UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Arhitecturi bazate pe microservicii: eQuiz

propusă de

Vitalie Bocicov

Sesiunea: iulie, 2024

Coordonator științific

Lect. Dr. Cristian Andrei Frăsinaru

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Arhitecturi bazate pe microservicii: eQuiz

Vitalie Bocicov

Sesiunea: iulie, 2024

Coordonator științific

Lect. Dr. Cristian Andrei Frăsinaru

Avizat,

Îndrumător lucrare de licență,

Lect. Dr. Cristian Andrei Frăsinaru.

Data: ..... Semnătura: .....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Bocicov Vitalie** domiciliat în **România, localitatea Bacău, jud. Bacău, Strada Letea, nr. 9, sc. B, ap. 20**, născut la data de **08 octombrie 2000**, identificat prin CNP **5001008226859**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2024, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Arhitecturi bazate pe microservicii: eQuiz** elaborată sub îndrumarea domnului **Lect. Dr. Cristian Andrei Frăsinaru**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: ..... Semnătura: .....

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Arhitecturi bazate pe microservicii: eQuiz**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea ”Alexandru-Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Vitalie Bocicov**

Data: ..... Semnătura: .....

Cuprins

Introducere

Digitalizarea este necesară în majoritatea domeniilor din ziua de azi, dar mai ales în educație. Tocmai din acest motiv, Facultatea de Informatică, din cadrul Universității "Alexandru Ioan Cuza" folosește aplicația web „eQuiz" pentru a administra testele online și pentru a evalua studenții.

Deși aplicația „eQuiz" servește în mod eficient cerințelor curente, fiind dezvoltată folosind o arhitectură monolitică, odată cu o posibilă adoptare a aplicației de către alte instituții de învățământ, aceasta ar putea să înceapă să-și arate limitările. O asemenea adoptare la scară mai largă, ar aduce cu sine mai mulți utilizatori ai aplicației, mai ales în mod concurent, ce ar pune o presiune considerabilă pe sistem. Sistemul, fiind solicitat la maxim, ar aduce timp de așteptare mai mare pentru utilizatori, sau, în cel mai rău caz, ar putea fi inaccesibil din cauza suprasolicitării.

O altă cerință posibilă ar fi necesitatea de personalizare la cererea instituțiilor, fiind mai greu de îndeplinit într-o arhitectură monolitică, componentele aplicației fiind strâns cuplate.

Aplicațiile monolitice vin cu provocări când vorbim de scalabilitate și sunt, în general, mai greu de modificat, pentru că sunt masive, iar pe lângă faptul că lucrul la o singură bază de cod aduce complexitatea sincronizării la nivelul echipei de dezvoltare, o singura greșeală ar putea cauza prăbușirea întregului sistem în producție.

Deși arhitectura monolitică este preferabilă în prima faza de dezvoltare a unui sistem, pentru că reduce complexitatea dezvoltării, odată cu creșterea cererii și a numărului de utilizatori, migrarea la o arhitectura bazată pe microservicii, în cazuri specifice, poate soluționa limitările monolitului.

Arhitectura bazată pe microservicii, deși oferă multe beneficii, aduce cu sine o serie nouă de provocări. Printre acestea se numără complexitatea sistemului, dificultatea de a-l înțelege și depanarea eventualelor erori, prelucrarea datelor, testarea și securitatea, fiind un sistem distribuit.

Mi-am propus, în lucrarea de față, să migrez aplicația facultătii "eQuiz" de la o arhitectură monolitică, la o arhitectură bazată pe microservicii, iar cea din urmă va soluționa limitările enunțate anterior, cu privire la scalabilitate, flexibilitatea de dezvoltare, reziliență, și va aduce și beneficii precum utilizarea resurselor într-un mod eficient, în eventuala integrării aplicației "eQuiz" de către alte instituții educaționale și nu numai.

Adoptarea arhitecturii bazate pe microservicii, mi-a permis să utilizez patternurile "Aggregator" și "Database per microservice", precum și tehnologii moderne: Spring Boot, Docker, Kubernetes, Prometheus, Grafana, cu scopul de a îmbunătăți aplicația "eQuiz" pentru o utilizare la scară largă.

Capitolul 1

Specificații functionale

1.1 Descrierea problemei

Aplicația web "eQuiz" a fost dezvoltată folosind o arhitectură monolitică, ceea ce înseamnă că întregul sistem constă dintr-un singur bloc de cod și toate operațiile sunt efectuate local. Această arhitectură aduce multe beneficii [1], printre care: dezvoltarea inițială rapidă, interacțiuni simplificate cu sistemul, depanarea și identificarea erorilor cu o viteza mai mare, operații mai eficiente, simplificarea procesului de deployment și de testare, dat fiind faptul că toate interacțiunile dintre componentele sistemului se efectuează local, și avem de-a face cu un singur sistem.

Trebuie să menționăm și potențialele dezavantaje ale arhitecturii monolitice [2]. Odată cu creșterea cerințelor de sistem, crește și dimensiunea monolitului, care poate ajunge la proporții considerabile. Autonomia echipei de dezvoltare scade odată cu creșterea numărului membrilor, din cauza faptului că toți contribuie și lucrează la același sistem, fapt ce aduce cu sine complexitatea coordonării și sincronizării. Un alt dezavantaj ar fi pipeline-ul deployment-ului cu o viteza redusă, pentru că la fiecare modificare trebuie construit și testat întregul proiect, timp de așteptare care ar crește odată cu dimensiunea sistemului.

Având un singur bloc de cod, apare constrângerea de a folosi aceleași tehnologii și același limbaj de programare în întreg proiectul, care poate aduce anumite limitări, odată cu obligația de a folosi instrumentele ce sunt compatibile cu sistemul, în timp ce un alt instrument sau limbaj de programare ar putea rezolva într-un mod mai eficient o anumită problemă.

Reziliența sistemului este un alt punct slab când vorbim despre această arhi-

tectură. Dacă sistemul ajunge într-o stare din care nu își poate reveni, deși nu este neapărat o eroare fatală din punct de vedere al funcționalității, tot sistemul se va prăbuși.

Deși arhitectura monolitică este, în teorie, scalabilă, un astfel de sistem nu ar permite gestionarea resurselor în mod eficient. În aplicația "eQuiz", doar unele părți ale sistemului ar putea fi intens solicitate, de exemplu, componenta care se ocupă de gestionarea testelor în timp real, în momentul unui test activ, pe când partea de sistem ce se ocupă de crearea de subiecte și întrebări ar putea sa nu fie folosită deloc sau nesemnificativ în acel moment. Scalând întregul sistem, ar presupune să avem mai multe instanțe ale monolitului, ce ar genera un consum mai mare de resurse utilizate în mod ineficient.

Pentru a soluționa limitările aplicației "eQuiz" pentru cazul în care cererea și numărul de utilizatori va crește, am decis sa migrez sistemul la o arhitectură bazată pe microservicii.

Arhitectura bazată pe microservicii oferă multe beneficii [3], întrucât componentele aplicației fiind izolate, datorită organizării acestora în microservicii, permit dezvoltarea individuală a componentelor. Astfel, echipa poate crește odată cu dimensiunea sistemului, implementarea poate fi făcută individual la nivel de microserviciu, reducând complexitatea sincronizării membrilor și echipelor, micșorând totodată timpul de dezvoltare a cerințelor.

O bună organizare a microserviciilor poate facilita înțelegerea sistemului, datorită modularității componentelor, organizate după funcționalități, astfel încât sistemul poate fi înțeles în ansamblu. Operațiunile de build și deployment ale aplicației devin mai rapide, pentru că doar componentele modificate vor trece din nou prin acest proces. Microserviciile pot fi implementate folosind limbaje de programare și tehnologii diferite, astfel încât să rezolve cât mai bine usecase-ul pe care îl gestionează.

Sistemul devine mai rezilient, datorită faptului că aceste componente sunt izolate: dacă un microserviciu se va prăbuși, sistemul ca un întreg va fi încă accesibil, microserviciul va fi repornit sau o altă instanță a acestuia va fi pornită.

Această arhitectură, prin separarea componentelor în microservicii, permite scalabilitatea aplicației doar la nivel de microserviciu, facilitând utilizarea optimă a resurselor. Scalarea pe verticală și pe orizontală permite sistemului să gestioneze eficient cererile sporite fără a compromite performanța.

Arhitectura aplicației "eQuiz" oferă toate beneficiile menționate anterior, cu aju-

torul integrării tehnologiilor Docker, care creeaza containere din microservicii, Kubernetes, ce orchestrează aceste containere, Prometheus, ce colectează și stochează metricile din cluster, și Grafana, responsabilă de vizualizarea metricilor colectate într-o manieră grafică.

1.2 Flux de lucru

Aplicația "eQuiz" are două tipuri de utilizatori: studenți, cei care sunt evaluați prin intermediul testelor, și instructori, cei care creează subiectele, întrebările și testele.

Studenții se autentifică în aplicație folosind datele primite de la instructori. Odată autentificați își vor putea modifica parola primită din motive de securitate. Aceștia vor fi întâmpinați de un dashboard, în care vor putea să vizualizeze testele la care sunt invitați să participe, istoricul cu testele deja efectuate și notificari relevante din partea instructorilor.

În momentul în care un test va fi disponibil, la o anumită perioada de timp, studentul va putea începe evaluarea, având un timp limitat ca să răspundă la întrebări. Acesta va înregistra răspunsurile la întrebări, astfel completând testul. După ce testul va fi verificat, manual sau automat, de către instructori sau de către aplicație, în funcție de preferință instructorilor, studentul va putea vedea rezultatul testului și feedback-ul primit de la instructor, dacă este cazul.

