UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și Tehnologia Informației
SPECIALIZAREA: Tehnologia Informației

LUCRARE DE DIPLOMĂ

Coordonator științific:
Șef. lucr. dr. ing. Cristian-Nicolae BUȚINCU

Absolvent:
Cosmin-Sergiu MILICA

Iași, 2024

UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și Tehnologia Informației
SPECIALIZAREA: Tehnologia Informației

Optimizarea Livrării Conținutului și Securitatea Site-urilor Web

LUCRARE DE DIPLOMĂ

Coordonator științific:
Șef. lucr. dr. ing. Cristian-Nicolae BUȚINCU

Absolvent:
Cosmin-Sergiu MILICA

Iași, 2024

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII PROIECTULUI DE DIPLOMĂ

Subsemnatul Milica [redacted] Cosmin-Sergiu,

legitimat cu [redacted] seria [redacted] nr. [redacted], CNP [redacted]

autorul lucrării

Optimizarea Livrării Conținutului și Securitatea Site-urilor Web

elaborată în vederea susținerii examenului de finalizare a studiilor de licență, programul de studii Licență-Tehnologia Informației organizat de către Facultatea de Automatică şi Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi" din Iaşi, sesiunea Iulie 2024 a anului universitar 2023-2024, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi" din Iaşi (Manualul Procedurilor, UTI.POM.02 - Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) şi a convențiilor internaționale privind drepturile de autor.

Data
30.06.2024

Semnătura
[redacted]

Cuprins

Optimizarea Livrării Conținutului și Securitatea Site-urilor Web

Cosmin-Sergiu MILICA

Rezumat

Lucrarea prezintă dezvoltarea unei platforme personalizate de Content Delivery Network (CDN), concepută pentru a îmbunătăți performanța și securitatea site-urilor web printr-o arhitectură distribuită și scalabilă. Această soluție are drept scop principal accelerarea livrării conținutului online și protejarea împotriva atacurilor cibernetice de tip DDoS (Distributed Denial of Service), asigurând disponibilitate continuă și performanțe optime pentru utilizatori. CDN reprezintă o tehnologie esențială pentru accelerarea livrării conținutului online prin dirijarea utilizatorilor către serverul cel mai apropiat, oferindu-le acestora o experiență rapidă în mediul online. Motivația acestui proiect derivă din creșterea exponențială a consumului de conținut digital, accelerată de pandemia COVID-19, care a subliniat nevoia de soluții eficiente pentru livrarea conținutului.

Soluția propusă se bazează pe dezvoltarea de servere edge utilizând Python și framework-ul FastAPI, care rulează în cloud-ul AWS. Aceste servere sunt responsabile pentru stocarea în cache atât la nivel local, cât și global, reducând semnificativ timpul de răspuns și lățimea de bandă utilizată. Informațiile sunt stocate în cache pentru a servi cererilor ulterioare, iar managementul cache-ului se realizează pe baza unui timp setat de utilizator sau prin utilizarea algoritmului Least Recently Used (LRU).

Platforma include servicii DNS integrate pentru direcționarea eficientă a cererilor către cele mai apropiate și disponibile servere edge, asigurând o distribuție echilibrată a traficului. Utilizatorii care vor accesa site-urile web din cadrul platformei se vor bucura de un timp mic de răspuns, iar clienții platformei vor reduce traficul pe serverele lor și vor diminua lățimea de bandă.

Un aspect esențial al acestei platforme este implementarea unui filtru DDoS (Distributed Denial of Service) pentru a proteja serverele edge și resursele clienților de atacurile cibernetice. Filtrul DDoS detectează și blochează traficul malițios, asigurând disponibilitatea continuă a serviciilor și minimizând impactul potențialelor atacuri.

Aplicația de control permite clienților să gestioneze resursele, oferindu-le posibilitatea de a configura parametrii de cache și de a alege modul de livrare a conținutului. Această flexibilitate este esențială pentru adaptarea la nevoile specifice ale fiecărui client, permițându-le să optimizeze performanța site-urilor lor.

Concluziile obținute evidențiază eficiența arhitecturii propuse în îmbunătățirea timpului de răspuns și a securității site-urilor web, demonstrând beneficiile utilizării unei platforme CDN personalizate pentru gestionarea și distribuirea conținutului digital.

Introducere

În ultimii ani, se observă o creștere semnificativă a consumului de conținut online, un fenomen accelerat de pandemia de COVID-19. Era digitală a atins niveluri neașteptate, determinând migrarea majorității activităților în mediul digital. De la afaceri, la informații, aproape totul este acum accesibil prin intermediul site-urilor web. Această creștere a cererii a generat presiuni semnificative asupra infrastructurii operatorilor de servere web de dimensiuni mici, evidențiind astfel necesitatea adoptării unor soluții eficiente pentru livrarea conținutului. Viteza de încărcare și disponibilitatea nu sunt doar așteptări ale utilizatorilor, ci și componente esențiale care pot determina alegerea clienților.

Odată cu creșterea rapidă a mediului online, cererea pentru comunicare și afaceri digitale a crescut exponențial. Acest lucru a accentuat importanța vitală a vitezei de răspuns, a preciziei și a disponibilității conținutului pentru a menține clienții implicați pe platforme. În acest context, tehnologiile de rețea pentru livrarea conținutului (CDN) au devenit indispensabile pentru optimizarea performanței și asigurarea unei experiențe excelente pentru utilizatori. Autorii studiului [25] subliniază faptul că serverele individuale întâmpină dificultăți în gestionarea traficului crescut, iar utilizarea CDN-urilor poate rezolva aceste probleme prin reducerea costurilor de replicare și distribuție a conținutului.

CDN-urile acționează ca o rețea globală de servere, distribuite strategic pe întregul glob, pentru a asigura livrarea continuă și eficientă a conținutului digital către utilizatori. Prin poziționarea serverelor în locații geografice diverse, CDN-urile reduc distanța fizică dintre utilizatori și conținut, îmbunătățind semnificativ timpul de încărcare al paginilor web și al altor resurse digitale. Acest lucru nu numai că îmbunătățește experiența utilizatorului, ci oferă și scalabilitate și securitate îmbunătățită. Mai mult decât atât, acestea oferă protecție împotriva atacurilor cibernetice de tip DDoS (Distributed Denial of Service), asigurând astfel securitatea și disponibilitatea continuă a serviciilor web în mediul online.

Această tehnologie are o istorie îndelungată, începând de la sfârșitul anilor 1990. Inițial, conceptul a fost conceput pentru a reduce congestia traficului pe internet. Cu trecerea timpului, CDN-urile au evoluat într-un mod spectaculos, devenind capabile să gestioneze conținut dinamic, streaming video și alte sarcini complexe. Companii de talie mondială din domeniul tehnologic, cum ar fi Google [16] și Cloudflare [15], oferă soluții de încredere. Progresele tehnologice în domeniu au avut un impact semnificativ în diverse sectoare de activitate. Rezultatele remarcabile s-au observat în special în domeniile comerțului electronic, media și divertisment, dar și în educație.

Utilizarea CDN-urilor a devenit importantă pentru numeroase companii de talie mondială, cum ar fi Netflix, Amazon și Facebook, care depind de livrarea eficientă a conținutului către milioane de utilizatori din întreaga lume. CDN-urile nu doar că reduc latența și îmbunătățesc viteza de încărcare, dar asigură și redundanța și securitatea necesare pentru a menține performanța serviciilor online. În acest context, dezvoltarea unei soluții CDN personalizate poate oferi un control total asupra infrastructurii și poate răspunde mai bine nevoilor specifice ale utilizatorilor.

