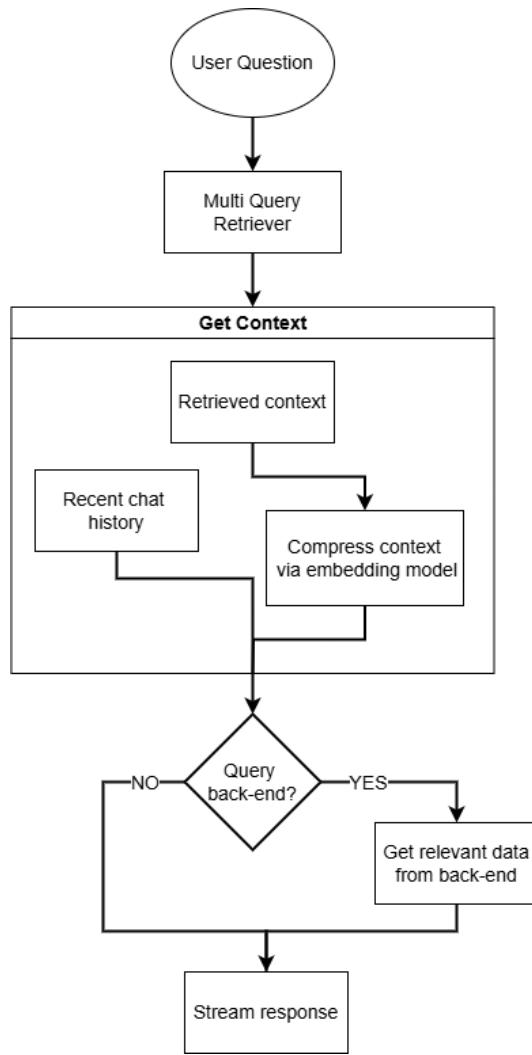


Figura Anexe-Figuri 6. Diagrama de interacție a utilizatorului cu chat bot-ul în aplicație.

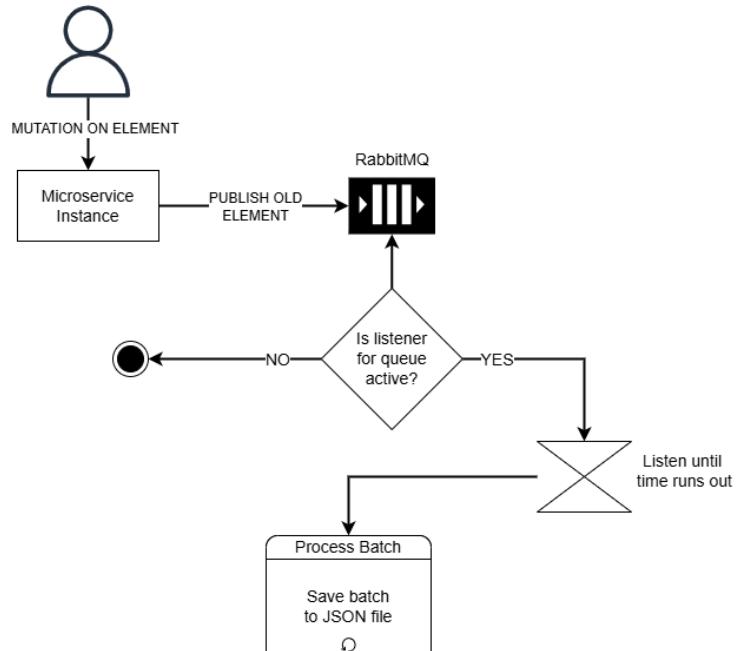


Figura Anexe-Figuri 7. Diagrama de flux privind mecanismul de arhivare în aplicație.

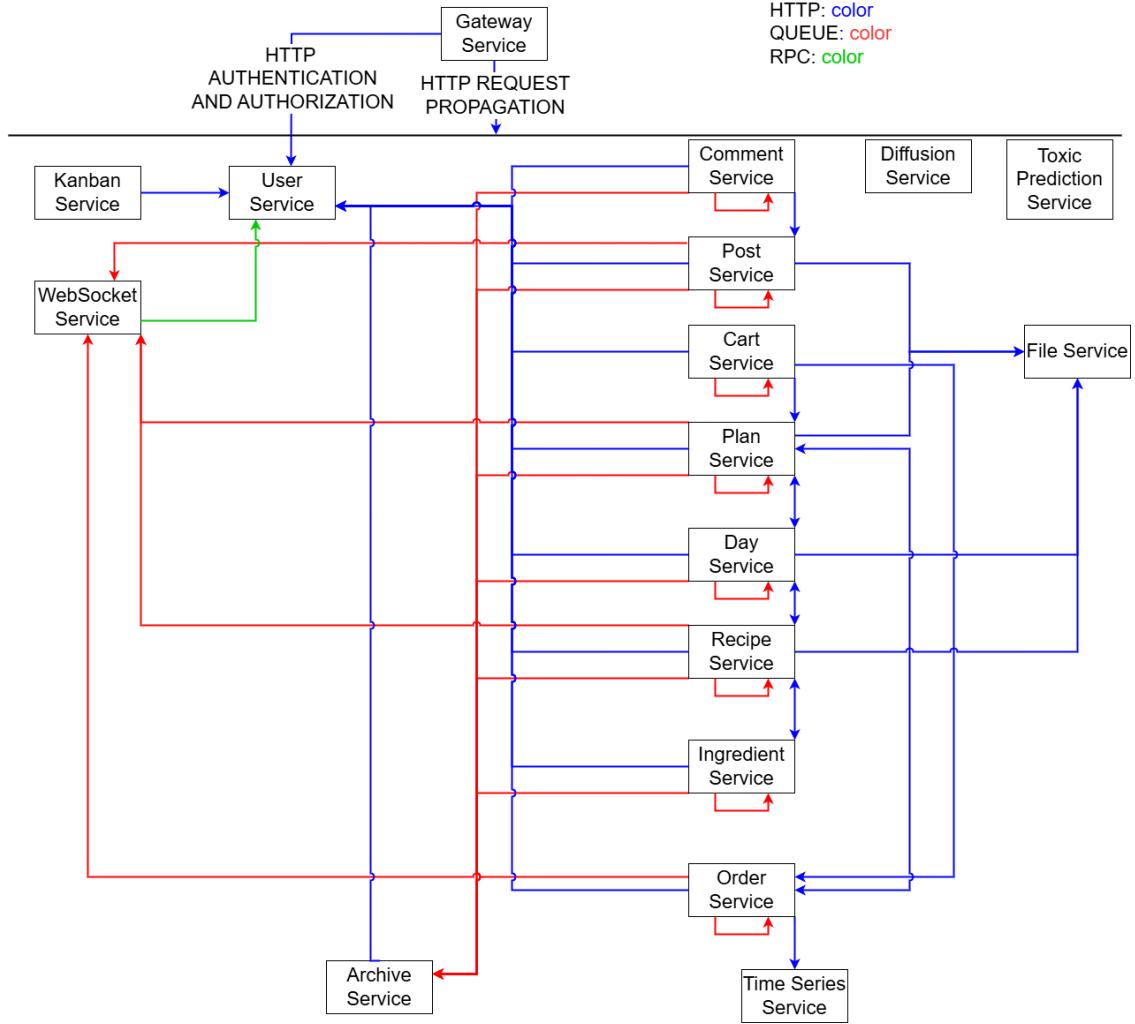


Figura Anexe-Figuri 8. Diagrama generală de comunicare între microserviciile din back-end.

Element	Class, %	Method, %	Line, %	Branch, %
com.mocicarazvan.templatemodule	77% (81/104)	79% (390/489)	80% (1257/1554)	83% (240/286)
TemplateModuleApplication	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
utils	92% (12/13)	95% (95/100)	96% (300/312)	86% (112/130)
services	92% (13/14)	90% (148/164)	92% (404/439)	85% (58/68)
repositories	100% (2/2)	100% (12/12)	100% (51/51)	100% (4/4)
models	83% (5/6)	62% (5/8)	55% (10/18)	100% (0/0)
mappers	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
jackson	0% (0/4)	0% (0/10)	0% (0/28)	0% (0/6)
hateos	33% (3/9)	28% (14/49)	26% (46/172)	100% (6/6)
exceptions	88% (15/17)	93% (15/16)	96% (27/28)	100% (0/0)
enums	100% (4/4)	100% (8/8)	100% (9/9)	100% (0/0)
email	25% (1/4)	69% (9/13)	70% (33/47)	83% (5/6)
dtos	88% (8/9)	81% (13/16)	85% (57/67)	25% (1/4)
dbCallbacks	100% (3/3)	100% (4/4)	100% (19/19)	100% (10/10)
crypt	100% (1/1)	100% (2/2)	100% (26/26)	100% (0/0)
convertors	100% (2/2)	100% (4/4)	100% (6/6)	100% (4/4)
controllers	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
clients	90% (10/11)	72% (48/66)	80% (242/300)	83% (40/48)
advices	100% (2/2)	81% (13/16)	87% (27/31)	100% (0/0)

Figura Anexe-Figuri 9. Rezultatele acoperirii codului de către teste în modulul de bază.