Instructorii se pot înregistra în aplicație folosind un email asociat instituției din care fac parte, după care se pot autentifica. Pagina principala va fi alcătuită dintr-un meniu de control, din care un instructor poate crea teste, verifica teste efectuate, manual sau automat și trimite notificări studenților.

Capitolul 2

Arhitectura sistemului

2.1 Arhitectura internă a microserviciilor

La nivel intern al fiecărui microserviciu am implementat arhitectura pe straturi "Layered Architecture"[6], aderând la principiul "Separation of concerns" [7], fiecare nivel fiind responsabil de o singură funcționalitate. Această arhitectură facilitează organizarea codului într-un mod clar și ușor de înțeles.

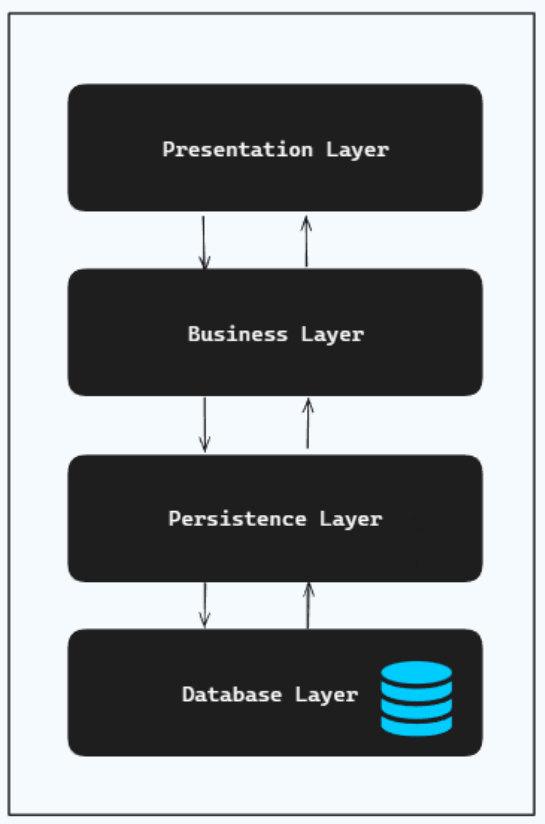

Figura 2.1: Layered Architecture

În continuare voi descrie fiecare strat în parte:

1. Presentation Layer: La acest nivel sunt implementate Controllerele, iar microserviciile sunt implementate conform standardul API-urilor RESTful, aici sunt expuse endpoint-urile specifice, fiind gestionate cererile de la clienți prin intermediul protocolului HTTP/S, și oferind răspunsuri în formatul JSON.

2. Business Layer: Acest strat conține clasele de tip Service, ce efectuează logica business a aplicației, fiind totodată responsabile de comunicarea între stratul de prezentare și cel de date.

3. Persistence Layer: Acesta este stratul care conține clasele Model, ce modelează baza de date și clasele Repository, ce se ocupă cu gestionarea și persistența datelor într-un mod abstract, fără a fi legat de o implementare a bazei de date specifică.

4. Database Layer: Acest strat este mai specific și se referă direct la bazele de date utilizate pentru stocarea datelor.

Fiecare strat din acest design poate comunica doar cu straturile vecine, iar transferul de date între acestea se efectuează prin intermediul DTO-urilor (Data Transfer Object). Motivația folosirii DTO-urilor constă în decuplarea straturilor, ce aduce beneficii, precum securitatea sporită, care reduce riscul de expunere a datelor sensibile în exterior și flexibilitate, care simplifică procesul de actualizare a modelelor.

Este de menționat că fiecare microserviciu este stateless, adică nu păstrează o stare internă, două cereri identice returnează, în general, același rezultat. Această caracteristică simplifică scalarea microserviciilor, pentru că fiecare instanța poate răspunde la orice cerere fără a depinde de date locale. Astfel, adăugarea sau eliminarea instanțelor pentru a gestiona cererile devine mult mai eficientă, facilitând distribuția echilibrată a traficului și optimizarea resurselor.

2.2 Arhitectura microserviciilor

2.2.1 Descompunerea monolitului în microservicii

Pentru a putea descompune aplicația "eQuiz" în microservicii, a fost necesară înțelegerea componentelor sistemului și responsabilitățile acestora, în funcție de logica business.

În urma studiul inițial am identificat următoarele componente:

- Domain Service: gestionează instituțiile ce utilizează aplicația (facultăți, școli, etc), asigurând astfel o organizare a subiectelor, testelor și utilizatorilor în funcție de instituția din care fac parte.
- Session Service: gestionează sesiunile academice, similar structurii semestrelor și anilor de studiu din contextul academic, asigurând o organizare temporală.
- User Service: gestionează instructorii, studenții și grupurile de studenți, se ocupă de înregistrare, autentificare și administrarea conturilor, asigurând controlul accesului și personalizarea aplicației în funcție de rolurile utilizatorilor.
- Course Service: facilitează organizarea și structurarea conținutului aplicației, în secțiuni și subiecte, a întrebărilor, în funcție de materia ce urmează a fi evaluată, permițând astfel asocierea conținutului cu instructorii și studenții, prin intermediul grupurilor de studenți.
- Test Service: gestionează testele, crearea de întrebări, monitorizarea participanților, colectarea răspunsurilor, verificarea testelor și afișarea rezultatelor.
- Image Service: gestionează încărcarea, stocarea și recuperarea imaginilor folosite în materialele didactice ale cursurilor și a testelor.
- Setting Service: administrează configurațiile aplicației, conform preferințelor specifice ale fiecărei instituții.

După ce am identificat componentele menționate anterior, am creat microservicii din fiecare componentă și am obtinut astfel prima versiune a arhitecturii aplicației:

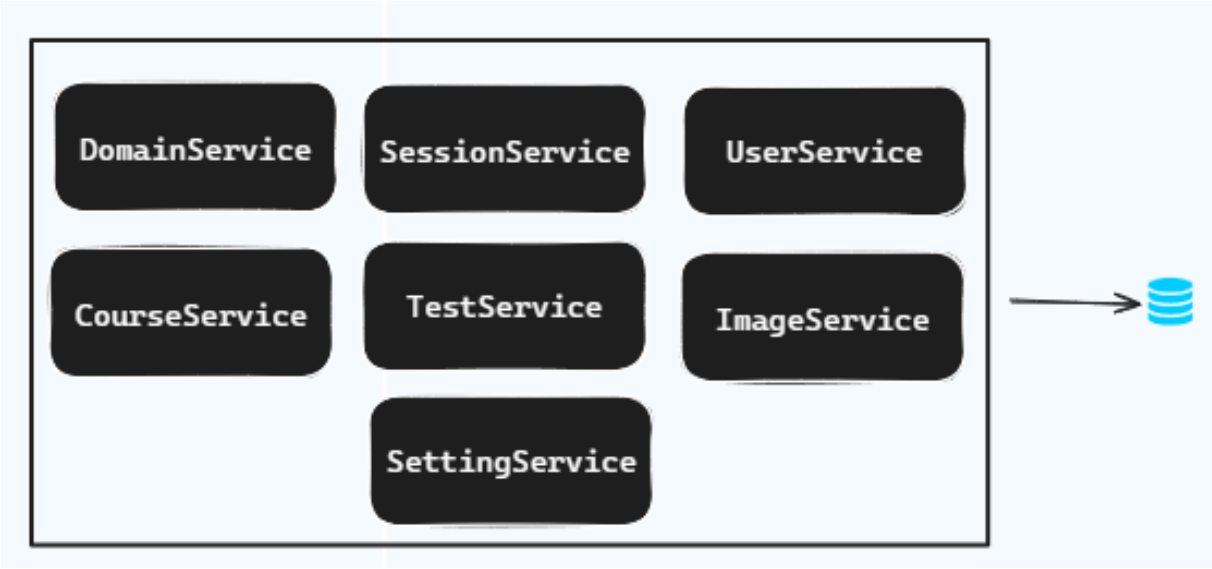

Figura 2.2: Arhitectura aplicației “eQuiz”, versiunea 1

2.2.2 Database per Microservice pattern

În urma implementării arhitecturii din Figura 2.2, am realizat că sistemul are un bottleneck semnificativ, din cauza faptului că toate microserviciile partajează o singură bază de date. Astfel, chiar dacă microserviciile sunt scalate atunci când cererea este sporită, există riscul de a apărea latență din cauza posibilei suprasolicitări a bazei de date, aducând simultan și riscul a unui punct singular de eșec.

Drept soluție, am decis să folosesc “Database per Microservice pattern” [4], ce presupune descompunerea bazei de date astfel încât fiecare bază de date să fie dedicată unui singur serviciu.

Această descompunere asigură independența microserviciilor, gestionând datele în mod izolat, facilitând scalabilitatea și flexibilitatea bazei de date, reducând astfel bottleneck-ul aplicației.

Deși aplicarea pattern-ului “Database per microservice” a rezolvat limitările utilizării unei singure baze de date de către toate microserviciile, a introdus o provocare nouă: consistența datelor.

În acest model arhitectural, cu bazele de date distribuite, avem de-a face cu o consistență eventuală a datelor, ce adaugă complexitate gestionării și sincronizării a datelor. Deși datele vor fi sincronizate, nu putem ști cu certitudine când se va întâmpla această operație.