Așa cum au observat și autorii [25], furnizorii comerciali creează arhitecturi costisitoare pe tot globul. Costul financiar uriaș implicat în crearea și funcționarea arhitecturii comerciale obligă furnizorii să perceapă o remunerație ridicată de la clienții lor. Autorii propun o arhitectură numită Distributed Content Delivery Network(*DCDN*), aceasta fiind o arhitectură hibridă care integrează unele dintre caracteristicile majore ale CDN convențional client/server și un CDN academic peer-to-peer.

În contextul actual al mediului online, CDN-urile joacă un rol semnificativ în garantarea furnizării conținutului către utilizatori. Acestea livrau aproximativ 72% din traficul de internet în 2022, iar valoarea pieței CDN este de așteptat să crească de la 11,76 miliarde de dolari în

2019 la 49,61 miliarde de dolari în 2025 [27]. Cu toate acestea, rețelele sunt supuse la diverse amenințări de securitate care pot afecta atât performanța CDN-ului, cât și experiența utilizatorului final. Prin urmare, implementarea mecanismelor de protecție este esențială pentru a asigura și proteja disponibilitatea conținutului pe internet.

Lucrarea își propune să dezvolte o soluție CDN personalizată, bazată pe o arhitectură distribuită și scalabilă pentru livrarea eficientă a conținutului. Prin implementarea serverelor edge, a cache-ului local și global, se urmărește reducerea timpului de răspuns și îmbunătățirea performanței site-urilor de origine. Serviciile DNS integrate vor direcționa cererile către cele mai potrivite servere, asigurând distribuția echilibrată a traficului și optimizarea resurselor. În plus, platforma va include o interfață de control destinată clienților, oferindu-le acestora posibilitatea de a-și gestiona resursele în mod autonom, ajustând parametrii de cache și alte setări personalizate conform nevoilor lor specifice. Scopul acestei platforme este de a crea un mediu flexibil și eficient, capabil să gestioneze un număr mare de utilizatori simultan, asigurând în același timp securitatea și disponibilitatea continuă a conținutului.

Aplicația (*Content Delivery Network*) își propune să ofere clienților o metodă rapidă și eficientă de livrare a conținutului găzduit de site-urile lor către utilizatorii finali, prin intermediul unei platforme robuste. Implementarea acestei platforme aduce multiple beneficii, inclusiv reducerea timpului de răspuns pentru utilizatorii finali și asigurarea securității prin transportul datelor prin protocolul *HTTPS* între clienți și servere.

Astfel, clienții se vor bucura de o scădere a lățimii de bandă utilizate, dar și de o creștere a capacității de a gestiona un număr mai mare de utilizatori simultan pe site-ul lor. Pe lângă serviciile CDN, se are în vedere implementarea de servicii DNS integrate pentru a direcționa eficient cererile către serverele edge. Toată interacțiunea cu serviciile de bază va fi gestionată prin intermediul unei platforme online.

Când utilizatorii accesează un domeniu de pe platformă, ei sunt redirecționați către serverul DNS. Aici, un server edge este asignat pe baza locației lor sau, dacă acest lucru nu este posibil, cererile sunt distribuite într-o manieră aleatoare. Odată ajunși la serverul edge, se verifică dacă resursele dorite sunt disponibile în cache-ul local al acestuia sau în cel global al întregii platforme. Dacă informațiile nu sunt găsite în cache, serverul edge face un apel către serverul de origine pentru a obține resursele solicitate. După obținerea acestora, resursele sunt salvate în cache-ul local al serverului edge pentru o livrare rapidă și eficientă în viitor.

Un aspect esențial al acestei platforme este implementarea unui filtru DDoS (Distributed Denial of Service) pentru a proteja serverele edge și resursele clienților de atacurile cibernetice. Filtrul DDoS detectează și blochează traficul malițios, asigurând disponibilitatea continuă a serviciilor și minimizând impactul potențialelor atacuri.

Această abordare asigură o experiență bună a utilizatorului, dar și îmbunătățește performanța și securitatea serviciilor. Prin distribuirea inteligentă a traficului și utilizarea eficientă a cache-ului, timpul de răspuns al aplicației este redus, ceea ce duce la o experiență mai fluidă și mai rapidă pentru utilizatori.

Aplicația de control permite crearea de conturi, iar ulterior este posibilă înregistrarea site-urilor web în cadrul platformei. Scopul acestei aplicații este să ofere clienților posibilitatea de a-și gestiona propriile resurse. Clientul poate opta pentru ca utilizatorii site-urilor să ocolească memoria cache, obținând resursele direct de la serverul de origine.

Rezultatele așteptate includ o reducere semnificativă a timpului de încărcare a paginilor web, îmbunătățirea experienței utilizatorului și asigurarea securității datelor transmise. Platforma va permite, de asemenea, scalabilitate și flexibilitate, permițând gestionarea unui număr mare de utilizatori simultan.

Capitolul 1. Fundamentarea teoretică și documentarea bibliografică

Datorită creșterii semnificative a conținutului online și a traficului pe internet, serverele web individuale întâmpină dificultăți în a satisface nevoile utilizatorilor finali, ceea ce adesea generează disconfort. Autorii articolului [25] propun ca soluție conceptul de Content Delivery Network (CDN). Acest concept permite mai multor furnizori de conținut să încarce resurse web în aceeași rețea de servere, reducând astfel costurile asociate cu replicarea și distribuția conținutului.

În contextul evoluției rapide a tehnologiilor informaționale și a extinderii internetului, rețelele de livrare a conținutului (CDN) au devenit esențiale pentru a asigura accesul rapid și eficient la conținut digital pentru utilizatorii din mediul online. CDN-urile îmbunătățesc considerabil performanța și disponibilitatea site-urilor web, a aplicațiilor și a serviciilor online, prin distribuirea conținutului pe servere amplasate strategic în diverse locații geografice. Această abordare permite utilizatorilor să acceseze informațiile dorite mai rapid, reducând timpii de încărcare și îmbunătățind experiența generală de navigare. Acestea sunt utilizate pe scară largă de numeroase companii, inclusiv Netflix, Amazon și Facebook pentru a livra conținut video, imagini, și alte tipuri de date către milioane de utilizatori din întreaga lume.

Tema propusă este „Optimizarea livrării conținutului și securitatea site-urilor web” prin dezvoltarea unei rețele de livrare a conținutului (CDN) pentru îmbunătățirea performanței și disponibilității serviciilor online. Obiectivul principal este de a crea o platformă CDN proprie, adaptată nevoilor specifice ale utilizatorilor și capabilă să gestioneze eficient distribuția de conținut. Justificarea abordării constă în necesitatea de a dezvolta o soluție customizată care să ofere un control total asupra infrastructurii și să optimizeze livrarea de conținut.

1.1. Aplicații CDN

CDN (***C**ontent **D**elivery **N**etwork*) este un ansamblu de servere distribuite geografic care stochează în cache conținutul aproape de utilizatorii finali. Scopul său principal este de a furniza disponibilitate și performanță ridicate prin distribuirea spațială a serviciului în raport cu utilizatorii finali [18].