Element	Class, %	Method, %	Line, %	Branch, %
com.mocicarazvan.rediscache	92% (24/26)	86% (176/203)	87% (618/704)	80% (153/191)
RedisCacheApplication	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
utils	100% (5/5)	97% (34/35)	86% (106/122)	79% (59/74)
services	100% (3/3)	90% (19/21)	96% (74/77)	90% (9/10)
local	100% (5/5)	75% (43/57)	77% (103/133)	82% (23/28)
enums	100% (1/1)	100% (7/7)	96% (25/26)	94% (18/19)
dtos	100% (3/3)	100% (5/5)	100% (8/8)	100% (0/0)
config	50% (1/2)	75% (6/8)	76% (19/25)	0% (0/4)
aspects	100% (6/6)	88% (62/70)	90% (283/313)	78% (44/56)
annotation	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)

Figura Anexe-Figuri 10. Rezultatele acoperirii codului de către teste în modulul de caching.

Element	Class, %	Method, %	Line, %	Branch, %
com.mocicarazvan.ollamasearch	81% (13/16)	60% (53/88)	66% (140/211)	60% (34/56)
OllamaSearchApplication	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
utils	100% (2/2)	100% (12/12)	100% (33/33)	100% (14/14)
services	100% (2/2)	100% (18/18)	100% (51/51)	100% (12/12)
repositories	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
models	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
exceptions	100% (1/1)	50% (1/2)	50% (1/2)	100% (0/0)
dtos	83% (5/6)	32% (16/49)	27% (26/95)	8% (2/24)
dbCallbacks	100% (1/1)	100% (1/1)	100% (5/5)	100% (2/2)
config	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
clients	100% (1/1)	100% (4/4)	100% (23/23)	100% (4/4)
cache	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)

Figura Anexe-Figuri 11. Rezultatele acoperirii codului de către teste în modulul de căutare semantică.

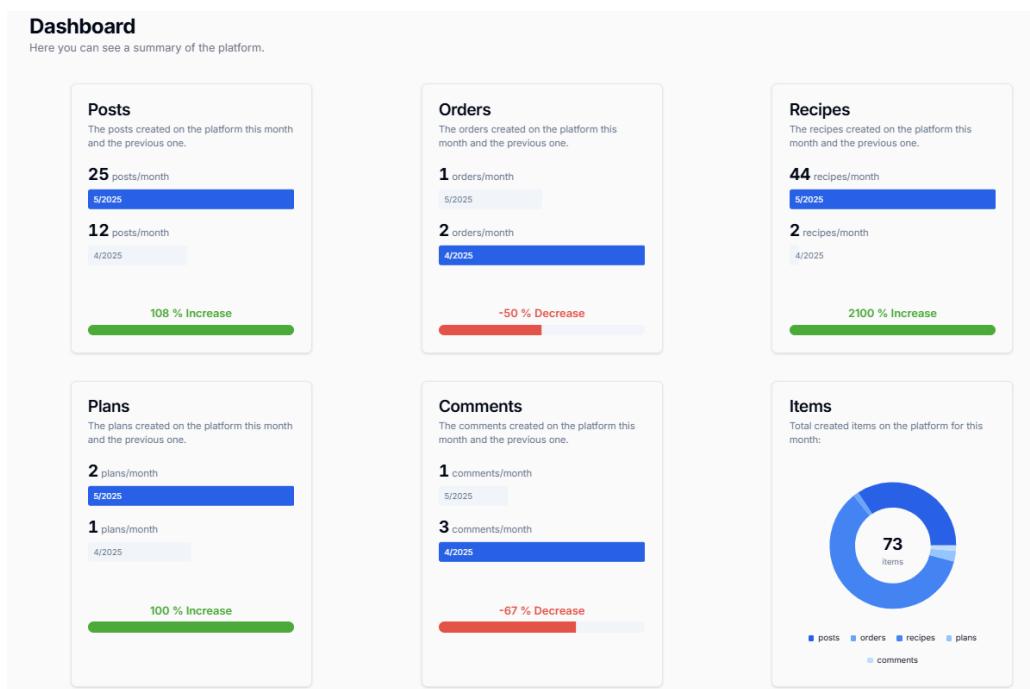


Figura Anexe-Figuri 12. Dashboard sumar pentru administrator.

Top Users

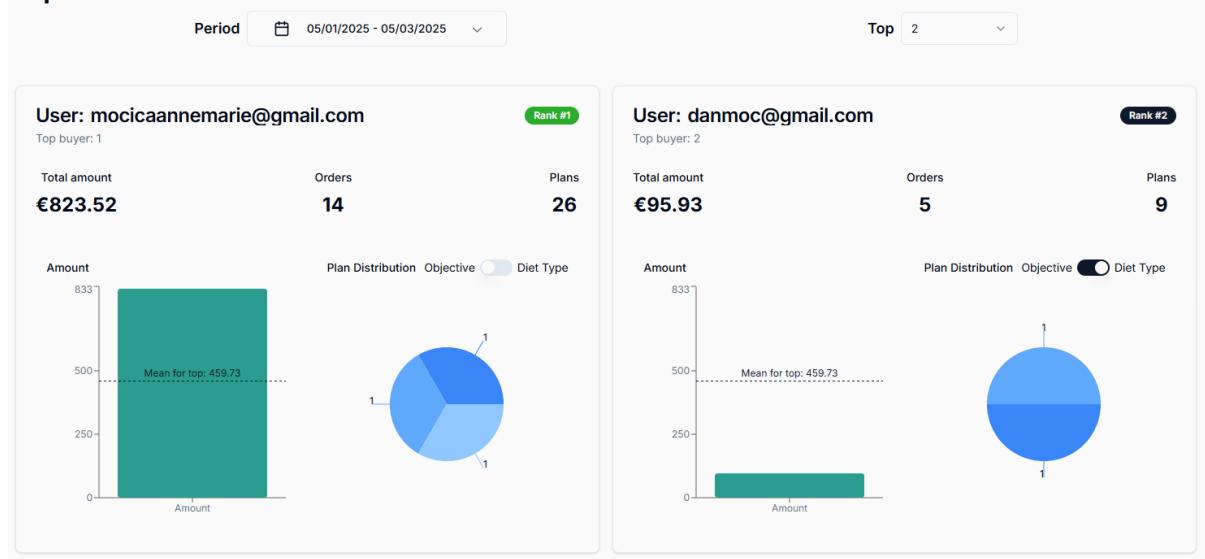


Figura Anexe-Figuri 13. Top utilizatori pentru administrator.

Top Trainers

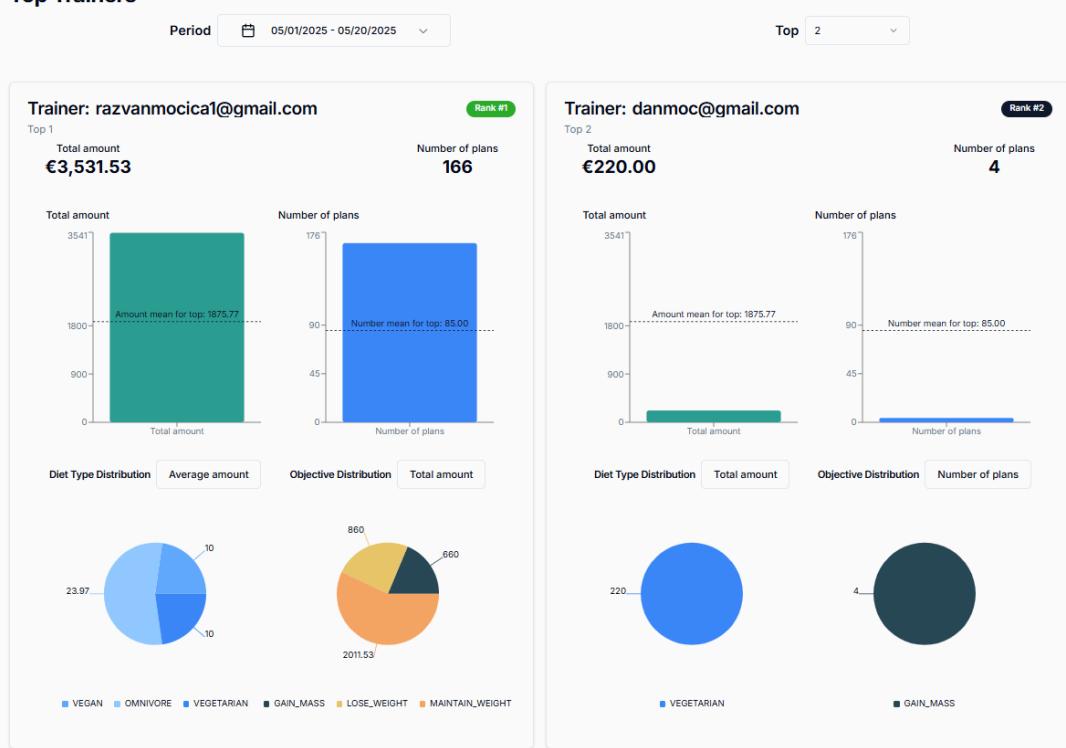


Figura Anexe-Figuri 14. Top utilizatori cu drepturi de postare pentru administrator.