În urma descompunerii bazei de date, a rezultat a doua versiune a arhitecturii

aplicației “eQuiz”:

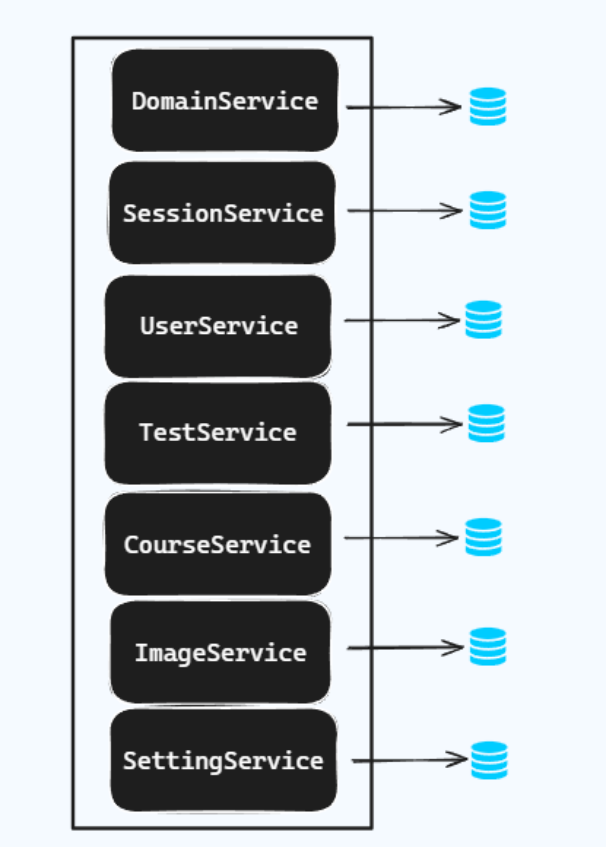

Figura 2.3: Arhitectura aplicației “eQuiz”, versiunea 2

2.2.3 Service Aggregator Pattern

Pentru a simplifica complexitatea gestionării datelor în aplicația “eQuiz”, fiind un sistem distribuit, am aplicat “Service Aggregator Pattern” [5], care acționează ca un mediator între microservicii, ce agregă datele între acestea.

Acest pattern decuplează microserviciile și simplifică interacțiunea cu sistemul, oferind un punct de acces unic pentru funcționalitățile ce au nevoie de date de la mai multe părți ale sistemului. Prin decuplarea microserviciilor, reducem dependențele directe dintre microservicii, ce permite o dezvoltare independentă a funcționalităților.

De asemenea, agregatorii pot implementa strategii de caching, reducând astfel numărul de cereri real de date către microservicii și îmbunătățind astfel performanța sistemului.

Numărul microserviciilor a scăzut de la 7 la 4, iar numărul agregatorilor a scăzut de la 4 la 3, “Session Aggregator” fiind eliminat.

După aplicarea acestui pattern, a rezultat versiuna cu numărul trei a arhitecturii aplicației eQuiz:

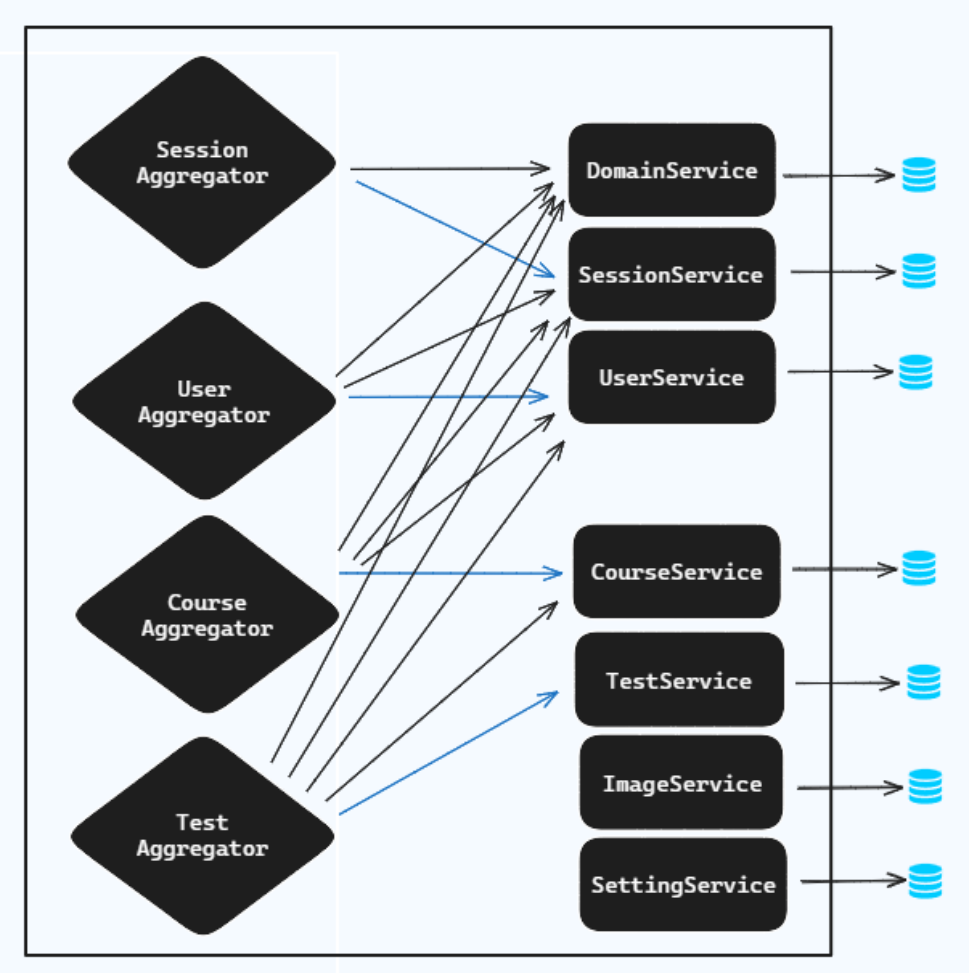

Figura 2.4: Arhitectura aplicației "eQuiz", versiunea 3

2.2.4 Arhitectura finală a sistemului

În continuare am integrat un gateway în arhitectura aplicației, cu scopul de a avea un unic punct de acces din exterior, ce rezultă în o mai bună gestionare a traficului pentru un control îmbunătățit asupra microserviciilor, fapt ce sporește securitatea sistemului.

În cele din urmă, am ajuns la o versiune ce satisface cerințele inițiale, cel puțin temporar. După ce am introdus Gateway-ul în sistem, am creat containere Docker din toate componentele aplicației și le-am dat deploy containerelor într-un cluster Kubernetes.