Rețelele de livrarea a conținutului stochează resurse statice, cum ar fi fișiere JavaScript, CSS, imagini și videoclipuri, în sistemul de caching, aproape de utilizatorii finali. Prin această configurare, un CDN facilitează livrarea rapidă a acestor resurse digitale, optimizând experiența online și reducând timpul de încărcare al paginilor web. Acestea nu înlocuiesc serverele de origine, ci funcționează complementar acestora, facilitând livrarea conținutului de la marginea rețele, ceea ce contribuie la îmbunătățirea performanței generale a site-ului. Astfel, utilizatorii finali beneficiază de o experiență online optimizată, cu timpi de încărcare mai rapizi și o disponibilitate mai mare a resurselor necesare.

Multe companii oferă servicii CDN, iar printre cele mai importante se numără Cloudflare și Google. Aceste companii sunt lideri în domeniu, asigurând livrarea rapidă și eficientă a conținutului digital către utilizatori din întreaga lume.

1.1.1. CloudFlare CDN

CloudFlare este una dintre platformele CDN lider pe piață, recunoscută pentru capacitatea sa de a îmbunătăți securitatea și performanța site-urilor web prin rețeaua sa extinsă de servere distribuite global. Această platformă nu doar că stochează conținutul static, precum JavaScript, CSS, imagini și videoclipuri, în locații geografice apropiate de utilizatorii finali, dar oferă și servicii suplimentare de optimizare și securitate.

Conform documentației oficiale, platforma memorează cache-ul bazându-se pe extensiile fișierelor, nu pe tipul MIME, și nu include în mod implicit conținutul HTML în cache. Aceasta

suportă o gamă variată de resurse statice precum JPG, PNG, CSS, MP3 etc [15].

Cloudflare CDN oferă și opțiuni avansate pentru personalizarea gestionării cache-ului [15]:

- Configurarea comportamentului de stocare în cache pentru URL-uri specifice prin stabilirea unor reguli personalizate.
- Modificarea strategiilor de stocare în cache folosind Cloudflare Workers.
- Ajustarea nivelului de cache, a timpului de viață al datelor în cache (TTL) și alte aspecte prin aplicația Cloudflare Caching.

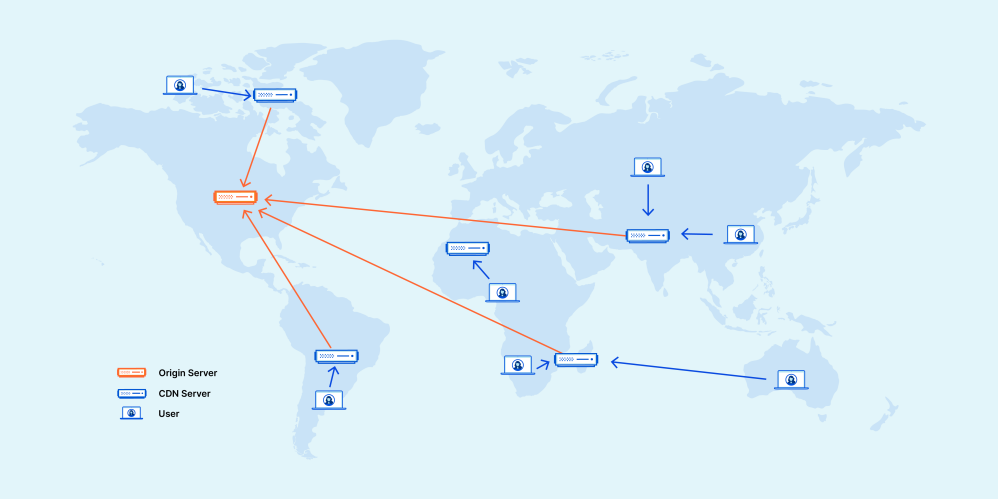

Figura 1.1. CloudFlare CDN[1]

Figura 1.1 ilustrează modul în care funcționează platforma CDN CloudFlare. În momentul în care o resursă este solicitată de pe un site web, Cloudflare o stochează în cache pentru a preveni accesul repetat la serverul de origine la solicitările ulterioare, ceea ce reduce latența și accelerează livrarea conținutului. Dacă o resursă din cache nu este accesată pentru o perioadă, aceasta va fi în cele din urmă eliminată pentru a face loc resurselor mai noi și mai frecvent solicitate. Acest proces este cunoscut sub numele de evacuare și face parte integrantă din gestionarea cache-ului. Pentru a elibera spațiu atunci când cache-ul este plin, Cloudflare utilizează un algoritm denumit Least Recently Used (LRU), care selectează obiectele cele mai puțin recent utilizate pentru a fi înlocuite. Durata de păstrare a unui obiect în cache depinde de frecvența cu care este accesat și de capacitatea totală a cache-ului Cloudflare, și nu poate fi configurată direct [15].

Fereastra temporală în care un obiect este considerat adecvat pentru utilizare în cache este determinată de prospețimea sa, cunoscută și sub denumirea de Time to Live (TTL). De exemplu, dacă un obiect are un TTL de cinci minute, aceasta indică faptul că, de la momentul primirii inițiale în cache, acesta poate fi folosit pentru următoarele cinci minute fără a fi necesară revalidarea lui de la serverul de origine. După expirarea celor cinci minute, în cazul în care Cloudflare primește o nouă solicitare pentru acel obiect, va trebui să verifice din nou cu serverul de origine pentru a confirma dacă obiectul este încă valid. Dacă un obiect din cache nu mai este proaspăt și Cloudflare primește o solicitare pentru acesta, va solicita serverului de origine să revalideze obiectul. Serverul de origine poate fie să trimită o versiune actualizată a obiectului, care va înlocui cea veche din cache, fie să confirme că versiunea din cache este încă validă, actualizând astfel TTL-ul. Această revalidare este necesară ori de câte ori perioada de păstrare a unui obiect depășește perioada de prospețime [15].

[1]What is a content delivery network (CDN)? | How do CDNs work? https://www.cloudflare.com/learning/cdn/what-is-a-cdn/

Cloudflare oferă un serviciu solid de CDN, punând la dispoziția clienților o multitudine de facilități care asigură un mediu optim și o creștere semnificativă a performanței site-urilor web. Funcționalități avansate permit companiilor să optimizeze livrarea conținutului, să reducă latența și să îmbunătățească experiența utilizatorilor finali, contribuind astfel la un site mai rapid și mai sigur.

1.1.2. Google Cloud CDN

Google Cloud CDN folosește punctele de prezență limită distribuite global ale Google pentru a stoca în cache conținutul extern cu încărcare echilibrată HTTP(S) aproape de utilizatori. Memorarea în cache a conținutului la marginile rețelei Google asigură livrarea mai rapidă a conținutului către utilizatori, reducând în același timp costurile de servire [16].

Google oferă două soluții de CDN distincte: Cloud CDN și Media CDN. Cloud CDN este conceput pentru a accelera aplicațiile web, utilizând rețeaua globală extinsă a Google pentru a reduce latența și a îmbunătăți accesul la conținut. Pe de altă parte, Media CDN se specializează în distribuția de conținut media, folosind infrastructura robustă a YouTube pentru a optimiza livrarea de fluxuri video, atât pentru conținutul video on-demand(VoD) cât și pentru transmisiunile live, precum și pentru descărcările de fișiere de mari dimensiuni.