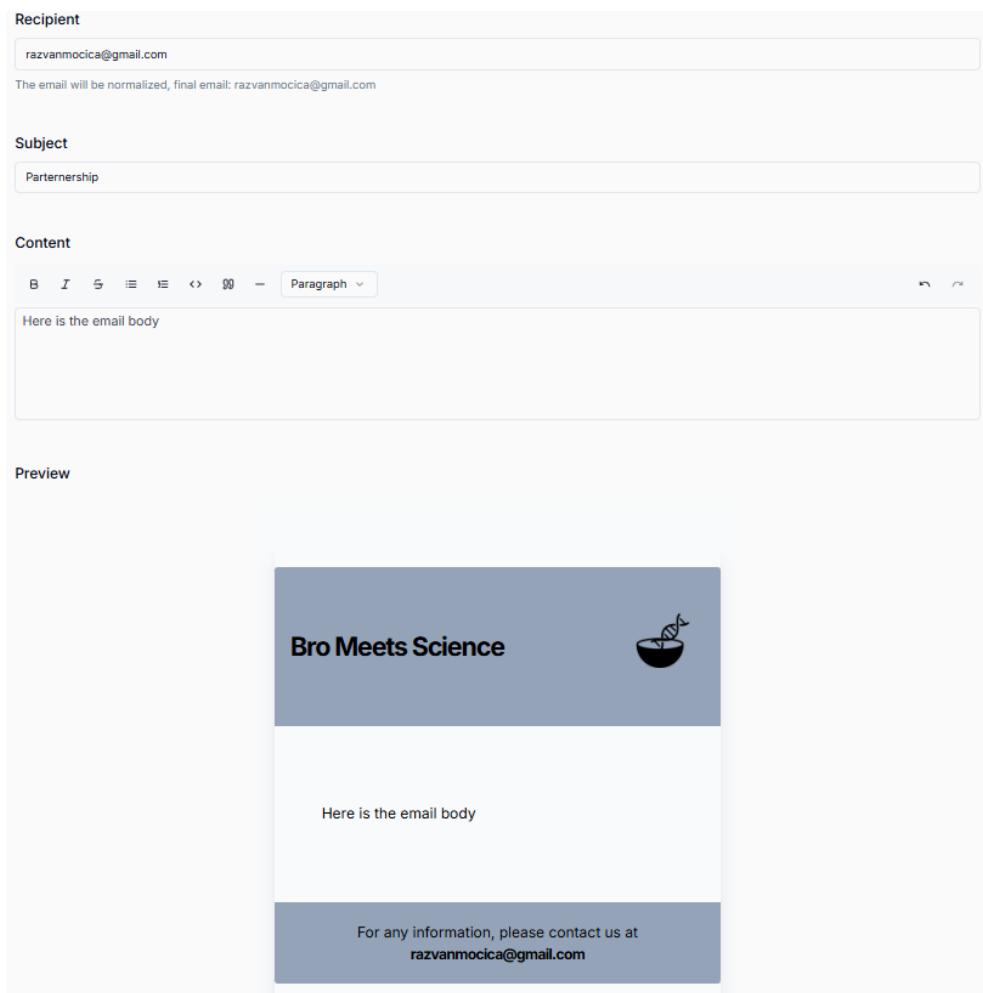


Figura Anexe-Figuri 15. Creare e-mail din partea companiei.

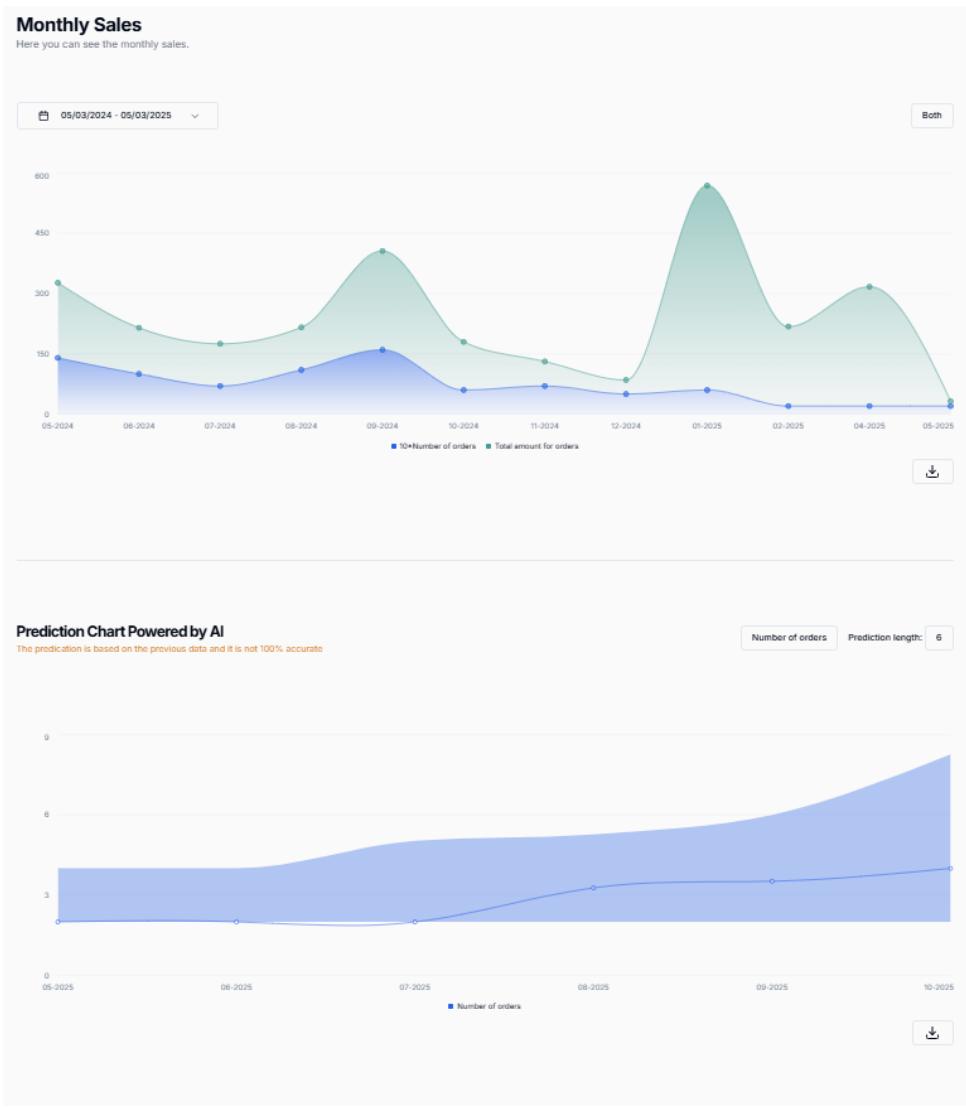


Figura Anexe-Figuri 16. Vânzări lunare pentru administrator-partea I.

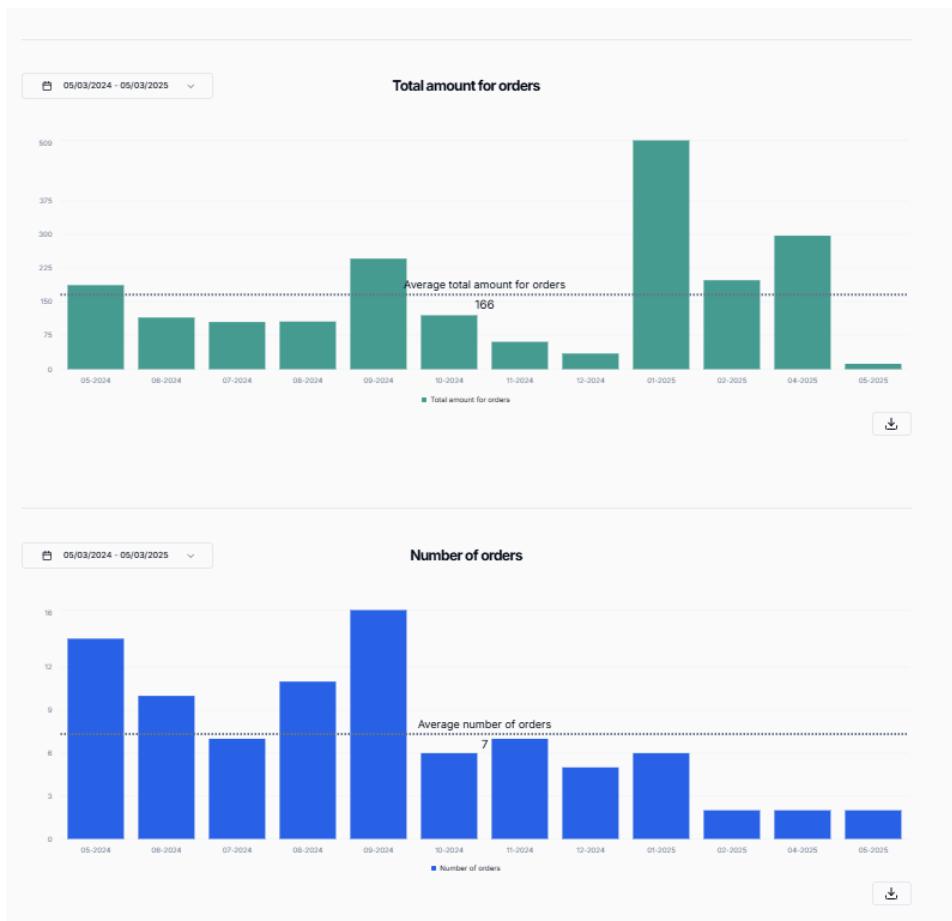


Figura Anexe-Figuri 17. Vânzări lunare pentru administrator-partea a II-a.