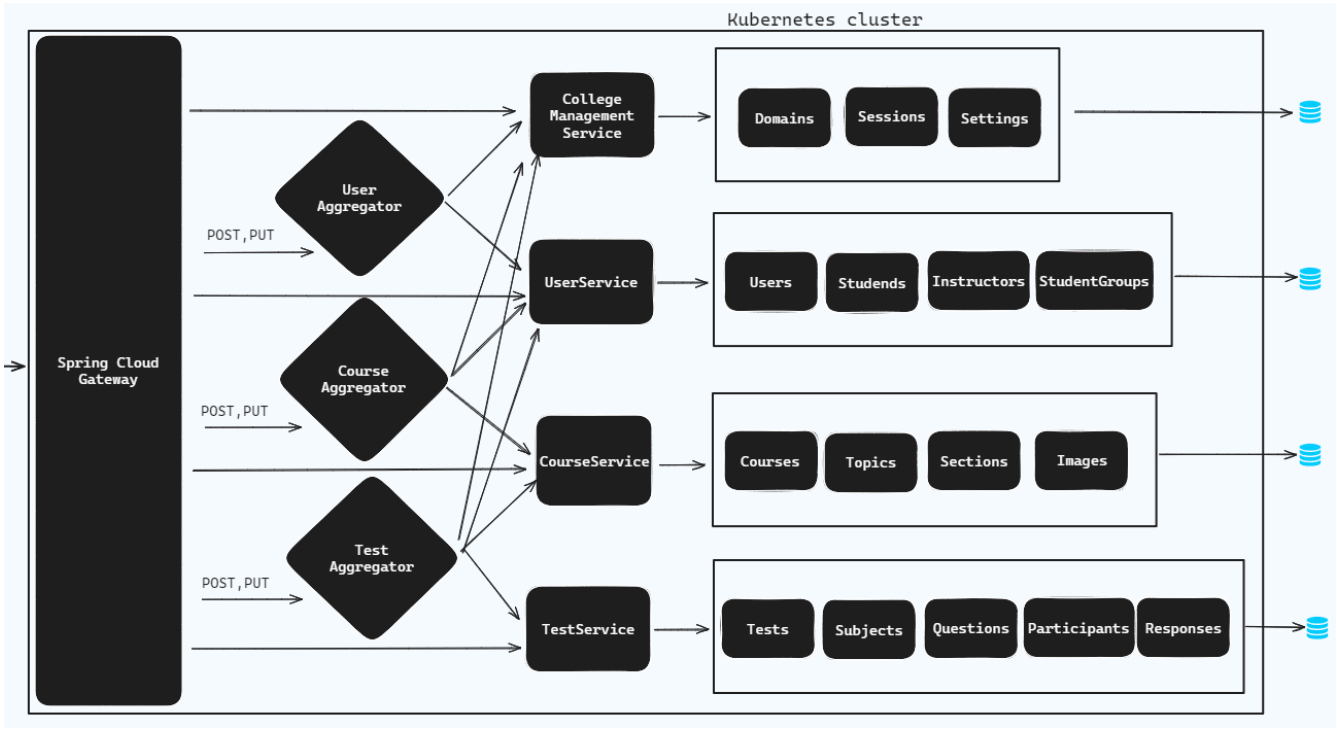

Figura 2.5: Arhitectura aplicației “eQuiz”, versiunea 4

2.3 Arhitectura bazei de date

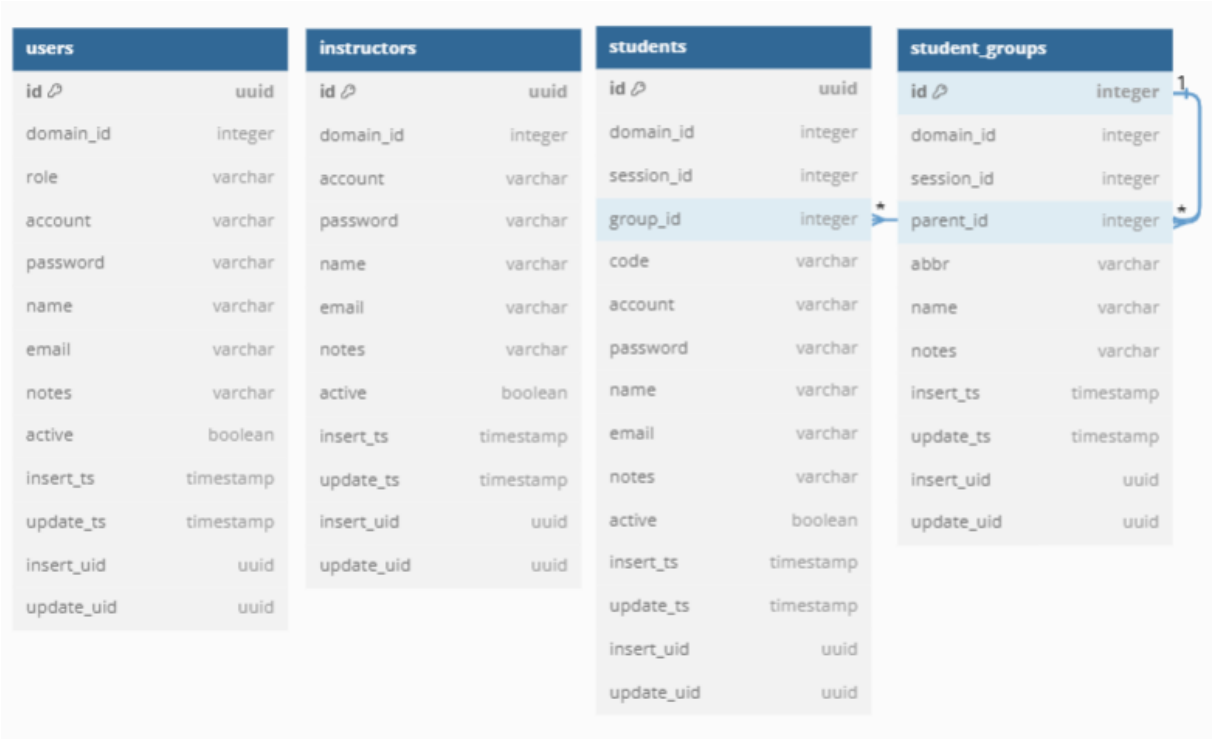

Figura 2.6: Baza de date a componentei User Service

Figura 2.7: Baza de date a componentei Domain Service

Figura 2.8: Baza de date a componentei Course Service

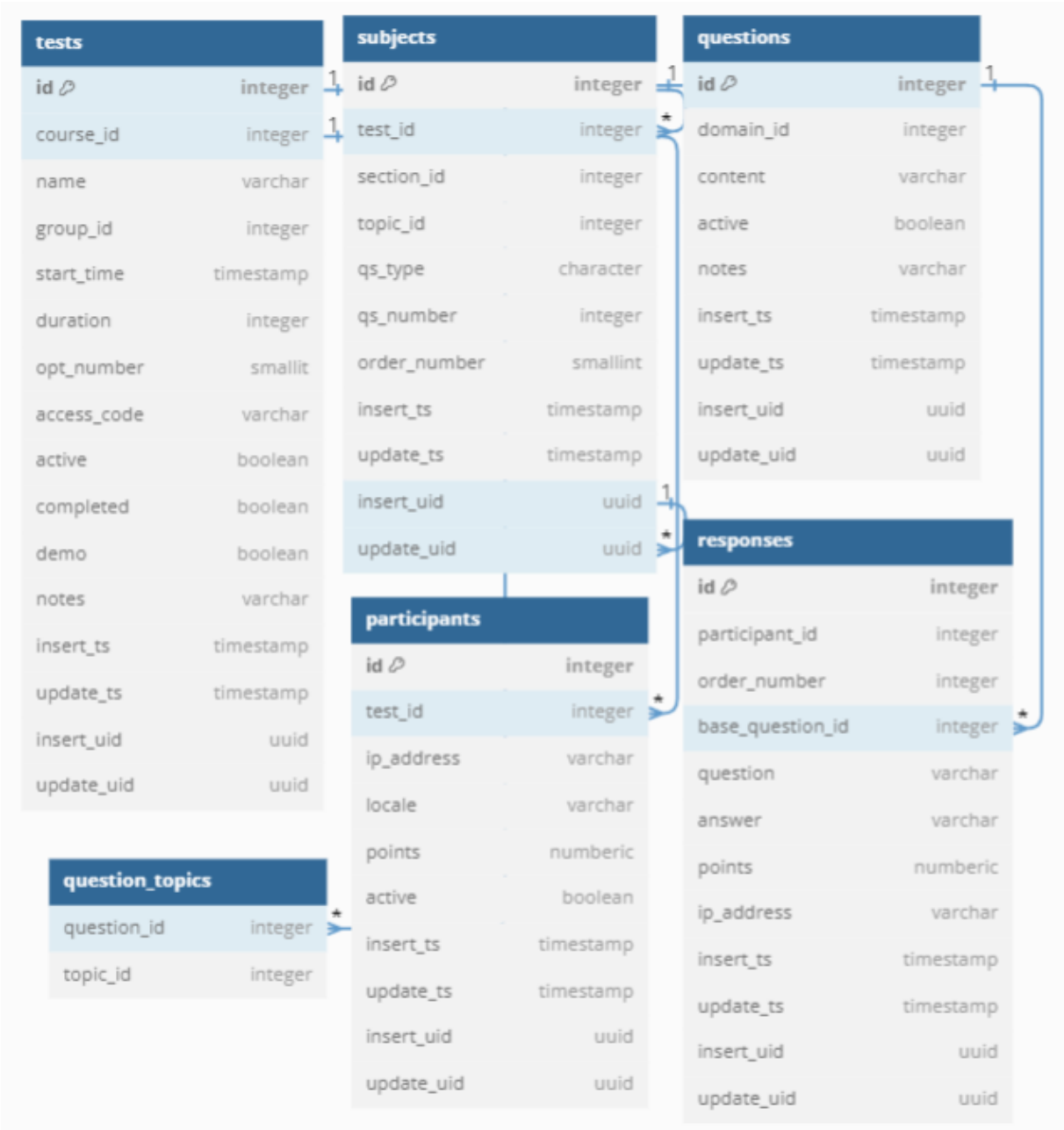

Figura 2.9: Baza de date a componentei Test Service

2.4 Arhitectura clusterului

Pentru colectarea și vizualizarea metricilor aplicației, am integrat tehnologiile Prometheus și Grafana. Această integrare permite înțelegerea și observabilitatea sistemului în timp real, ce facilitează detectarea rapidă a problemelor ce pot apărea.