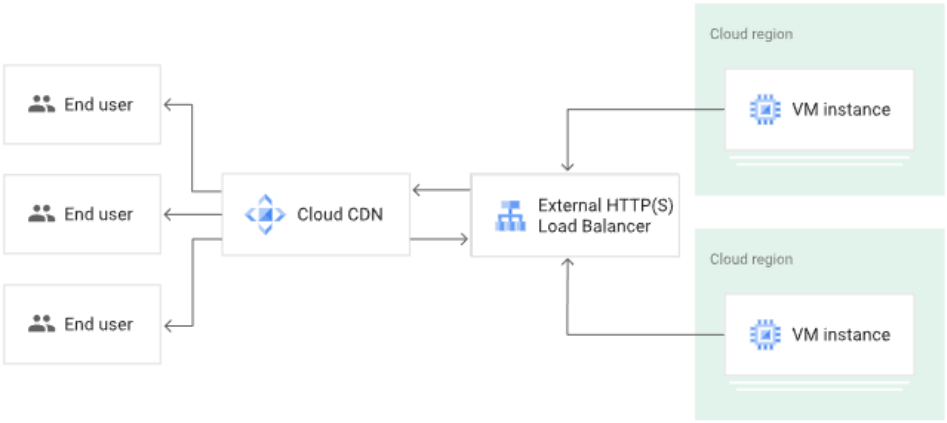

Figura 1.2. Fluxul de răspunsuri de la serverele de origine prin Cloud CDN către clienți.[2]

Figura 1.2 prezintă modul în care sunt procesate cererile în cadrul platformei Google, împreună cu componentele sale cheie. Platforma funcționează cu aplicații de load balancing pentru a furniza conținut utilizatorilor. Aceste load balancere furnizează adrese IP și porturile către serverele de backend care răspund solicitărilor.

Google Front End (GFE) reprezintă componenta principală care gestionează și distribuie cererile utilizatorilor către serverele backend. GFE termină sesiunile TCP și SSL, inspectează cererile HTTP pentru a determina destinația corectă și recriptează datele înainte de a le trimite mai departe. Aceasta asigură o distribuție eficientă a traficului și contribuie la performanța și securitatea rețelei CDN a Google, optimizând livrarea conținutului către utilizatori [17]. GFE optimizează traseul solicitărilor către cel mai apropiat server de cache, reducând astfel latența și îmbunătățind viteza de livrare a conținutului.

Google Cloud CDN se folosește de adresele IP Anycast pentru a ghida utilizatorii spre cel mai apropiat server edge, garantând cea mai scurtă cale posibilă pentru date și minimizând întârzierile de rețea.

Când un utilizator accesează conținut prin intermediul unui Load Balancer extern, solicitarea ajunge la un GFE, localizat la periferia rețelei Google și cât mai aproape de utilizator. Un cache

[2]Cloud CDN overview https://cloud.google.com/cdn/docs/overview/

este un grup de servere care stochează și gestionează conținut, astfel încât cererile viitoare pentru acel conținut să poată fi servite mai rapid. Conținutul stocat în cache este o copie a conținutului stocat pe serverele de origine. Dacă GFE caută în memoria cache Cloud CDN și găsește un răspuns la cererea utilizatorului, acesta trimite răspunsul din cache. Utilizarea memoriei cache implică doua situații: cache hit și cache miss.

Cache Hit se referă la cazul în care GFE identifică conținutul solicitat în cache, răspunde direct utilizatorului, scurtând timpul de răspuns și ocolind necesitatea de a interoga serverul de origine. Iar Cache Miss este atunci când conținutul solicitat nu este disponibil în cache, GFE redirecționează cererea către Load Balancer, care ulterior o trimite către serverul de origine. Răspunsul primit este apoi stocat în cache pentru utilizări ulterioare.

Fiecare cache are o capacitate limitată de stocare. Totuși, Cloud CDN continuă să adauge conținut în cache chiar și când acestea sunt pline. Pentru a face loc conținutului nou într-un cache plin, sistemul elimină mai întâi alte elemente, un proces cunoscut sub denumirea de evacuare. Cache-urile sunt frecvent la capacitate maximă, astfel evacuarea conținutului vechi este o practică obișnuită. În general, conținutul eliminat este cel care nu a fost accesat recent, indiferent de timpul său de expirare. Evacuarea nu este influențată de setările de expirare. Conținutul considerat nepopular este acela care nu a fost accesat de ceva vreme. Acesta informează memoria cache să nu difuzeze conținut învechit, chiar dacă conținutul nu a fost eliminat din memoria cache.

1.2. Sisteme de caching

Sistemul de cache joacă un rol esențial în eficientizarea livrării conținutului web prin intermediul unei platforme CDN. Tehnologia de caching web este utilizată pe scară largă pentru a îmbunătăți performanța infrastructurii web și pentru a reduce latențele percepute de utilizatori în rețea. Caching-ul proxy este o tehnică principală de caching web care încearcă să servească cererile web ale utilizatorilor dintr-o rețea de proximități situate între utilizatorii finali și serverele web care găzduiesc copiile originale ale obiectelor solicitate.

În funcție de locația cache-urilor, sistemele de caching web pot fi clasificate în trei tipuri principale: cache-uri de browser, cache-uri proxy și cache-uri surrogate [24].

Cache-urile de browser sunt localizate în cadrul programelor de navigare ale utilizatorilor și servesc pentru a reduce timpul de încărcare a paginilor web vizitate frecvent. Atunci când un utilizator accesează un site web, anumite resurse, cum ar fi imagini, fișiere CSS și scripturi JavaScript, sunt stocate temporar pe dispozitivul local al utilizatorului. La următoarea vizită pe același site, browser-ul urmează să încarce aceste resurse din cache-ul local în loc să le descarce din nou de pe server, reducând semnificativ timpul de încărcare a paginilor web și economisește lățimea de bandă.

Cache-urile proxy sunt situate între site-urile clientului utilizatorului final și serverele web de origine, de obicei mai aproape de clienți decât de servere [24]. Acestea funcționează ca intermediari, stocând copii ale resurselor solicitate frecvent pentru a reduce timpul de răspuns și pentru a îmbunătăți performanța rețelei. Atunci când un client solicită o resursă, cache-ul proxy verifică dacă resursa este deja stocată local și, dacă este disponibilă, o servește direct din cache. Dacă resursa nu este găsită în cache, cererea este transmisă către serverul de origine, iar resursa este apoi stocată în cache-ul proxy pentru cereri viitoare.

Cache-urile surrogate sunt situate în apropierea serverelor web și sunt deținute și operate de furnizorii de conținut. Cache-urile surrogate ajută la livrarea rapidă a conținutului prin stocarea copiilor obiectelor web în apropierea utilizatorilor finali [24]. Prin cache-urile surrogate, furnizorii reduc încărcarea pe serverele de origine și îmbunătățesc experiența utilizatorilor prin servirea conținutului dintr-o locație apropiată.

În cadrul platformelor CDN, se utilizează în principal cache-uri la nivel de proxy pentru a optimiza livrarea conținutului și a reduce latențele percepute de utilizatori. Aceste cache-uri sunt amplasate strategic în diverse locații geografice, apropiate de utilizatorii finali, pentru a asigura o

livrare rapidă și eficientă a resurselor web.

În articolul [24], autorii observă că performanța generală a internetului poate fi îmbunătățită și congestia rețelei poate fi redusă la minimum prin utilizarea memoriei cache. De asemenea, din perspectiva utilizatorilor finali, memoria cache poate reduce semnificativ latența percepută de utilizatori și poate îmbunătăți experiența lor pe web. Din perspectiva infrastructurii internetului, memoria cache poate contribui la reducerea traficului web. Sistemele de caching permit gestionarea mai eficientă a creșterii cererii de conținut web, facilitând astfel scalarea infrastructurii web pentru a face față traficului crescut.