Figura Anexe-Figuri 18. Vânzări lunare pentru administrator-partea a III-a.

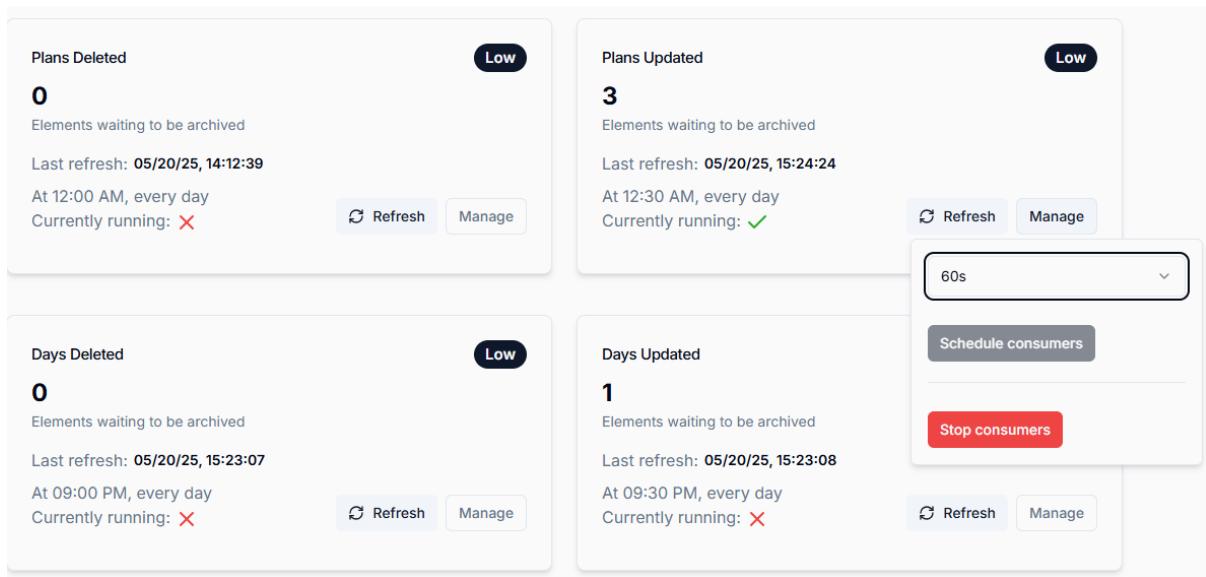


Figura Anexe-Figuri 19. Arhivare planuri pentru administrator.

Create A New Recipe

Title
Chunky Turkey Chili: A Hearty and Nutritious Meal for Any Time of Day AI

Body
AI
B I S H M D C P Paragraph

Ingredients Add new line

Select An Ingredient	Quantity
Turkey, breast, lean flesh, raw	200

Add Ingredient Remove Ingredient Clear the inputs

Select An Ingredient	Quantity
Garlic, peeled, fresh, raw	50

Add Ingredient Remove Ingredient Clear the inputs

Figura Anexe-Figuri 20. Formular de creare rețetă-partea I.

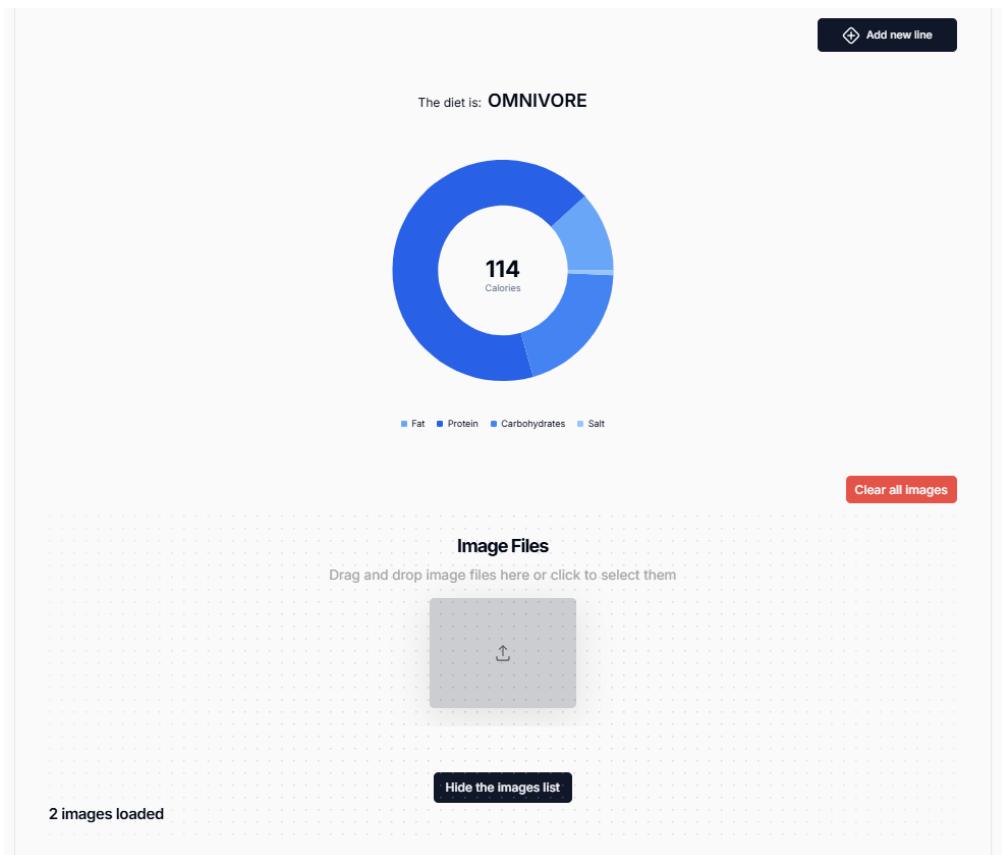


Figura Anexe-Figuri 21. Formular de creare rețetă-partea a II-a.

2 images loaded

Header

Generate images with AI

Clear all videos

Video Files
Drag and drop video files here or click to select them

One video loaded

Show the videos list

Submit

Figura Anexe-Figuri 22. Formular de creare rețetă-partea a III-a.

Your Posts
Here you can see and manage all the posts you have created.

Search in title...
Select tags to filter
Columns
Download selected

<input checked="" type="checkbox"/>	ID	Title ↑	#of likes	#of dislikes	Creation date ↓	Last update ↑	Approved	
<input checked="" type="checkbox"/>	146	Enhanc...	0	0	03/05/2025	03/05/2025	Yes	...
<input checked="" type="checkbox"/>	123	How Fi...	0	0	30/04/2025	30/04/2025	Yes	...
<input checked="" type="checkbox"/>	122	Elevate...	0	0	29/04/2025	29/04/2025	Yes	...
<input type="checkbox"/>	121	Explori...	0	0	27/04/2025	27/04/2025	Yes	...
<input type="checkbox"/>	120	Boost Y...	0	0	27/04/2025	27/04/2025	Yes	...
<input type="checkbox"/>	119	The Es...	0	0	27/04/2025	27/04/2025	Yes	...
<input type="checkbox"/>	118	The Mi...	0	0	27/04/2025	27/04/2025	Yes	...
<input type="checkbox"/>	117	Fish-Fo...	0	0	27/04/2025	27/04/2025	Yes	...
<input type="checkbox"/>	115	Nourish...	0	0	27/04/2025	27/04/2025	Yes	...
<input type="checkbox"/>	114	Boostin...	0	0	27/04/2025	27/04/2025	Yes	...

3 of 47 row(s) selected

Page size 10
Page 1 of 5
<< < > >>

Figura Anexe-Figuri 23. Pagina pentru administrarea postărilor-partea I.

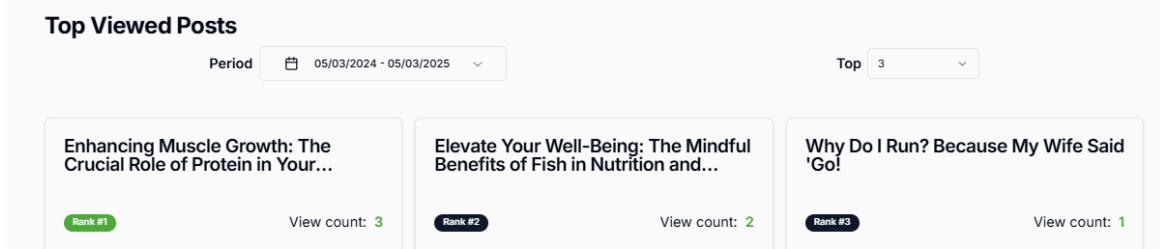
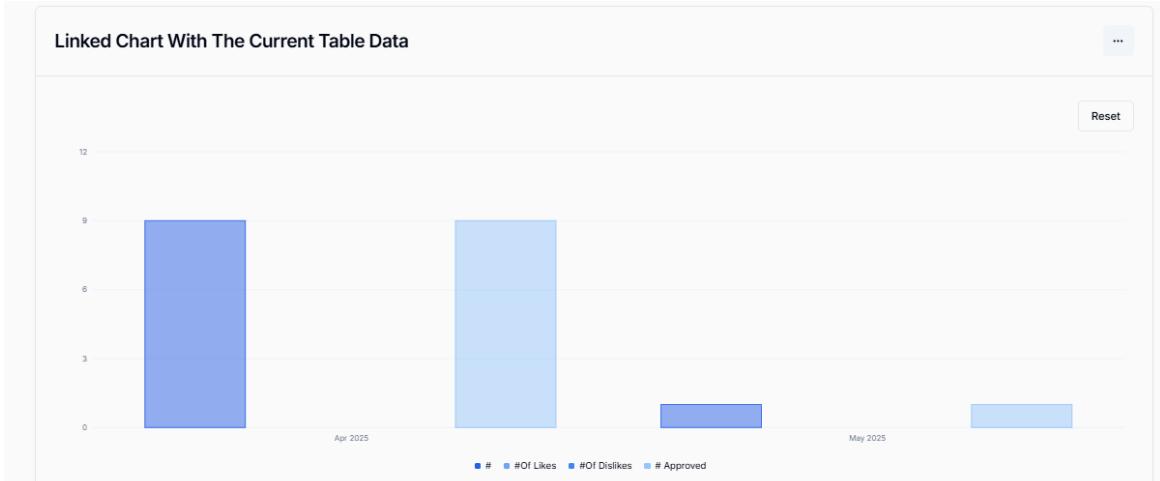


Figura Anexe-Figuri 24. Pagina pentru administrarea postărilor-a II-a.