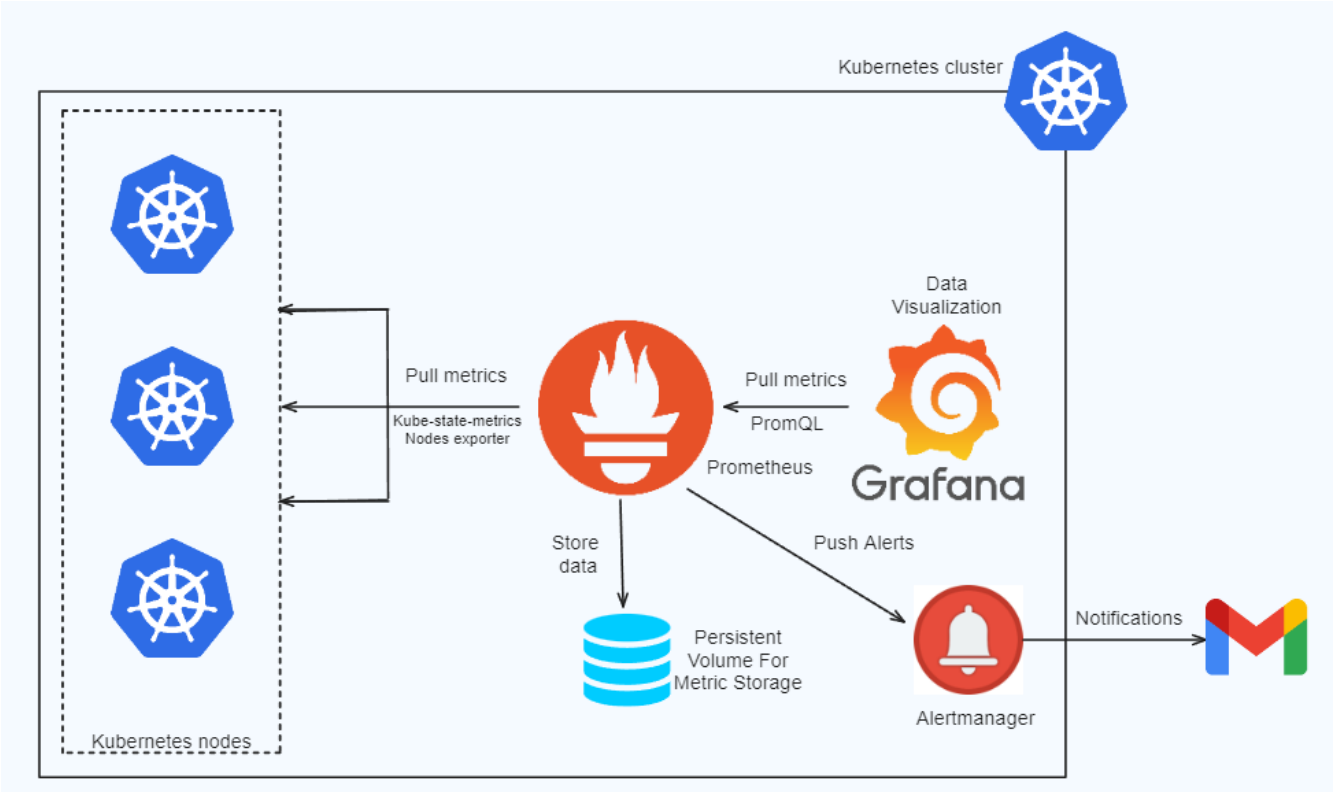

Figura 2.10: Arhitectura clusterului aplicației eQuiz

Capitolul 3

Tehnologii folosite

3.1 PostgreSQL

Pentru baza de date am folosit soluția ce era deja utilizată pentru aplicația "eQuiz", PostgreSQL. Acest sistem de gestionare a bazelor de date relaționale este open-source, puternic, robust, flexibil și cu securitate solidă. PostgreSQL oferă tranzacții ACID (Atomicity, Consistency, Isolation, Duration) și suportă toate tipurile de date necesare aplicației "eQuiz".

3.2 Spring Boot

Microserviciile au fost implementate în limbajul de programare Java, în cadrul framework-ului Spring Boot, ce face parte din ecosistemul Spring. Acest framework facilitează dezvoltarea rapidă a aplicației, automatizând configurația inițială și minimizând codul boilerplate prin adnotări. Vine cu un server încorporat, Tomcat și cu pachete "starter" pentru simplificarea dezvoltării aplicației, oferă configurare automată a framework-ul Spring și a librăriilor terțe, când este posibil.

Include functionalități ce favorizează monitorizarea, colectarea metricilor și datelor cu privire la sănătătatea microserviciilor, simplificând integrarea cu tehnologia Prometheus.

Alături de Spring Boot, am utilizat pachete suplimentare din ecosistem:

- Spring Starter Web: este un starter kit ce include tot ce este necesar pentru un server web, suport pentru API-uri RESTful, ce utilizează Tomcat ca server încorporat.

Acest pachet simplifică configurarea necesară pentru aplicațiile web, prin utilizarea adnotărilor.

- Spring Data: facilitează integrarea rapidă cu baza de date, de asemenea, gestionează operațiunile CRUD și oferă suport pentru interogari complexe. Acest pachet abstractizează implementarea bazei de date.
- Spring Boot Actuator: oferă capabilități extinse de monitorizare și gestionare a aplicațiilor, precum informații despre starea și performanța aplicației în timp real, prin expunerea diverselor endpoint-uri HTTP.
- Lombok: este un instrument ce reduce codul boilerplate din Java, cum ar fi getteri, setteri, constructori generând la compilare codul necesar, prin utilizarea adnotărilor.
- Mapstruct: este un instrument ce facilitează transformarea DTO-urilor în modele și invers.

3.3 Docker

Docker este o platformă ce facilitează dezvoltarea, livrarea și rularea aplicațiilor într-un mod simplu, independent de infrastructură. Docker împachetează aplicația alături de toate dependețele acesteia, care este rulată într-un mediu izolat și cu securitate sporită numit container. Această izolare permite rularea mai multor containere simultan pe mașina gazdă.

Docker utilizează virtualizarea la nivel de sistem de operare, astfel încât fiecare container are propriul său set de procese, rețea și stocare. Deși folosește virtualizarea, containerele Docker sunt mai eficiente decât mașinile virtuale, pentru că folosește direct kernel-ul, alături de sistemul de operare al mașinii gazdă.

Containerele Docker pot fi distribuite și partajate în timpul dezvoltării, asigurându-se astfel că fiecare membru al echipei lucrează cu aceleași versiuni ale tehnologiilor, simplificând astfel procesul de dezvoltare.

Am ales sa containerizez microserviciile cu scopul de a simplifica procesul de deployment al containerelor în cluster, Docker oferind portabilitate ridicată. Kubernetes este cea mai populară soluție de orchestrare a containerelor Docker, iar împreună facilitează gestionarea sistemelor distribuite.

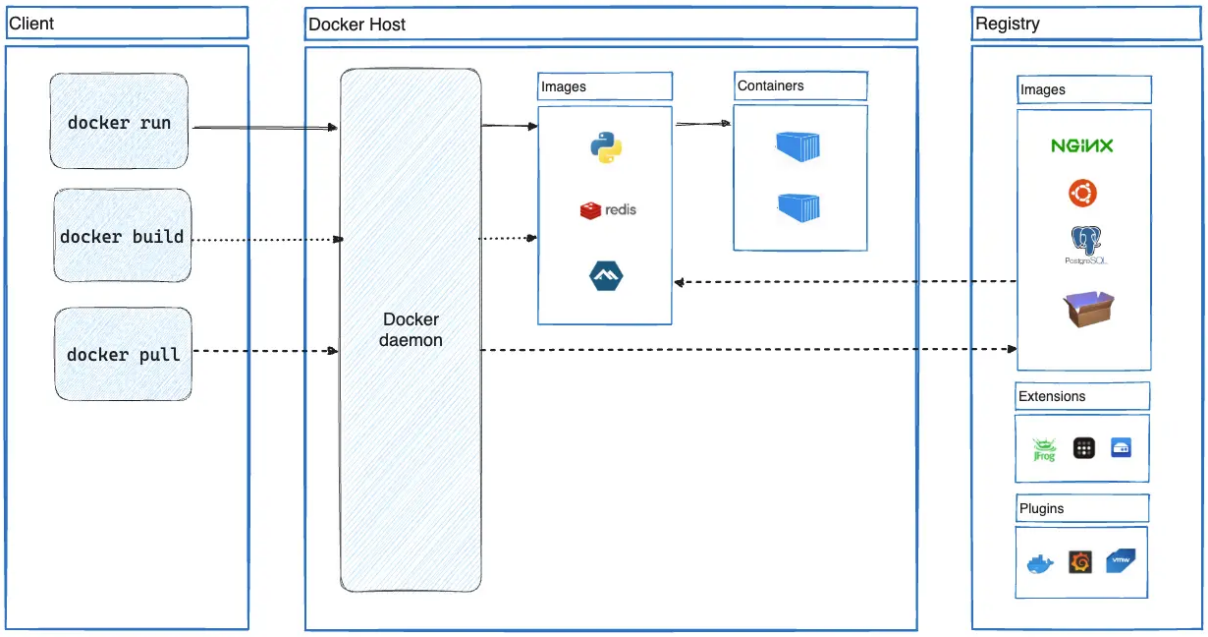

Figura 3.1: Arhitectura Docker [14]

Componentele principale ale Arhitecturii Docker:

- Docker utilizează o arhitectură client-server. Clientul Docker, ce are rolul de interfață pentru utilizatori, comunică prin intermediul REST API cu Docker daemon, care se ocupă de construirea, rularea și gestionarea containerelor Docker.
- Docker Registry este un registru în care sunt stocate imaginile Docker, de unde sunt descarcate în mod implicit de către Docker daemon.
- Când lucrăm cu Docker, folosim în mare parte imagini și containere.
- Imaginile sunt șabloane pentru a crea containere, ce pot fi bazate pe alte imagini cu specificații suplimentare. Imaginile sunt configurate și create folosind Dockerfile, în care sunt specificați pașii necesari pentru a crea și rula imaginea.
- Containerele sunt instanțe rulabile ale imaginilor, definite de acestea. În mod implicit, un container este bine izolat de alte containere și de mașina sa gazdă.
- Un container este definit de imaginea sa, precum și de orice alte opțiuni de configurare care pot fi oferite la crearea sau la pornirea acestuia. Când un container este șters, orice modificare a stării sale care nu este stocată, va fi pierdută.

3.4 Kubernetes

Kubernetes, abreviat k8s, este o platformă open-source, creată de Google, ce orchestreaza aplicațiile containerizate și automatizează procesul de deployment, scalare și gestionare a sistemelor distribuite.

Kubernetes administrează ciclului de viață al containerelor, scalabilitatea și disponibilitatea, auto-repararea, oferă funcționalități de load balancing, service discovery, gestionează configurările și datele sensibile, suportă aplicații stateless și stateful, simplifică gestionarea resurselor infrastructurii, în mare parte cloud, într-un mod eficient.