Sistemele de caching de tip proxy trebuie să ia în considerare trei decizii operaționale cheie pentru o transmisie eficientă și sigură a datelor. Prima decizie este înlocuirea, care se referă la modul de gestionare a capacității limitate de stocare în cache pentru a obține performanțe optime, prin eliminarea obiectelor mai puțin utilizate pentru a face loc celor noi. A doua decizie vizează consistența datelor, asigurând că datele livrate din cache sunt identice cu cele de pe serverul de origine. A treia decizie este preîncărcarea, care îmbunătățește performanța sistemului prin stocarea anticipată a unor resurse în cache, chiar înainte ca acestea să fie solicitate de utilizatori [24].

Algoritmii de înlocuire a memoriei cache sunt esențiali pentru gestionarea eficientă a memoriei cache, controlând ce informații sunt stocate și eliminând elementele atunci când capacitatea cache-ului este atinsă. Așa cum menționează și autorii, sistemele care folosesc tehnici de caching trebuie să aibă în vedere o modalitate eficientă de înlocuire a resurselor, datorită capacității limitate de stocare. În acest context, se explorează diferite tehnici de înlocuire, precum cele bazate pe Least Recently Used (LRU) și Least Frequently Used (LFU), care ajută la gestionarea acestor limitări și la asigurarea unei utilizări eficiente a spațiului de stocare disponibil în cache [24].

Cel mai utilizat algoritm de înlocuire a memoriei cache este principiul Least Recently Used. Este atractiv datorită implementării sale simple și complexității reduse a timpului de rulare [28]. Acesta funcționează pe principiul că datele accesate cel mai recent sunt cel mai probabil să fie accesate din nou în viitorul apropiat. Prin eliminarea elementelor accesate cel mai puțin recent, algoritmul LRU se asigură că cele mai relevante date rămân disponibile în cache. Cache-ul LRU la nivelul serverelor Redis menține o listă de elemente de date, cu cele mai recent accesate elemente în față și cele mai puțin recent accesate în spate. Când un element de date este accesat sau adăugat, acesta este mutat în partea de sus a listei. Dacă memoria cache își atinge capacitatea și trebuie să scoată un articol pentru a face loc pentru unul nou, elementul din spate, fiind cel mai puțin recent folosit, este eliminat. [10]

Un alt algoritm de eliminare a resurselor din cache este Least Frequently Used (LFU). Acesta evacuează elementele cel mai puțin accesate, menținând un contor al frecvenței de accesare pentru fiecare articol și eliminând elementul cu cel mai mic număr de accesări, atunci când este necesară evacuarea. Abordarea poate fi eficientă în cazul în care modelele de acces sunt consecvente, totuși, poate păstra datele rar utilizate dacă acestea au fost accesate frecvent într-o anumită perioadă, ceea ce poate duce la ineficiență în gestionarea memoriei cache [10].

Majoritatea serverelor web își actualizează frecvent conținutul publicat, iar unele resurse de pe servere pot deveni indisponibile. În aceste condiții, resursele stocate în cache pot deveni învechite sau chiar invalide. În contextul serverelor care utilizează tehnici de caching pot apărea două probleme majore observate de autorii [24].

Prima problemă apare atunci când serverul proxy livrează conținut învechit utilizatorilor finali. Având loc atunci când serverul proxy păstrează o copie a unei resurse web care a fost actualizată pe serverul de origine, dar modificările nu au fost propagate la proxy. Ca urmare, utilizatorii finali primesc o versiune mai veche a conținutului, ceea ce poate duce la informații incorecte sau experiențe de utilizare necorespunzătoare.

O altă problemă este supraîncărcarea lățimii de bandă consumată de comunicările legate de consistența datelor între proxy și serverul web de origine si apare atunci când există un flux constant de solicitări și răspunsuri între serverul proxy și serverul de origine pentru a verifica dacă

datele din cache sunt actualizate.

Pentru a asigura o livrare eficientă și actualizată a conținutului, este esențial să se țină cont atât de riscul de învechire, cât și de consumul de lățime de bandă.

Astfel, soluțiile de caching trebuie să fie proiectate pentru a echilibra necesitatea de a furniza date actualizate cu nevoia de a minimiza traficul de rețea. Aceasta implică utilizarea algoritmilor eficienți de înlocuire a memoriei cache, gestionarea corectă a politicilor de consistență și implementarea de mecanisme pentru preîncărcarea resurselor relevante.

1.3. Distribuirea încărcăturii

Echilibrarea sarcinii reprezintă practica de distribuire a sarcinilor de lucru computaționale între două sau mai multe computere. Pe internet, această tehnică este folosită frecvent pentru a împărți traficul de rețea între mai multe servere. În acest mod se reduce presiunea asupra fiecărui server, crescând eficiența acestora, accelerând performanța și reducând latența. Echilibrarea sarcinii este esențială pentru funcționarea corectă a majorității aplicațiilor de internet [12].

Distribuirea conținutului la scară largă de pe un singur server de origine nu mai este sustenabilă [26]. Retelele de livrare de conținut asigură o experiență fluidă a utilizatorilor în mediul online. În [26], autorii afirmă că *„CDN-urile funcționează pe baza redirecționărilor. Prin redirecționarea cererilor clientului către serverul surogat cel mai apropiat sau optim prin intermediul router-ului de solicitare”*. În cadrul unui CDN, redirecționarea solicitărilor clienților către cel mai apropiat sau optim server este crucială pentru livrarea rapidă a conținutului și pentru reducerea latenței.

Soluția propusă de autori vizează atingerea următoarelor obiective:

- Scăderea încărcării pe serverele de origine
- Utilizarea eficientă a resurselor de rețea în rețea
- Identificarea și eliminarea blocajelor de rețea
- Distribuție egală a sarcinii pe surogate CDN

Mecanismele de echilibrare a încărcării determină ce server ar trebui să gestioneze fiecare cerere pe baza unui număr de algoritmi diferiți. Acești algoritmi se împart în două categorii principale: statici și dinamici [12].

Algoritmii statici de echilibrare a sarcinii distribuie sarcinile de lucru fără a ține cont de starea curentă a sistemului. Un mecanism de echilibrare a încărcării static nu are informații despre modul în care servere funcționează lent sau care sunt subutilizate. În schimb, atribuie sarcinile de lucru pe baza unui plan prestabilit. Deși configurarea echilibrării sarcinii statice este rapidă și simplă, aceasta poate duce la ineficiențe semnificative [12].

Algoritmii dinamici de echilibrare a sarcinii iau în considerare disponibilitatea curentă, volumul de lucru și starea de sănătate a fiecărui server. Aceștia pot redistribui traficul de la serverele supraîncărcate sau cu performanțe slabe, către serverele subutilizate, menținând o distribuție uniformă și eficientă a sarcinilor. Totuși, configurarea echilibrării dinamice a sarcinii este mai complexă. Disponibilitatea serverului este influențată de mai mulți factori, inclusiv starea de sănătate și capacitatea generală a fiecărui server, dimensiunea sarcinilor distribuite și alți parametri relevanți [12].

Echilibrarea încărcăturii bazată pe DNS este un tip specific de echilibrare a sarcinii care utilizează sistemul DNS pentru a distribui traficul pe mai multe servere. Aceasta se realizează prin furnizarea unor adrese IP diferite ca răspuns la interogările DNS. Echilibratoarele de încărcare folosesc diverse metode sau reguli pentru a alege adresa IP care va fi partajată ca răspuns la o interogare DNS [13].