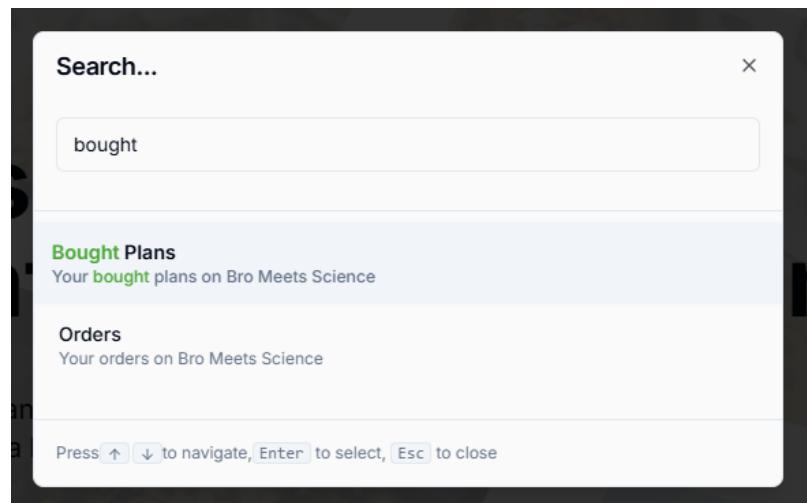


Figura Anexe-Figuri 25. Căutare în site.

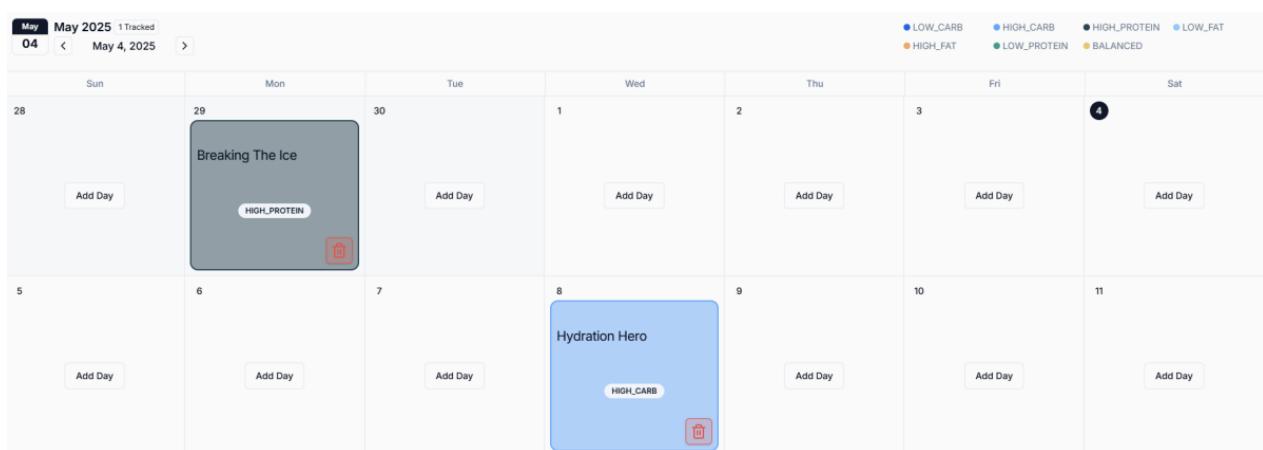


Figura Anexe-Figuri 26. Calendarul zilelor urmărite-partea I.



Figura Anexe-Figuri 27. Calendarul zilelor urmărite-partea a II-a.

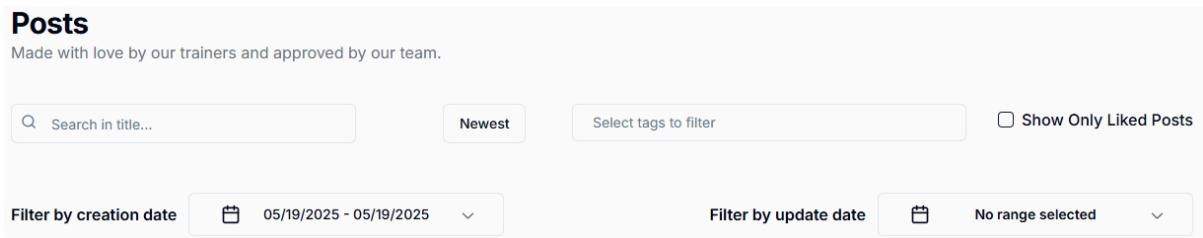


Figura Anexe-Figuri 28. Exemplu internaționalizare site limba engleză.

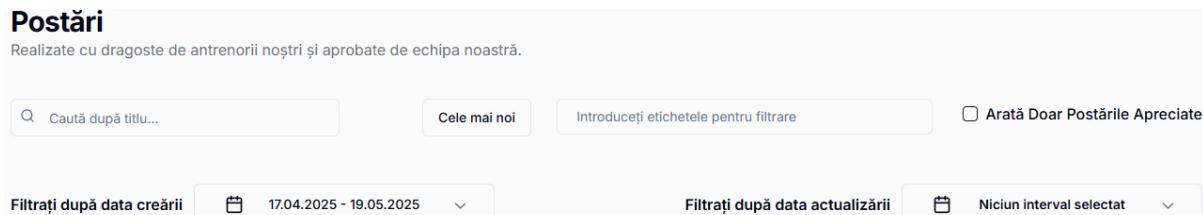


Figura Anexe-Figuri 29. Exemplu internaționalizare site limba română.

Secvențe De Cod

```
protected Flux<Object> methodFluxResponseToCache(ProceedingJoinPoint joinPoint, String key,
                                                String savingKey, String idPath, boolean saveToCache) {
    try {
        // fluxul original al metodei
        Flux<Object> original = (Flux<Object>) joinPoint.proceed(joinPoint.getArgs());
        /* Fluxul cache-uit pentru a-l putea consuma de mai multe ori.
         * Există și o limită pentru a preveni un flux căld/infiniț.
         */
        Flux<Object> cached = original.cache(Duration.ofSeconds(maxCacheFluxSeconds));
        // consumarea fluxului cache-uit de către client, pastrand transparentă
        return cached
            .doOnComplete(() -> {
                if (saveToCache) {
                    // salvarea fluxului în cache, în background pe un thread virtual
                    asyncTaskExecutor
                        .submit(() -> saveFluxResultToCache(joinPoint, key,
                                                               savingKey, idPath, cached));
                }
            });
    } catch (Throwable e) {
        return Flux.error(e);
    }
}

protected void saveFluxResultToCache(ProceedingJoinPoint joinPoint, String key, String
                                    savingKey, String idPath, Flux<Object> original) {
    /*
     * Mutarea procesării fluxului pe un thread 'normal' în acest caz ForkJoinPool
     * deoarece se poate ajunge la un necesar computational destul de mare.
     * Folosind ForkJoinPool se poate asigura o concurență mai mare în cazuri
     * în care majoritatea procesării este însă relativ mică din punct de vedere
     * computational.
     */
    Scheduler fallbackScheduler = scheduler != null ? scheduler : Schedulers.immediate();
    original
        .publishOn(fallbackScheduler)
        /*
         * Crearea unui publisher care va emite id-urile fluxului pentru
         * invalidările viitoare
         * și a listei care va fi salvată în cache
         */
        .reduce(Tuples.of(Sinks.many().unicast().<Long>onBackpressureBuffer(), new
                        ArrayList<>()), (acc, cr) -> {
            long id = aspectUtils.assertLong(
                aspectUtils.evaluateSpelExpressionForObject(idPath, cr,
                joinPoint));
            acc.getT1().tryEmitNext(id);
            acc.getT2().add(cr);
            return acc;
        })
        .doOnSuccess(t -> {
            // completarea publisher-ului de id-uri
            t.getT1().tryEmitComplete();
        })
        .flatMapMany(tuple -> {
            Flux<Long> ids =
                tuple.getT1().asFlux().timeout(Duration.ofSeconds(maxCacheFluxSeconds));
        })
}
```

```

        List<Object> values = tuple.getT2();
        return reactiveRedisTemplate.opsForValue().set(savingKey, values,
Duration.ofMinutes(expireMinutes))
            // salvarea in cache-urile locale doar daca in redis s-a salvat cu succes
            .filter(Boolean::booleanValue)
            /*
             * Nu este nevoie de un pipeline pentru a eficientiza
             * deoarece libraria reactiva face automat unul
             */
            .flatMapMany(_ -> ids
                // adaugarea in cache-ul de invalidare
                .flatMap(id -> addToReverseIndex(key, id, savingKey)))
            .doOnComplete(() -> localReactiveCache.put(savingKey, values));
    }).subscribe(
        success -> {
            return;
        },
        error -> log.error("Failed to set key: {}", savingKey, error));
}