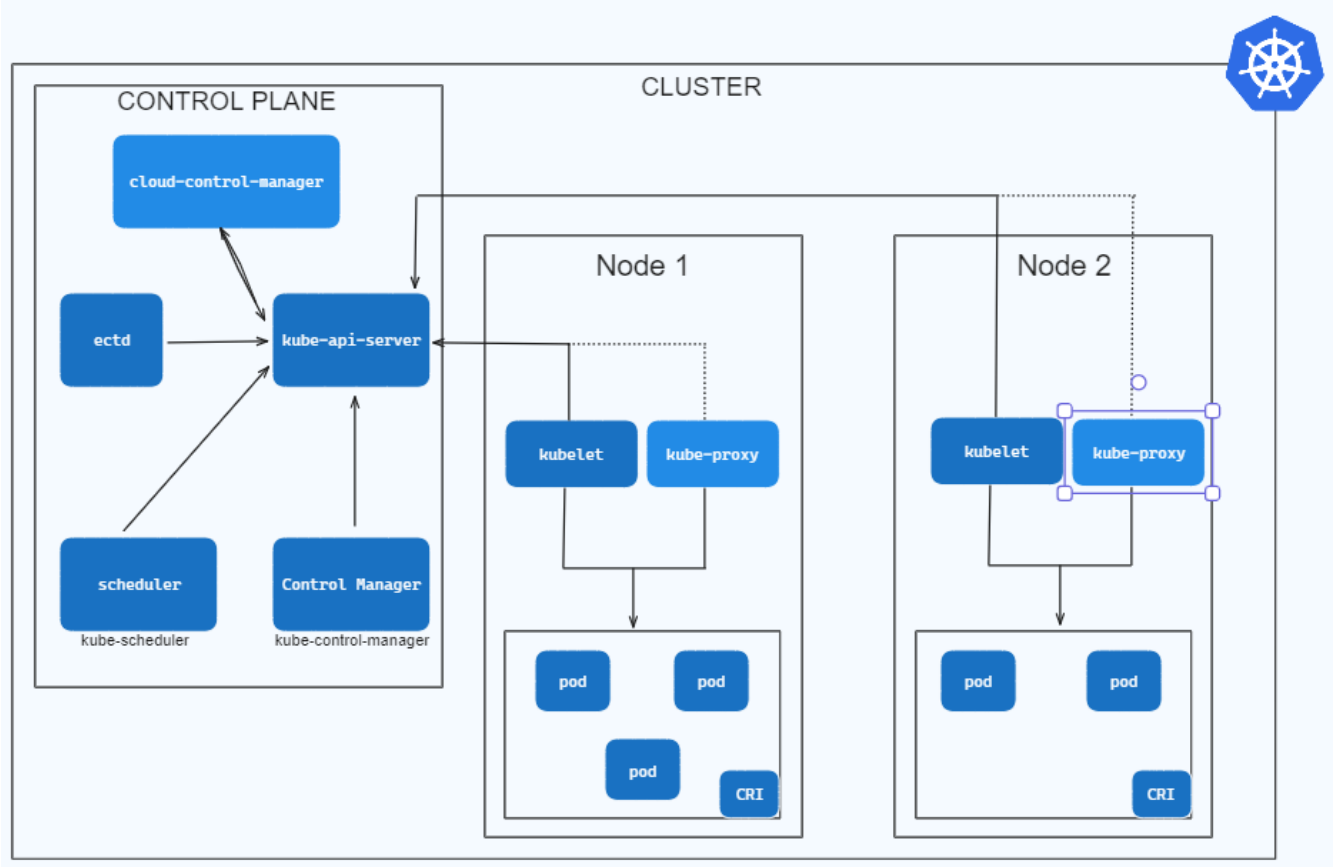

Figura 3.2: Arhitectura clusterului Kubernetes [8]

Componentele principale ale Kubernetes sunt:

1. Master Node (Control Plane): coordonează starea clusterul și gestionează nodurile de tip worker. Include componentele:

- API Server (kube-api-server) expune API-ul RESTful al clusterului, către clienți (UI, sau CLI), fiind interfața primară de comunicare între Control Plane și restul clusterului.

- etcd: bază de date distribuită de tip cheie-valoare ce stochează toate informațiile critice ale clusterului, necesare sistemului distribuit pentru rulare, cum ar fi: datele de configurare, starea curentă a clusterului și metadatele.
- Scheduler: atribuie pod-uri noi la nodurile de tip worker, în funcție de resursele disponibile pe nodurile de tip worker, asigurând că un pod nou va fi alocat unui nod potrivit din punct de vedere al resurselor clusterului.
- Controller Manager: este un daemon ce monitorizează starea curentă a clusterului prin intermediul instanțelor de tip controller, folosind serverului API și aplică schimbările necesare astfel încât starea sistemului curentă să ajungă în starea specificată în configurație (numărul de replici, rolling update, rollback, resurse alocate etc).
- Cloud-control-manager: facilitează interacțiunea clusterului cu infrastructură cloud.

2. Worker Nodes: sunt alcătuite din mașini de lucru ce rulează containerele sistemului. Din componența nodurilor fac parte:

- pod: componenta ce rulează containerul aplicației, este cea mai mică unitate ce poate fi creată și gestionată în cadrul clusterului. Pod-ul este capabil să ruleze mai multe containere, oferă stocare și o rețea partajată pentru containerele din același pod.
- kubelet: este un daemon (agent) al nodului ce gestionează pod-urile în nodul din care face parte, asigură rularea și funcționarea, conform configurației a fiecărui container din pod în parte. Acesta comunică cu Control Plane-ul, de la care primește instrucțiuni, cum ar fi ce pod-uri trebuie să ruleze pe nod.
- CRI (Container runtime): este responsabil de descărcarea imaginii container-ul din registru, gestionează ciclul de viață și resursele containerelor.
- kube-proxy: gestionează traficul de rețea dintr-un cluster, actionând ca un proxy de rețea, rulând pe fiecare nod. Direcționează traficul către pod-urile corecte, oferă load-balancing, asigurând o distribuție egală a workload-ului între pod-uri.

La nivel functional, în Kubernetes se lucrează cu următoarele componente:

1. Deployment [9]: pentru fiecare aplicație, se crează un fișier de configurație, în care se declară starea dorită pentru pod-uri și replicile acestora, iar instanțele de tip controller vor aplica modificările clusterului. Tot în acest fișier se pot specifica numărul minim și maxim de replici a pod-urilor, strategii de update, resurselor minime și maxime alocate de către Kubernetes.

   Fișierul de deployment pentru componenta Test Service:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-depl
  namespace: quiz
  labels:
    app: test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
  template:
    metadata:
      labels:
        app: test
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/path: "actuator/prometheus"
        prometheus.io/port: "5004"
    spec:
      containers:
        - image: vital7b/test-service:V1
```

```
            imagePullPolicy: Always
            name: test
            ports:
              - containerPort: 5004
            resources:
              requests:
                memory: "800Mi"
                cpu: "500m"
              limits:
                memory: "1000Mi"
                cpu: "1200m"
            env:
              - name: SPRING_PROFILES_ACTIVE
                value: k8s
            livenessProbe:
              httpGet:
                path: /actuator/health
                port: 5004
              initialDelaySeconds: 60
              periodSeconds: 10
            readinessProbe:
              httpGet:
                path: /actuator/health/readiness
                port: 5004
              initialDelaySeconds: 15
              periodSeconds: 5
          dnsPolicy: ClusterFirst
          restartPolicy: Always
          terminationGracePeriodSeconds: 30
```

2. Service: în Kubernetes un serviciu este o abstractizare care expune pod-urile în rețeaua clusterului. Pod-urile sunt efemere, fiind create și distruse în funcție de configurația din Deployment și de starea sistemului, iar adresa IP a acestora este dinamică, asociată la pornirea pod-ului. Drept consecință, componentele care apelează aceste pod-uri nu dețin informații despre starea pod-ului, dacă este în

starea running sau ready, sau dacă adresa IP mai este valabilă.

Service abstractizează pod-urile, definește reguli pentru acces la poduri și gestionează traficul, oferă o adresa IP virtuală și funcționalitatea de "service discovery" prin DNS, care facilitează atât comunicarea între pod-uri, cât și comunicarea cu exteriorul clusterului, fără a fi necesare detalii despre adresa IP a pod-urilor sau starea acestora.