Una dintre cele mai comune tehnici de echilibrare a încărcăturii DNS este DNS Round-Robin. Metoda implică distribuirea ciclică a răspunsurilor DNS, asigurând astfel că fiecare server primește trafic într-un mod echilibrat și continuu. Deși acest algoritm este simplu de implementat, nu ia în considerare capacitatea serverelor sau latența rețelei, ceea ce poate duce la supraîncărcarea serverelor mai lente.

Un algoritm dinamic utilizat pe scară largă este cel bazat pe geo-locație. În această configurație, mecanismul de echilibrare a încărcării direcționează cererile dintr-o anumită regiune către un server sau un set de servere specifice, optimizând livrarea conținutului în funcție de locația geografică a utilizatorului. Această metodă duce la o latență redusă și o experiență îmbunătățită pentru utilizatori, însă necesită o infrastructură complexă pentru monitorizarea locației și a performanței, de asemenea, poate fi afectată de schimbările în topologia rețelei.

O altă modalitate de rutare a cererilor este printr-o rețea anycast. Rutarea prin rețeaua anycast permite direcționarea cererilor de conexiune primite către mai multe servere edge într-o rețea CDN. Atunci când solicitările sunt trimise către o singură adresă IP asociată rețelei Anycast, rețeaua distribuie datele pe baza unei metodologii de prioritizare. Procesul de selecție a serverului edge este de obicei optimizat pentru a reduce latența, alegând serverul cel mai apropiat de utilizator. Anycast se caracterizează printr-o asociere „unu la unu" între multe puncte și este una dintre cele cinci metode principale de protocol de rețea utilizate în protocolul internet [13].

Prin adoptarea diverselor metode de echilibrare a sarcinii, rețelele de livrare a conținutului (CDN) pot optimiza distribuția traficului și pot menține performanța ridicată și disponibilitatea constantă a serviciilor web. Implementarea unor algoritmi dinamici, cum ar fi cei bazați pe geo-locație sau anycast, asigură o adaptabilitate sporită la condițiile variabile ale rețelei și cerințele utilizatorilor, contribuind la o experiență îmbunătățită pentru utilizatorii finali.

1.4. Securitatea în aplicații CDN

În contextul actual al mediului online, CDN-urile joacă un rol crucial în garantarea furnizării conținutului către utilizatori. Acestea livra aproximativ 72% din traficul de internet în 2022, iar valoarea pieței CDN este de așteptat să crească de la 11,76 miliarde de dolari în 2019 la 49,61 miliarde de dolari în 2025 [27]. Totuși, rețelele sunt supuse la diverse amenințări de securitate care pot afecta atât performanța CDN-ului, cât și experiența utilizatorului final. Cu toate acestea, mecanismele de protecție sunt esențiale pentru a asigura și proteja disponibilitatea conținutului pe internet.

În studiul [27], autorii afirmă riscul la care sunt supuse CDN-urile, acestea *„trebuie să protejeze conținutul de furt și pierdere, păstrând în același timp disponibilitatea conținutului prin atenuarea atacurilor de securitate"*. O mare amenințare este reprezentată de armonizarea infrastructurilor împotriva intereselor utilizatorilor finali și ale site-urilor web care utilizează serviciile CDN. Atacatorii pot exploata vulnerabilitățile mecanismelor de stocare în cache ale CDN-urilor pentru a genera volume mari de trafic împotriva acestor site-uri web, afectându-le astfel performanța și disponibilitatea. Această manipulare a infrastructurii CDN poate lua diverse forme, fiecare având consecințe specifice și grave pentru integritatea și funcționarea serviciilor online.

Principalul tip de atac, care poate „paraliza" întreaga rețea, este Distributed Denial of Service(DDoS). Un atac DDoS este o încercare rău intenționată de a perturba traficul normal al unui server, serviciu sau rețea țintă prin copleșirea țintei sau a infrastructurii înconjurătoare cu un flux de trafic internet [14]. Acest tip de atac nu afectează doar platforma CDN, poate duce și la colapsul serverelor de origine, făcând conținutul indisponibil pentru utilizatorii finali.

Într-un atac de tip DDoS, adversarii urmăresc să împiedice utilizatorii legitimi să acceseze serviciile CDN. În acest tip de atac se utilizează mai multe entități atacatoare pentru a atinge același obiectiv. Atacatorii inundă serverele edge cu un volum mare de cereri, consumând resursele acestora și făcând conținutul inaccesibil pentru utilizatorii finali. Acesta poate duce la pierderi considerabile pentru platforma CDN din cauza indisponibilității serviciilor și a veniturilor pierdute.

Figura 1.3. Atac DDoS

În figura 2.4 se ilustrează un atac de tip DDoS (Distributed Denial of Service) asupra unui server edge dintr-o rețea CDN. Atacatorul inițiază atacul și coordonează rețeaua de calculatoare compromise pentru a viza un server edge. Serverul devine incapabil să răspundă cererilor obișnuite din cauza suprasolicitării, iar utilizatorii legitimi sunt respinși. Acest lucru poate duce în cele din urmă la căderea completă a serverului.

Atacatorii profită de vulnerabilitățile serverelor edge în servirea conținutului dinamic pentru a amplifica atacurile împotriva serverelor de origine. Aceștia exploatează, de asemenea, vulnerabilitățile din anteturile cererilor HTTP pentru a introduce conținut falsificat în cache-ul serverelor edge, ceea ce face ca utilizatorii să primească pagini de eroare în loc de conținutul original.

Mecanismele de protecție împotriva atacurilor DDoS includ tehnici precum limitarea ratei (Rate limiting), utilizarea filtrelor de tip firewall pentru aplicații web (Web Application Firewall) și difuzarea rețelei prin Anycast (Anycast Network Diffusion).

Rate limiting presupune limitarea numărului de solicitări pe care un server le va accepta într-o anumită perioadă de timp [14]. Implementarea rate limiting ajută la prevenirea utilizării excesive a resurselor serverului (CPU, memorie, lățime de bandă), asigurând o performanță stabilă și prevenind scăderea performanței sau căderea serverului. În plus, asigură că utilizatorii legitimi beneficiază de o experiență constantă și de calitate, fără întârzieri sau erori cauzate de supraîncărcare. Tehnici comune de rate limiting includ algoritmii Token Bucket și Leaky Bucket, care gestionează fluxul cererilor pentru a preveni depășirea capacității serverului.

Un Web Application Firewall(WAF) este un instrument care poate ajuta la atenuarea unui atac DDoS de nivel 7. Prin plasarea unui WAF între internet și serverul de origine, acesta poate acționa ca un proxy invers, protejând serverul țintă de anumite tipuri de trafic rău intenționat [14]. WAF-ul poate detecta și bloca traficul rău intenționat bazat pe un set de reguli predefinite care identifică tiparele suspecte și comportamentele anormale. De exemplu, poate bloca cererile care conțin injecții SQL, scripturi cross-site (XSS), și alte tipuri de atacuri web.