```

Extras Cod-Sursă 1. Salvarea în cache-urile din back-end pentru un publisher cu mai mult de un element.

```

// factorul pentru TTL, putin peste o secunda
const DEDUPLICATION_FACTOR = 1.05 as const;

// cache global care tine fie o functie ce produce AsyncGenerator,
// fie un Promise care se va rezolva cu acea functie
const globalMemoizedIterators = new TTLCache<
    string,
    ((() => AsyncGenerator<any>) | Promise<() => AsyncGenerator<any>>
>({
    ttl: DEDUPLICATION_FACTOR * 1000,
    updateAgeOnGet: true,
});

// functia care memoreaza iteratorii asincroni pentru a nu fi recreati concurrent
async function memoizeAsyncIterator<T>(
    key: string,
    asyncIteratorFn: () => AsyncIterable<T>,
): Promise<() => AsyncGenerator<T>> {
    // verificam daca exista deja ceva in cache pentru acest key
    const existingIterator = globalMemoizedIterators.get(key);
    if (existingIterator) {
        // daca este deja o functie gata creata, o returnam imediat
        if (typeof existingIterator === "function") {
            return existingIterator;
        } else {
            // daca este inca un Promise, asteptam pana se rezolva si intoarcem functia
            // caz de race condition
            return await existingIterator;
        }
    }
}

```

```

}

// initializam cu functii no-op ca sa nu avem eroare TS
let resolveFactory: (genFn: () => AsyncGenerator<T>) => void = (_)> {};
let rejectFactory: (err: any) => void = (_)> {};

// cream un Promise pentru a evita race condition: orice apel concurrent
// va gasi acest Promise si va face await pe el, in loc sa reconstruiasca iteratorul
const factoryPromise = new Promise<() => AsyncGenerator<T>>(
  (resolve, reject) => {
    resolveFactory = resolve;
    rejectFactory = reject;
  },
);
globalMemoizedIterators.set(key, factoryPromise);

// pornim un IIFE asincron care va construi in background iteratorul partajat
(async () => {
  try {
    // array pentru valorile primite din sursa
    const cache: T[] = [];
    // colectam rezolvatorii de Promise pentru generare valorilor in timp real
    const resolvers: ((value: T | undefined) => void)[] = [];
    // apelam functia originala care intoarce AsyncIterable
    const iterator = asyncIteratorFn();
    // flag pentru oprire
    let aborted = false;

    // definim un generator care mai intai extrage ce e in cache,
    // apoi asteapta valori noi sau semnal de terminare
    const sharedIterator = async function* (): AsyncGenerator<T> {
      let index = 0;
      while (index < cache.length) {
        yield cache[index++];
      }
      while (!aborted) {
        if (index < cache.length) {
          yield cache[index++];
        } else {
          const promise = new Promise<T | undefined>((resolve) =>
            resolvers.push(resolve),
          );
          const value = await promise;
          if (value === undefined) break;
          yield value;
          index++;
        }
      }
    };
  }
});

```

```

// construim o functie care returneaza un nou generator din sharedIterator
// si care are o metoda abort pentru a opri tot si a elibera
const generatorWithAbort = () => {
  const generator = sharedIterator();
  return Object.assign(generator, {
    abort: () => {
      aborted = true;
      if ("abort" in iterator && typeof iterator.abort === "function") {
        iterator.abort();
      }
      while (resolvers.length > 0) {
        resolvers.shift()?.(undefined);
      }
    },
  });
};

// IIFE care citeste in background din iterator si pune valorile in cache
(async () => {
  for await (const value of iterator) {
    cache.push(value);
    while (resolvers.length > 0) {
      resolvers.shift()?.(value);
    }
  }
  while (resolvers.length > 0) {
    resolvers.shift()?.(undefined);
  }
})();

// dupa ce functia generatorWithAbort e definita, o salvam in cache
globalMemoizedIterators.set(key, generatorWithAbort);
// apoi rezolvam Promise-ul initial cu aceasta functie
resolveFactory(generatorWithAbort);

} catch (err) {
  // in caz de eroare, stergem intrarea din cache si respingem Promise-ul
  globalMemoizedIterators.delete(key);
  rejectFactory(err);
}
})();

// intoarcem Promise-ul; apelul resultant va astepta pana se va rezolva
return factoryPromise;
}

// functia publica ce se apeleaza pentru a obtine un AsyncGenerator deduplicat
export const deduplicateFetchStream = async <

```

```

T,
E extends BaseError = BaseError,
>(
  params: FetchStreamBatchedProps<T> & {
    dedupKey: string;
  },
) => {
  // definim modul de obtinere al AsyncIterable-ului, cu callback pentru abort si eroare
  const asyncIteratorFn = () =>
    fetchStreamAsyncGenerator<T, E>({
      ...params,
      onAbort: () => {
        params.onAbort?.();
      },
      errorCallback: (err) => {
        params.errorCallback?.(err);
        // stergem cache-ul daca apare o eroare ca sa se poata reface cererea
        globalMemoizedIterators.delete(params.dedupKey);
      },
    });
  // obtinem factory-ul memorat si apoi il apelam pentru a primi AsyncGenerator-ul
  return (await memoizeAsyncIterator(params.dedupKey, asyncIteratorFn))();
};

```

Extras Cod-Sursă 2. Deduplicare de request-uri în front-end.

Observație pentru secvența de Extras Cod-Sursă 2: În JavaScript, elementele, fie ele anonime sau nu, sunt păstrate în memorie atât timp cât există referințe către ele prin *closure*. Când *memoizeAsyncIterator* creează un nou iterator, *array*-ul de cache rămâne activ deoarece este accesat de funcțiile interne (*sharedIterator*, *IIFE*, *generatorWithAbort*). Atât timp cât iteratorul există în *globalMemoizedIterators* (sau este folosit), întreaga închidere rămâne în memorie. După ce expiră cheia din *TTL Cache* și iteratorul nu mai este folosit, contextul este eliberat automat. Acest model de caching bazat pe *closure* este eficient, pentru că fiecare *dedupKey* are un context propriu, care se curăță singur când nu mai este nevoie de el.

```

public class ContainerScheduler {

  // containerul listener asociat cu o coada RabbitMQ
  private final SimpleMessageListenerContainer simpleMessageListenerContainer;
  // scheduler asincron pentru executia sarcinilor
  private final SimpleAsyncTaskScheduler simpleAsyncTaskScheduler;
  // configuratii legate de cozi, inclusiv cron-ul
  private final QueuesPropertiesConfig queuesPropertiesConfig;
  // componenta care notifica pornirea/opririle containerului
  private final ContainerNotify containerNotify;
  // referinta catre task-ul programat care opreste containerul
  private AtomicReference<ScheduledFuture<?>> stopTask = new AtomicReference<>();
  // numar maxim de incercari pentru a verifica daca containerul s-a oprit
  private int maxAttempts = 150;
}

```

```

// intervalul intre incercari, in milisecunde
private int intervalMillis = 500;
// programeaza pornirea containerului conform expresiei cron din configuratie
public void scheduleContainer() {
    simpleAsyncTaskScheduler.schedule(
        this::scheduleCron,
        new
        CronTrigger(queuesPropertiesConfig.getQueueJob(simpleMessageListenerContainer.getQueue
Names
        ()[0])));
}
// porneste containerul conform cron-ului si programeaza oprirea automata dupa
// un timp
private void scheduleCron() {
    String queueName = simpleMessageListenerContainer.getQueueNames()[0];
    stopPrevious(); // opreste orice task anterior
    stopContainerGracefully(); // inchide containerul daca este deja pornit
    log.info("Starting container for queue: {}", queueName);
    simpleMessageListenerContainer.start();
    containerNotify.notifyContainersStartCron(queueName); // notifica pornirea
    // (mod cron)
    // programeaza oprirea containerului dupa perioada specificata
    stopTask.set(simpleAsyncTaskScheduler.schedule(
        this::stopContainerGracefully,
        new AfterMillisTrigger(queuesPropertiesConfig.getSchedulerAliveMillis())));
}

log.info("Container will be stopped after {} s",
    Objects.requireNonNull(stopTask.get()).getDelay(TimeUnit.SECONDS));
}
// porneste containerul pentru un timp fix, dat in milisecunde
public void startContainerForFixedTime(long aliveMillis) {
    simpleAsyncTaskScheduler.submit(() ->
    startContainerForFixedTimeInternal(aliveMillis));
}
// executa efectiv pornirea containerului pentru un timp fix
private void startContainerForFixedTimeInternal(long aliveMillis) {
    String queueName = simpleMessageListenerContainer.getQueueNames()[0];
    stopPrevious(); // opreste orice oprire programata
    stopContainerGracefully(); // inchide containerul daca este deja pornit
    log.info("Manually starting container for queue: {} for {} s",
        queueName, Duration.ofMillis(aliveMillis).getSeconds());
    simpleMessageListenerContainer.start();
    containerNotify.notifyContainersStartManual(queueName); // Notifica pornirea
    // manuala
    // programeaza oprirea dupa durata specificata
    stopTask.set(simpleAsyncTaskScheduler.schedule(
        this::stopContainerGracefully,
        new AfterMillisTrigger(aliveMillis)));
}
// initiaza oprirea manuala a containerului
public void stopContainerManually() {
    simpleAsyncTaskScheduler.submit(this::stopContainerManuallyInternal);
}
// opreste containerul daca este pornit (varianta manuala)
private void stopContainerManuallyInternal() {
    String queueName = simpleMessageListenerContainer.getQueueNames()[0];
    log.info("Manually stopping container for queue: {}", queueName);
    stopPrevious();
    stopContainerGracefully();
}