```
apiVersion: v1
kind: Service
metadata:
  name: test-service
  namespace: quiz
spec:
  selector:
    app: test
  ports:
    - protocol: TCP
      name: http-traffic
      port: 6004
      targetPort: 5004
  type: LoadBalancer
```

Tipuri de servicii (Service [10]) în Kubernetes:

- ClusterIP: este tipul default de serviciu, care expune serviciul la o adresă IP internă cluster-ului, făcând serviciul accesibil doar din interiorul clusterului.
- NodePort: expune serviciul în exterior, la un port specific al nodului din cluster.
- LoadBalancer: expune serviciul in exterior, folosind un load balancer extern.
- ExternalName: mapează serviciul la un nume DNS. Atunci când serviciul este accesat, traficul este redirectionat la numele DNS specificat. Expune serviciul în exteriorul clusterului.

3. Horizontal Pod Autoscaling (HPA):

   Funcționalitatea HPA [11] scalează pe orizontală în mod automat volumul de lucru pentru a fi potrivit cererii, în funcție de metricile de performanță și configurația definită. HPA monitorizează resursele și volumul de lucru prin intermediul serverului ce expune metricile - kube-metrics-server - si ajustează numărul de replici ale pod-urilor pentru a utiliza resursele clusterului într-un mod eficient, adaptat la cerere.

   Când sarcina de lucru crește, conform metricilor stabilite în fișierul de configurație, care pot include utilizarea CPU, memoria, numărul de solicitări pe secundă, numărul de utilizatori activi, și altele, HPA va genera noi replici ale pod-urilor pentru a răspunde nevoilor crescute. Pe măsură ce sarcina scade, HPA scade numărul de replici active, până la limita minimă specificată în fișierul Deployment.

   Fișierul de deployment pentru componenta Test Service:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: test-hpa
  namespace: quiz
  labels:
    app: test
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: test-depl
  minReplicas: 1
  maxReplicas: 2
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

```
    behavior:
      scaleDown:
        stabilizationWindowSeconds: 60
        policies:
          - type: Pods
            value: 1
            periodSeconds: 30
```

3.5 Prometheus

Prometheus este un set de instrumente open-source care oferă funcționalități de monitorizare și alertare a sistemelor, fiind compatibil nativ cu tehnologia Kubernetes.

Prometheus colectează date în timp real și stochează metricile sistemului monitorizat sub formă de serii temporale - TSDB (time series database), ce permit mai apoi să fie interogate folosind limbajul PromQL(Prometheus Query Language), un limbaj de interogare puternic, oferind datele într-o manieră cât mai specifică.

PromQL permite utilizatorilor să creeze interogări complexe pentru a analiza performanța sistemului monitorizat. Câteva funcționalități oferite:

- PromQL suportă operații matematice de bază între seriile de timp, cum ar fi adunarea, scăderea, înmulțirea și împărțirea.
- PromQL oferă funții de agregare precum sum(), avg(), min(), max(), și count(), care sunt folosite pentru rezumarea datelor de monitorizare.
- Interogările pot specifica intervalul de timp pentru care să fie returnate datele, folosind selectori de timp precum [5m] pentru ultimele 5 minute sau [30m] pentru ultimele treizeci. Această funcționalitate facilitează monitorizarea sistemului pe o perioadă specifică de timp.
- PromQL permite efectuarea de sub-interogări, acestea facilitează calcule complexe și derivarea de noi metrici din datele existente.

Alertmanager din cadrul Prometheus facilitează definirea de alerte complexe în funcție de metricile și starea sistemului monitorizat, care poate trimite notificare prin diverse canale, cum ar fi email, Slack, etc, atunci când este atinsă o stare a sistemului.

Această funcționalitate poate ajuta la prevenirea prăbușirii sistemului, prin detectarea timpurie a situațiilor problematice, cum ar fi o creștere neașteptată a utilizării resurselor.

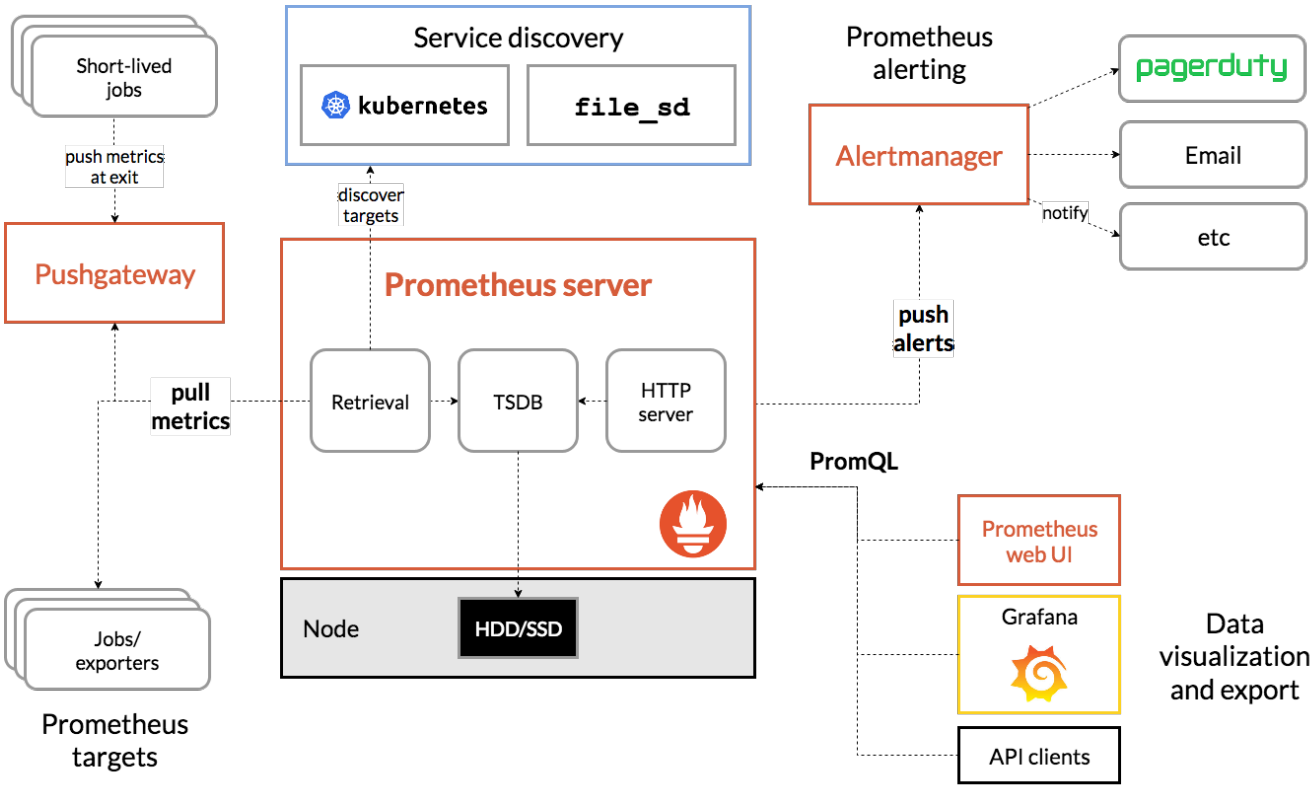

Figura 3.3: Arhitectura Prometheus[12]

3.6 Grafana

Grafana [13] este o platformă open-source pentru interogarea, explorarea și vizualizarea datelor, prin intermediul dashboard-urilor grafice interactive și personalizate. Această platformă permite prezentarea datelor, stocate într-o bază de date de serii temporale (TSDB), într-un mod grafic, folosind grafice liniare, bare și diagrame complexe, facilitând astfel procesul de observabilitate a sistemului. Utilizatorii pot colabora în crearea de dashboard-uri, pe care le pot partaja pe pagina web a platformei.

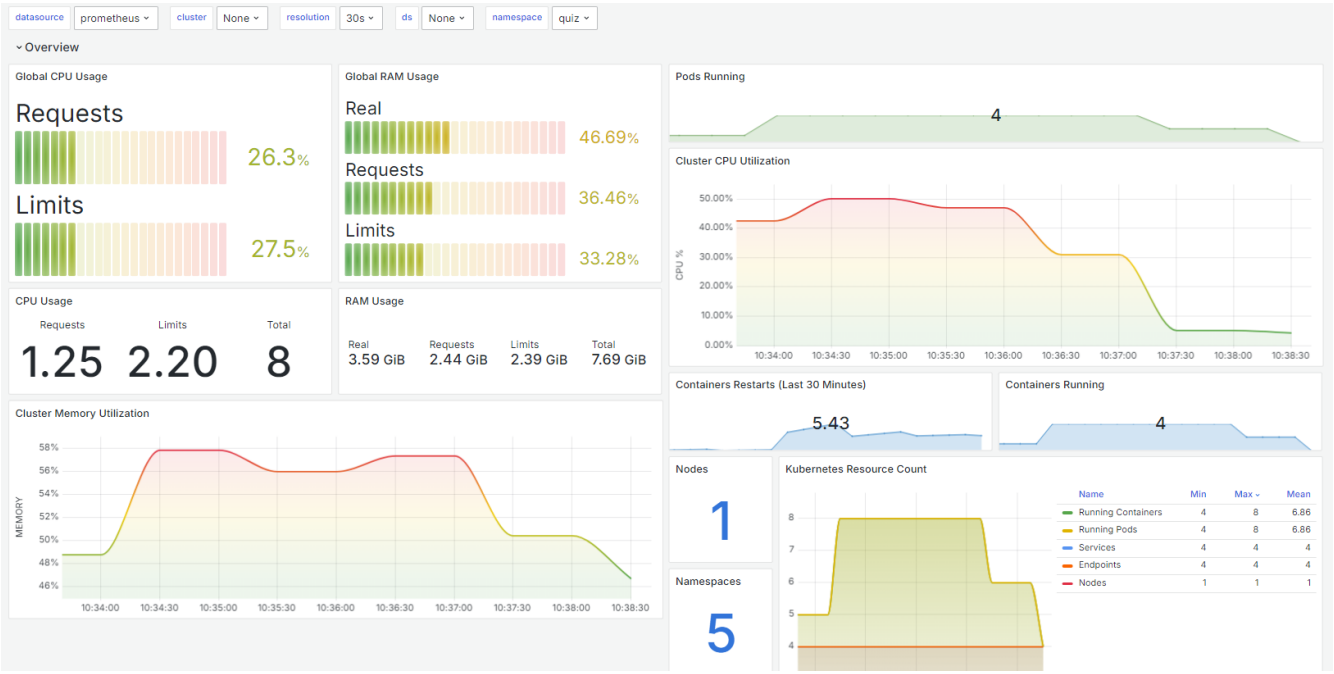

Figura 3.4: Dashboard-ul Grafana pentru aplicația eQuiz

În continuare voi descrie câteva interogări PromQL folosite în acest dashboard:

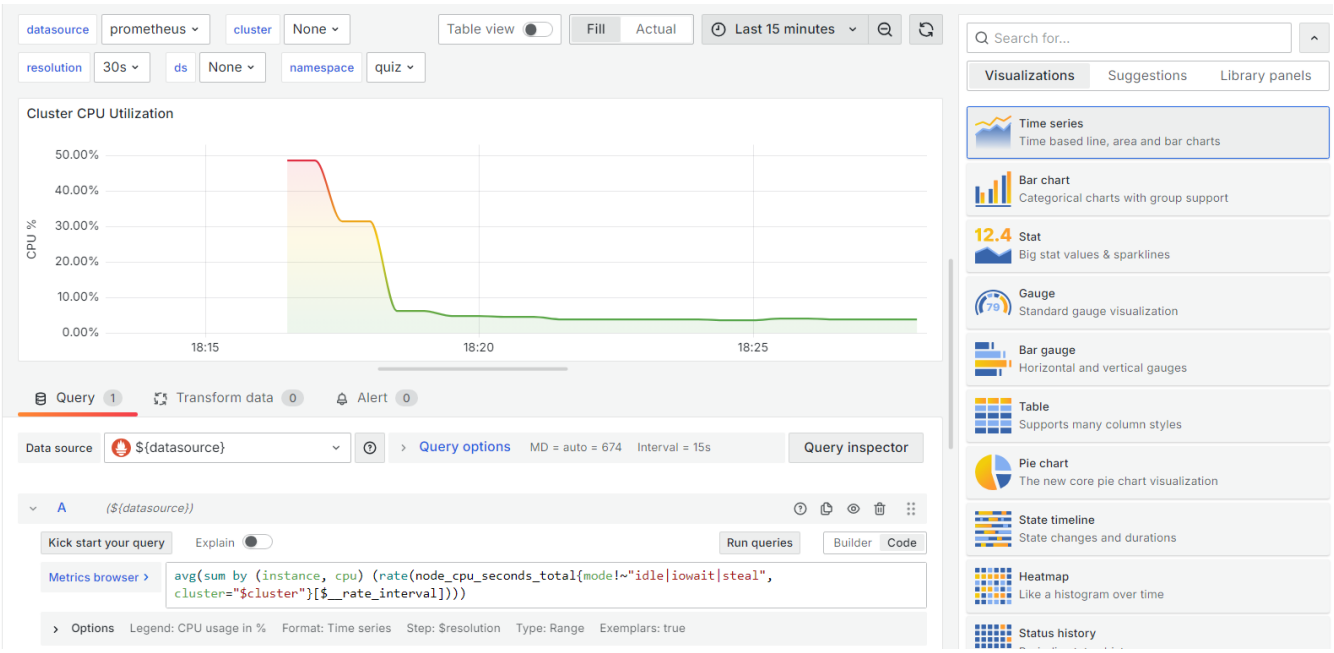

Figura 3.5: Utilizarea medie a CPU-ului