Anycast Network Diffusion folosește o rețea anycast pentru a dispersa traficul de atac într-o rețea de servere distribuite, până când traficul este absorbit de rețea [14]. În loc să direcționeze tot traficul către un singur server, anycast permite distribuirea acestuia către mai multe servere localizate în diferite regiuni geografice. Această metodă reduce efectul unui atac DDoS asupra unui singur punct al rețelei, deoarece traficul este răspândit pe o arie largă, diminuând impactul asupra fiecărui server individual. Prin dispersarea traficului, rețeaua poate absorbi mai eficient volumul mare de cereri, prevenind supraîncărcarea și menținând performanța și disponibilitatea serviciilor web

Platformele CDN gestionează un volum considerabil de resurse provenite de la clienții lor. Atacatorii pot compromite performanța cache-ului prin injectarea de conținut falsificat, ceea ce face ca utilizatorii să primească date eronate până la actualizarea resurselor. Ei pot folosi tehnici de „cache poisoning”, înlocuind răspunsurile cache legitime cu date falsificate, afectând integrita-

tea și autenticitatea conținutului livrat. În plus, adversarii pot exploata vulnerabilitățile traficului criptat pentru a ocoli mecanismele de securitate ale CDN-urilor și a viza direct serverele de origine, generând atacuri greu de detectat.

Criptarea traficului devine o practică standard în contextul securității rețelelor de livrare a conținutului, datorită adoptării protocoalelor web de generație următoare, cum ar fi HTTP/2. Criptarea traficului este realizată în principal prin utilizarea protocolului Secure Sockets Layer (SSL) și Transport Layer Security (TLS), care permit utilizatorilor finali să stabilească conexiuni securizate de tip end-to-end cu serverele de origine pentru accesarea conținutului. Atacatorii pot ascunde atacurile la nivelul aplicației în traficul criptat pentru a evita mecanismele de securitate ale CDN-ului și serverele edge, vizând direct serverele de origine. Deoarece traficul este criptat, CDN-urile au capacități limitate de a analiza traficul criptat între utilizatorii finali și serverele de origine, deoarece nu au acces la cheile private ale deținătorilor de conținut.

În contextul actual al amenințărilor cibernetice, protejarea platformelor CDN împotriva atacurilor DDoS și a altor tipuri de atacuri este esențială pentru asigurarea continuității și securității serviciilor online.

Capitolul 2. Proiectarea aplicației

2.1. Tehnologii utilizate

Pentru a implementa serverele în cadrul *CDN*-ului, s-a convenit asupra utilizării framework-ul Python FastAPI. Potrivit documentației oficiale, FastAPI [8] este un framework modern și eficient pentru construirea de API-uri(*Application Programming Interface*) cu Python. Am optat pentru FastApi pentru că este cunoscut pentru performanța și viteza sa, este capabil să gestioneze un volum mare de cereri cu timpi de răspuns rapizi, făcându-l o opțiune excelentă pentru nevoile aplicației. De asemenea, suportul pentru programarea asincronă înseamnă că poate gestiona eficient sarcinile paralele, contribuind la eficiența sistemului.

Cache-ul va fi susținut de baza de date Redis, datorită performanței sale și flexibilității pe care o oferă. Redis [9] oferă un acces rapid la date, deoarece acestea sunt stocate sub formă de perechi cheie-valoare în memoria RAM a sistemului. Acesta este dotat cu un sistem automat de eliminare a resurselor pe baza unui TTL (*Time-to-Live*), ceea ce permite gestionarea eficientă a spațiului de stocare și menținerea cache-ului actualizat în mod dinamic.

Pentru echilibrarea încărcăturii la nivelul DNS-ului (*Domain Name System*), se utilizează un server implementat in Python. Această abordare permite distribuirea echilibrată a traficului către serverele platformei, prevenind supraîncărcarea și reducând riscul unui singur punct de eșec, spre deosebire de utilizarea unui server reverse proxy.

În cadrul sistemului, sunt utilizate două baze de date principale: MariaDB [5], ce este utilizată pentru stocarea informațiilor despre utilizatori și MongoDB [4], ce este folosită pentru a gestiona și stoca informațiile referitoare la site-urile web.

Interfața utilizatorului este construită utilizând framework-ul React. React[7] este cunoscut pentru abilitatea sa de a crea interfețe utilizator grafice, dinamice și eficiente. Acesta permite dezvoltarea de componente reutilizabile și gestionarea eficientă a stării aplicației, asigurând o experiență fluidă și interactivă pentru utilizatori.

Pentru a asigura consistența și portabilitatea mediului de dezvoltare, este utilizat Docker și Docker Compose. Docker este o platformă care permite dezvoltatorilor să construiască, să testeze și să desfășoare aplicații în containere izolate, asigurându-se că acestea funcționează în mod consecvent, indiferent de mediul în care sunt rulate. Docker Compose permite definirea și gestionarea aplicațiilor multi-container, facilitând configurarea și orchestrarea serviciilor necesare pentru a rula aplicația într-un mod simplificat și automatizat [3].

2.2. Proiectarea Platformei CDN

O rețea de livrare a conținutului (CDN) este o rețea globală de servere distribuite strategic pentru a face față unui volum mare de cereri și pentru a asigura o livrare rapidă și sigură a conținutului. Aceasta este concepută să fie scalabilă și este alcătuită din numeroase servere situate în apropierea utilizatorilor finali, cunoscute sub numele de servere edge. Serverele de origine reprezintă serverele clienților platformei, care stochează versiunea originală a conținutului, în timp ce serverele edge păstrează copii ale acestuia.

Platforma CDN nu are acces la codul sursă al serverelor de origine și funcționează independent de acestea. În esență, platforma acționează ca un simplu client care solicită resurse de la serverele de origine, sunt un intermediar între utilizatorul final și serverele de origine. Atunci când un utilizator solicită o resursă, cererea este direcționată către serverul edge cel mai apropiat, care poate servi resursa direct din cache, reducând timpul de răspuns. Dacă resursa solicitată nu este disponibilă în cache, serverul edge acționează ca un client și solicită resursa de la serverul de origine.

În figura 2.1, este evidențiată distribuția geografică a serverelor pentru a acoperi un număr cât mai mare de utilizatori. Această amplasare strategică a serverelor va avea loc în cadrul infrastructurii cloud oferite de AWS (*Amazon Web Services*) [1]. Platforma este concepută să livreze cererile în mod dinamic, funcționând indiferent de numărul de servere, datorită arhitecturii scalabile care permite adăugarea flexibilă de resurse după necesitate.

Figura 2.1. Dispunerea geografică a edge serverelor

Platforma CDN este implementată folosind infrastructura cloud oferită de AWS. Alegerea AWS este justificată de scalabilitatea, fiabilitatea și disponibilitatea ridicată a serviciilor oferite. AWS permite distribuirea geografică a serverelor edge pentru a acoperi un număr mare de utilizatori și pentru a asigura livrarea rapidă a conținutului.

Arhitectura sistemului reprezintă un element esențial pentru a asigura livrarea rapidă a conținutului online către utilizatorii finali din întreaga lume. În conformitate cu detaliile discutate în secțiunile anterioare, platforma implică amplasarea mai multor edge servere în diverse locații geografice, acoperind o gamă extinsă de utilizatori, asigurându-se că acestea sunt plasate în zone geografice diferite.

În figura 2.2 sunt ilustrate componentele platformei care este compusă din server edge, server DNS, componenta de verificare a cache-ului, componenta de verificare a sănătății serverelor si serverul de origine. Mai sunt ilustrate și bazele de date și interacțiunea acestora cu componentele.