```

```

// anuleaza orice task anterior programat pentru oprirea containerului
private void stopPrevious() {
    ScheduledFuture<?> currentTask = stopTask.get();
    if (currentTask != null && !currentTask.isDone()) {
        currentTask.cancel(false);
        log.info("Cancelled previously scheduled stop task for queue");
    }
}
// opreste containerul si asteapta oprirea completa; daca nu reuseste, forteaza
// shutdown
private void stopContainerGracefully() {
    if (!simpleMessageListenerContainer.isRunning()) {
        log.info("Container is not running. Skipping stop operation.");
        return;
    }
    String queueName = simpleMessageListenerContainer.getQueueNames()[0];
    log.info("Stopping container for queue: {}", queueName);
    try {
        simpleMessageListenerContainer.stop();
        boolean stopped = waitForContainerToStop();
        if (stopped) {
            log.info("Container stopped gracefully for queue: {}", queueName);
        } else {
            log.warn("Container did not stop gracefully. Forcing shutdown for queue: {}",
                    queueName);
            simpleMessageListenerContainer.shutdown();
        }
    } catch (Exception e) {
        log.error("Error while stopping container for queue: {}", queueName, e);
        Thread.currentThread().interrupt();
    }
    // notifica oprirea doar daca containerul chiar s-a oprit
    if (!simpleMessageListenerContainer.isRunning()) {
        containerNotify.notifyContainersStop(queueName);
    } else {
        log.warn("Container did not stop within the expected time: {}",
                Arrays.toString(simpleMessageListenerContainer.getQueueNames()));
    }
}
// verifica repetat daca containerul s-a oprit, cu un timeout configurabil
private boolean waitForContainerToStop() {
    final int[] attempts = { 0 };
    ScheduledFuture<?> scheduledStop = null;
    try {
        // marker pentru a verifica daca containerul s-a oprit
        CompletableFuture<Boolean> containerStopped = new CompletableFuture<>();
        scheduledStop = simpleAsyncTaskScheduler.schedule(() -> {
            log.info("Attempt {} to check if container stopped.", attempts[0]);
            if (!simpleMessageListenerContainer.isRunning()) {
                containerStopped.complete(true);
            } else if (attempts[0]++ >= maxAttempts) {
                containerStopped.complete(false);
            }
        }, new PeriodicTrigger(Duration.ofMillis(intervalMillis)));
        return containerStopped.get((long) maxAttempts * intervalMillis,
                TimeUnit.MILLISECONDS);
    } catch (Exception e) {
        log.error("Error while waiting for container to stop.", e);
        Thread.currentThread().interrupt();
        return false;
    } finally {

```

```

        if (scheduledStop != null && !scheduledStop.isDone()) {
            scheduledStop.cancel(false);
            log.info("Cancelled containerStopped scheduled stop task.");
        }
    }
}

// este apelata automat la distrugerea bean-ului
// opreste containerul si task-ul programat
@PreDestroy
public void destroy() {
    stopPrevious();
    stopContainerGracefully();
}
}

```

Extras Cod-Sursă 3. Controlul ciclului de viață al listenere-lor RabbitMQ în serviciul de arhivare.

```

public class RedisDistributedLockImpl implements RedisDistributedLock {

    // cheia din Redis care reprezinta lock-ul
    private final String lockKey;
    // durata de viata a lock-ului in secunde (TTL)
    private final Long lockTTLsecs;
    private final ReactiveStringRedisTemplate redisTemplate;

    // valoarea unica generata pentru identificarea lock-ului
    private final AtomicReference<String> lockValue = new AtomicReference<>();
    @Override
    public Mono<Boolean> tryAcquireLock() {
        /*
         * Genereaza un UUID pentru acest lock si il seteaza ca valoare
         * unica per instanta microserviciului.
         * Acest UUID va fi folosit pentru a verifica daca lock-ul este detinut de
         * aceasta instanta.
         */
        lockValue.set(UUID.randomUUID().toString());
        // incerc sa seteze cheia in Redis doar daca nu exista deja (NX), cu un TTL
        return redisTemplate
            .opsForValue().setIfAbsent(
                lockKey,
                lockValue.get(),
                Duration.ofSeconds(lockTTLsecs));
    }
    @Override
    public Mono<Boolean> removeLock() {
        // daca nu avem nicio valoare salvata local, nu avem ce lock sa eliberam
        if (lockValue.get() == null) {
            return Mono.just(false);
        }
        return Mono.defer(() -> {
            /*
             * Script Lua care verifica daca valoarea curenta este egala cu cea salvata,
             * si doar atunci sterge cheia
             */
            String script = """
                if redis.call("GET", KEYS[1]) == ARGV[1] then
                    redis.call("DEL", KEYS[1])
                    return true
                end
            """;
            return redisTemplate.execute(script, lockKey, lockValue.get());
        });
    }
}

```

```

        else
            return false
        end
    """
}
RedisScript<Boolean> redisScript = RedisScript.of(script, Boolean.class);
/*
 * executa scriptul in Redis pentru a elibera lock-ul doar daca este detinut de
 * aceasta instanta
 */
return redisTemplate.execute(redisScript,
    List.of(lockKey),
    List.of(lockValue.get()))
.single()
.map(Boolean::booleanValue)
.onErrorResume(e -> {
    // in caz de eroare, logheaza si returneaza false
    log.error("Error while removing lock", e);
    return Mono.just(false);
})
// curata valoarea locala a lock-ului indiferent de rezultat
.doFinally(_ -> {
    lockValue.set(null);
});
}
}
}

```

Extras Cod-Sursă 4. Blocări distribuite în Redis.

```

public void addFluxToCacheAllCachesMiss() {
    // apeleaza metoda reala care furnizeaza lista de dummies (fara caching)
    var res = testServiceReactive.getDummies();

    // verifica daca serviciul cache-uit returneaza aceleasi date ca cele originale
    StepVerifier.create(testServiceReactive.getAllDummies().collectList())
        .expectNextMatches(r -> r.equals(res))
        .verifyComplete();
    // capteaza cheia de cache si cheia de reverse index folosita
    var savingKeyCaptor = ArgumentCaptor.forClass(String.class);
    var reverseIndexCaptor = ArgumentCaptor.forClass(String.class);
    // asteapta pana cand toate operatiile asincrone s-au efectuat si se pot
    // verifica
    await().atMost(AssertionTestUtils.AWAITABILITY_TIMEOUT_SECONDS)
        .untilAsserted(() -> {
            // verifica daca cheia a fost extrasă din adnotare
            verify(aspectUtils,
                atLeastOnce()).extractKeyFromAnnotation(eq(TestServiceReactive.CACHE_KEY), any());
                // verifica daca hash-ul cheii a fost generat corect
                verify(aspectUtils, atLeastOnce()).getHashString(any(),
                    eq(TestServiceReactive.CACHE_KEY),
                    eq("getAllDummies"));
                // verifica daca fluxul de date a fost creat pentru caching
                verify(redisReactiveCacheAspect,
                    atLeastOnce()).createBaseFlux(savingKeyCaptor.capture(),
                        any(Method.class));
                // verifica daca s-a incercat accesarea datelor din cache-ul local si din
                redis
                    // (cache miss)
                    verify(localReactiveCache,
                        atLeastOnce()).getFluxOrEmpty(savingKeyCaptor.getValue());