```
avg(sum by (instance, cpu) (rate(node_cpu_seconds_total{mode!~"
    idle|iowait|steal", cluster="$cluster"}[$__rate_interval])))
```

Această interogare calculează utilizarea medie a procesorului alocat clusterului pe toate instanțele, excluzând modurile idle, iowait și steal, pe un interval de timp specificat. Datele sunt grupate după instanță și procesor și afișate într-un grafic de tip ”Time series” în Grafana.

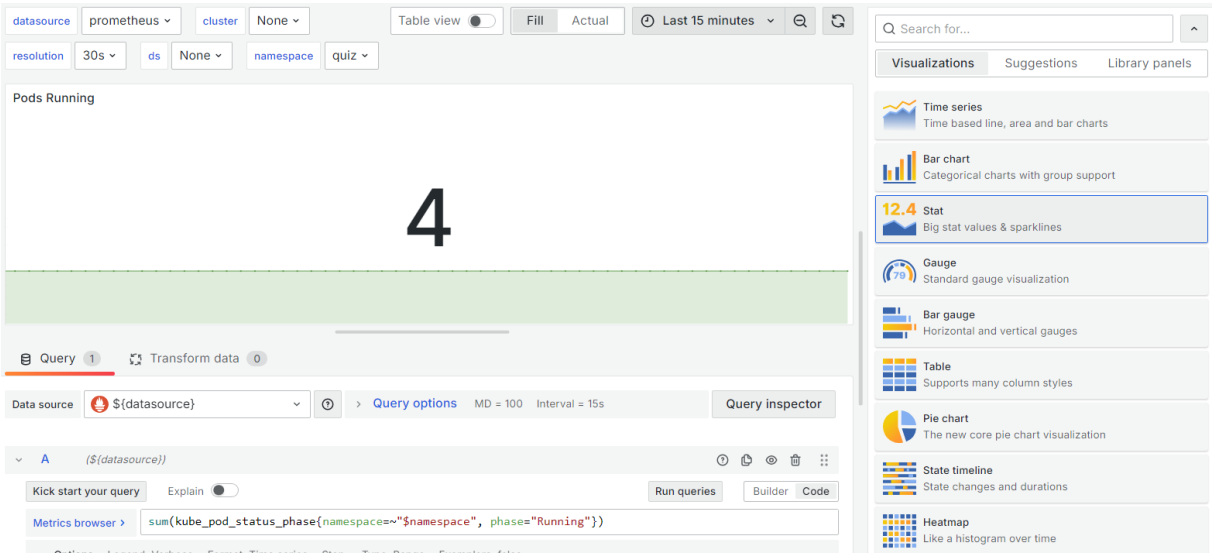

Figura 3.6: Numărul de pod-uri active din cluster

```
sum(kube_pod_status_phase{namespace=~"$namespace", phase="Running
    "})
```

Această interogare calculează numărul total de pod-uri aflate în starea ”Running” în namespace-ul selectat.

Concluzii

Migrarea aplicației eQuiz, de la o arhitectură monolitică la o arhitectură bazată pe microservicii, facilitează adoptarea aplicației de către alte instituții educaționale, oferind astfel disponibilitate chiar și atunci când sistemul se va confrunta cu o solicitare mare, cu număr mare de utilizatori în mod concurent, fără a compromite performața acestuia. Arhitectura implementată a făcut posibilă scalarea și flexibilitatea sistemului, asigurând utilizarea resursele fizice într-un mod eficient.

Utilizarea tehnologiilor moderne precum Spring Boot, Docker și Kubernetes mi-a permis să contruiesc un sistem robust, ușor de întreținut și adaptabil la cerințele în continuă schimbare din cadrul mediului educațional.

Prin integrarea serviciilor de colectare și monitorizare a metricilor, cu ajutorul tehnologiilor Prometheus și Grafana, am asigurat o vizibilitate sporită asupra performanței sistemului și am facilitat observabilitatea sistemului.

Pentru a extinde și îmbunătăți funcționalitățile aplicației eQuiz, pot fi considerate următoarele direcții:

1. Analiza predictivă: Utilizarea datelor colectate din teste, pentru a oferi predicții despre tendințele de performanță ale studenților, ajutând instructorii să identifice punctele slabe ale fiecărui student în parte, astfel încât să îi poată îndruma din timp în direcțiile potrivite.

2. Sporirea interactivității testelor: Adăugarea suportului prin integrarea de conținut multimedia, inclusiv video și audio, prin întrebări care permit răspunsuri audio de la utilizatori, precum și opțiuni pentru răspunsuri în format text la întrebările bazate pe conținut video și audio.

3. Securitate îmbunătățită: Implementarea soluțiilor avansate de securitate pentru a proteja integritatea datelor și confidențialitatea utilizatorilor.

Bibliografie

[1] Martin Fowler, *Monolith First*, 2015.
Available at: https://martinfowler.com/bliki/MonolithFirst.html

[2] Chris Richardson, *Microservice Architecture pattern*
-https://microservices.io/patterns/microservices.html

[3] Martin Fowler, *Microservices Guide*, 2019
-https://martinfowler.com/microservices/

[4] Mehmet Ozkaya, *The Database-per-Service Pattern*, 2021
-https://medium.com/design-microservices-architecture-with-patterns/the-database-per-service-pattern-9d511b882425

[5] Mehmet Ozkaya, *Service Aggregator Pattern*, 2021
-https://medium.com/design-microservices-architecture-with-patterns/service-aggregator-pattern-e87561a47ac6

[6] Mark Richards, *Software Architecture Patterns*, O'Reilly Media, Inc., 2015
-https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

[7] Joseph Ingeno, *Software Architect's Handbook*, Packt Publishing, 2018

[8] Kubernetes documentation, *Architecture*
-https://kubernetes.io/docs/concepts/architecture/

[9] Kubernetes documentation, *Deployment*
-https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[10] Kubernetes documentation, *Service*
-https://kubernetes.io/docs/concepts/services-networking/service/

[11] Kubernetes documentation, *HPA*
-https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[12] Prometheus docs, *Architecture*
-https://prometheus.io/docs/introduction/overview/

[13] Grafana documentation, *Overview*
-https://grafana.com/docs/grafana/latest/

[14] Docker documentation, *Architecture*
-https://docs.docker.com/guides/docker-overview/