```

```

        verify(reactiveValueOperations,
atLeastOnce()).get(savingKeyCaptor.getValue());

        // verifica daca raspunsul a fost salvat in redis cu ttl de 30 minute
        verify(reactiveValueOperations, atLeastOnce())
            .set(eq(savingKeyCaptor.getValue()), eq(res),
eq(Duration.ofMinutes(30)));

        // verifica daca fiecare entitate a fost adaugata in reverse index (pentru
        // invalidare)
        verify(redisReactiveCacheAspect,
atLeastOnce()).addToReverseIndex(TestServiceReactive.CACHE_KEY, 1L,
savingKeyCaptor.getValue());
        verify(redisReactiveCacheAspect,
atLeastOnce()).addToReverseIndex(TestServiceReactive.CACHE_KEY, 2L,
savingKeyCaptor.getValue());
        verify(redisReactiveCacheAspect,
atLeastOnce()).addToReverseIndex(TestServiceReactive.CACHE_KEY, 3L,
savingKeyCaptor.getValue());

        // verifica daca metoda a fost salvata complet in cache prin aspect
        verify(redisReactiveCacheAspect, atLeastOnce())
            .methodFluxResponseToCache(any(ProceedingJoinPoint.class),
eq(TestServiceReactive.CACHE_KEY),
eq(savingKeyCaptor.getValue()), anyString(), eq(true));

        // verifica daca cheia a fost adaugata intr-un set pentru reverse index in
redis
        verify(reactiveSetOperations,
atLeast(res.size())).add(reverseIndexCaptor.capture(),
eq(savingKeyCaptor.getValue()));

        // asteapta ca setul reverse index sa fie expirat corect in redis
await().atMost(AssertionTestUtils.AWAITABILITY_TIMEOUT_SECONDS)
.untilAsserted(() -> {
    verify(reactiveRedisTemplate, atLeastOnce())
        .expire(reverseIndexCaptor.getValue(),
Duration.ofMinutes(31)); // 30 ttl + 1 extra
});

        // verifica daca reverse index-ul a fost salvat si in cache-ul local
verify(reverseKeysLocalCache, atLeastOnce())
.add(reverseIndexCaptor.getValue(), savingKeyCaptor.getValue());

        // verifica din nou salvarea valorii in redis
verify(reactiveValueOperations, atLeastOnce())
.set(eq(savingKeyCaptor.getValue()), eq(res),
eq(Duration.ofMinutes(30)));

        // verifica din nou expirarea setului reverse index
verify(reactiveRedisTemplate, atLeastOnce())
.expire(reverseIndexCaptor.getValue(), Duration.ofMinutes(31));

        // verifica daca nu s-au apelat metodele pentru caching de tip mono
verify(aspectUtils, never()).evaluateSpelExpression(anyString(), any());
verify(redisReactiveCacheAspect, never()).methodMonoResponseToCache(any(),
any(), any(), any(), any());

```

```

        verify(redisReactiveCacheAspect, never()).saveMonoResultToCache(any(),
any(), any(), any(), any());
        verify(redisReactiveCacheAspect, never()).saveMonoToCacheNoSubscribe(any(),
any(), any(), any());

        // verifica daca objectMapper nu a fost folosit pentru convertiri
neasteptate
        verify(objectMapper, never()).convertValue(any(), any());

        // verifica daca raspunsul a fost salvat si in cache-ul local
        verify(localReactiveCache,
atLeastOnce()).put(eq(savingKeyCaptor.getValue()), anyList());
    });

// validari finale pentru a confirma ca datele se gasesc efectiv in cache
await().atMost(AssertionTestUtils.AWAITABILITY_TIMEOUT_SECONDS)
    .untilAsserted(() -> {

        // verifica daca local cache returneaza raspunsul corect
        StepVerifier.create(localReactiveCache.getFluxOrEmpty(savingKeyCaptor
.getValue()).collectList())
            .expectNextMatches(res::equals)
            .verifyComplete();
        // verifica daca setul din redis si reverse cache-ul local contin aceleasi
        chei
        StepVerifier.create(
            reactiveRedisTemplate.opsForSet()
                .members(reverseIndexCaptor.getValue())
                .collectList())
            .expectNextMatches(redisList -> new HashSet<>(redisList)
                .equals(new
        HashSet<>(reverseKeysLocalCache.get(reverseIndexCaptor.getValue()))))
            .verifyComplete();

        // verifica daca valoarea cache-uita in redis poate fi citita si convertita
        corect
        StepVerifier.create(
            reactiveRedisTemplate.opsForValue().get(savingKeyCaptor.getValue())
                .map(v -> objectMapper.convertValue(v,
        objectMapper.getTypeFactory())
                    .constructCollectionType(List.class,
                        objectMapper.getTypeFactory()
                            .constructType(TestServiceReactive.
        Dummy.class))))))
            .expectNext(res)
            .verifyComplete();
    });
}

```

Extras Cod-Sursă 5. Test salvare publisher cu mai mult de un element în cache-ul din back-end când valoarea nu este prezentă în niciun cache.

```

public Flux<DataBuffer> getVideoByRange(ReactiveGridFsResource file, long start,
    long end) {
    if (start > end || start < 0) {
        return Flux.empty();
    }
    // masura fiecareui chunk luat din metadatele fisierului
    int chunkSize = file.getOptions().getChunkSize();
    // chunk-ul de inceput
    int startChunk = (int) (start / chunkSize);
    // chunk-ul de sfarsit
    int endChunk = (int) (end / chunkSize);
    // offset-ul pentru primul chunk
    int headOffset = (int) (start % chunkSize);
    // offset-ul pentru ultimul chunk
    int tailOffset = (int) (end % chunkSize);
    /*
     * Extrage manual doar chunk-urile necesare, pentru a nu incarca
     * toata resursa in memorie. De asemenea, chunk-urile sunt
     * ordonate crescatoare in Mongo, deci nu este nevoie
     * sa se sorteze si in Java.
     */
    Query q = Query.query(Criteria
            .where("files_id").is(file.getId())
            .and("n").gte(startChunk).lte(endChunk)).with(Sort.by("n"));
    return reactiveMongoTemplate
        .find(q, Document.class, "fs.chunks")
        // limiteaza throughput-ul pentru a nu supra-solicita serverul
        .limitRate(videoLimitRate, videoLimitRate / 2)
        // mapeaza fiecare chunk in: index-ul sau si datele binare
        .map(doc -> {
            int chunkIdx = doc.getInteger("n");
            Binary bin = doc.get("data", Binary.class);
            DataBuffer buf = dataBufferFactory.wrap(bin.getData());
            return Tuples.of(chunkIdx, buf);
        })
        .map(tuple -> {
            // indexul chunk-ului
            int n = tuple.getT1();
            // chunk-ul in sine, indexarea acestuia este de la 0
            DataBuffer buf = tuple.getT2();
            // un singur chunk
            if (startChunk == endChunk) {
                int length = tailOffset - headOffset + 1;
                buf.split(headOffset); // scoate [0..headOffset]
                return buf.split(length); // buf incepe de la headOffset
            }
            // primul chunk
            if (n == startChunk) {
                buf.split(headOffset); // scoate [0..headOffset]
                return buf; // buf incepe de la headOffset
            }
            // ultimul chunk
            if (n == endChunk) {
                return buf.split(tailOffset + 1); // scoate [tailOffset..end]
            }
            // chunk intermediar
            return buf;
        });
}

```

Extras Cod-Sursă 6. Interogarea bazei de date și procesarea stream-ului binar pentru un request de tip Range.

```

def get_pipeline() -> StableDiffusionPipeline:
    # foloseste o variabila globala pentru a reutiliza pipeline-ul deja incarcat
    global pipeline

    # daca pipeline-ul este deja incarcat, il returneaza direct
    if pipeline is not None:
        return pipeline
    # foloseste un lock pentru a evita conditii de concurenta in care mai multe thread-uri
    # incearca sa initializeze pipeline-ul simultan
    with pipeline_lock:
        if pipeline is not None:
            return pipeline
        # detecteaza daca GPU-ul este compatibil cu bfloat16
        # OBS: placile de pe server sunt vechi si nu sunt compatibile
        p8_cuda = torch.cuda.get_device_capability(0)[0] >= 8
        # seteaza dtype-ul optim in functie de capacitatea GPU-ului
        torch_dtype = torch.bfloat16 if p8_cuda else torch.float16
        # verifica daca modelul este disponibil local
        if os.path.exists(LOCAL_MODEL_PATH):
            pipe = load_from_local(safety_checker, torch_dtype)
        else:
            # daca modelul nu exista local, il descarca si il salveaza
            pipe = StableDiffusionPipeline.from_pretrained(
                MODEL_ID, torch_dtype=torch_dtype
            )
            pipe.save_pretrained(LOCAL_MODEL_PATH)
            # reincarca modelul local dupa salvare
            pipe = load_from_local(safety_checker, torch_dtype)
        # muta pipeline-ul pe device-ul specificat (ex: cuda0)
        pipe = pipe.to(DEVICE)
        # activeaza attention slicing pentru reducerea consumului de memorie
        pipe.enable_attention_slicing("auto")
        if p8_cuda:
            # activeaza atentie eficienta pentru GPU-urile compatibile cu xformers
            # OBS: xformers nu este compatibil cu GPU-urile de pe server
            pipe.enable_xformers_memory_efficient_attention(
                attention_op=MemoryEfficientAttentionFlashAttentionOp)
            pipe.vae.enable_xformers_memory_efficient_attention(
                attention_op=None)
        # seteaza tipul de date pentru UNet si VAE
        pipe.unet.to(dtype=torch_dtype)
        pipe.vae.to(dtype=torch_dtype)
        if PIPE_ENABLE_GRADIENT_CHECKPOINTING:
            # activeaza gradient checkpointing pentru a reduce utilizarea memoriei
            pipe.unet.enable_gradient_checkpointing()

        # salveaza pipeline-ul global pentru reutilizare ulterioara
        # se implementeaza un 'singleton'
        pipeline = pipe
    return pipeline

```

Extras Cod-Sursă 7. Crearea pipeline-ului de difuzie folosind HuggingFace.