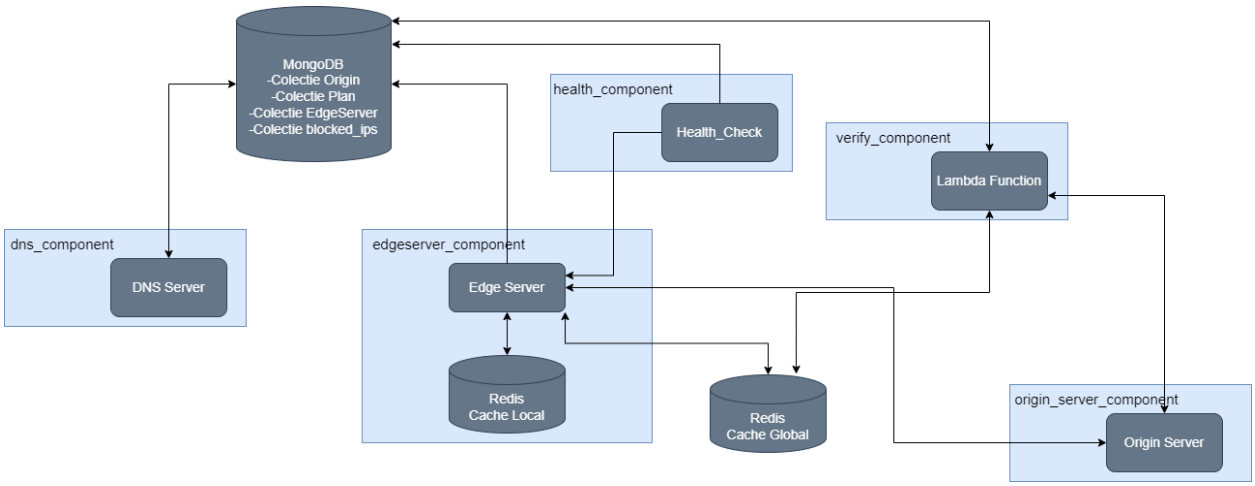

Figura 2.2. Diagrama de componente ale platformei CDN

Un edge server este un computer care există la extrema logică sau „marginea" unei rețele, servește adesea ca și conexiune între rețele separate. Scopul principal al unui server edge CDN este de a stoca conținut cât mai aproape de o mașină client care solicită resurse, reducând astfel latența și îmbunătățind timpii de încărcare a paginii [11]. Acestea acționează ca un scut de protecție, stocând în cache conținutul pentru a reduce sarcina pe serverele de origine [27].

Serverele au fost dezvoltate în Python folosind framework-ul FastAPI. Suportul serverului este asigurat de Uvicorn [6], datorită faptului că este un server ASGI (*Asynchronous Server Gateway Interface*). Acest lucru oferă suport pentru programarea asincronă, fiind benefic aplicațiilor ce necesită gestionarea eficientă a mai multor cereri în paralel. Serverele edge sunt implementate pe instanțe *EC2 t2.micro* datorită costurilor reduse și a capacității de gestionare a sarcinilor de dificultate ușoară până la moderată. Deși instanțele *t2.micro* pot prezenta un dezavantaj datorită resurselor limitate, pentru un mediu de dezvoltare sunt adecvate.

Protocolul HTTPS (*Hypertext Transfer Protocol Secure*) este utilizat între utilizatori și serverele edge pentru a asigura securitatea și integritatea datelor transmise. Platforma CDN utilizează certificate SSL/TLS (*Secure Sockets Layer/Transport Layer Security*), generate automat pentru a facilita criptarea traficului și pentru a oferi o comunicare securizată. Acest lucru protejează informațiile sensibile și asigură confidențialitatea utilizatorilor. Pentru a optimiza performanța și a reduce costurile, serverele edge folosesc protocolul HTTP (*Hypertext Transfer Protocol*) în comunicarea cu serverele de origine, lucru ce permite preluarea rapidă a resurselor fără a adăuga un overhead suplimentar datorat criptării.

Fiecare server va fi echipat cu o componentă locală de cache, bazată pe Redis, concepută pentru a stoca temporar conținutul frecvent accesat, astfel încât să poată răspunde rapid la cererile utilizatorilor. Această abordare de cache locală are ca scop îmbunătățirea timpului de încărcare a paginilor web și a experienței generale a utilizatorului prin minimizarea latenței și a timpilor de acces la conținutul solicitat.

Pentru a gestiona stocarea pe termen lung a conținutului, se dezvoltă o componentă de cache globală, tot bazată pe Redis, concepută pentru a stoca datele pe suporturi de stocare pe disc. Cache-ul global va fi accesibil tuturor serverelor din cadrul aplicației, permițând o distribuire uniformă a resurselor și o gestionare eficientă a conținutului. Servind ca o sursă centralizată de stocare a conținutului, asigură redundanță și disponibilitate. De asemenea, va facilita partajarea datelor între servere și va contribui la optimizarea capacității de stocare și a performanței generale a aplicației. Comunicarea între servere și componenta de cache global se realizează prin protocolul securizat TLS, asigurând integritatea și confidențialitatea datelor transmise.

Pentru a asigura un mod de rutare dinamic al cererilor către serverele edge, platforma dispune de un server DNS. Cererile sunt direcționate către un server Python care comunică prin `UDP` (*User Datagram Protocol*) pe portul `53`. DNS este un serviciu de nume ierarhic și distribuit care oferă un sistem de denumire pentru computere, servicii și alte resurse din internet sau alte rețele IP(Internet Protocol). Are rolul de a gestiona atribuirea și maparea numelor de domeniu în internet. Procesul implică desemnarea de servere de nume autoritative, care sunt responsabile pentru stocarea și distribuirea informațiilor despre domenii specifice [19].

Serverul DNS este responsabil de direcționarea cererilor utilizatorilor către cel mai apropiat server edge, optimizând traseul de livrare a conținutului. Abordarea permite distribuirea echilibrată a traficului către serverele platformei, prevenind supraîncărcarea și reducând riscul unui singur punct de eșec, spre deosebire de utilizarea unui server reverse proxy.

Pentru a asigura consistența și actualitatea datelor stocate în cache-ul global, platforma utilizează o funcție AWS Lambda, programată să ruleze la fiecare 24 de ore prin Amazon EventBridge, un serviciu de bus de evenimente complet gestionat, care permite declanșarea automată a funcțiilor Lambda pe baza unei reguli care va declanșa funcția Lambda la fiecare 24 de ore. Funcția Lambda verifică starea resurselor din cache-ul global, asigurându-se că datele stocate sunt actualizate și corecte. Verificarea implică compararea `ETag`-urilor resurselor sau utilizarea unei funcții

hash pentru a detecta modificările, actualizând resursele în cache-ul global dacă au fost modificate pe serverul de origine.

Platforma include o componentă de monitorizare a stării de sănătate a serverelor edge, implementată printr-un script Python. Această componentă efectuează verificări periodice la intervale de 5 minute pentru toate serverele edge care sunt active. Verificările se concentrează pe răspunsul serverelor la cererile HTTP și pe timpul de răspuns al acestora. Dacă un server nu răspunde după trei încercări consecutive, acesta este marcat automat ca fiind în stare de „CRASH”. În plus, dacă timpul de răspuns al unui server depășește 1 secundă, serverul este etichetat ca fiind „SLOW”, indicând performanțe reduse. Prin această metodologie, platforma asigură o distribuție optimă a traficului și o reacție rapidă la problemele de performanță și disponibilitate ale serverelor edge.

Figura 2.3 ilustrează fluxul de date în cadrul platformei CDN, prezentând modul în care cererile utilizatorilor sunt gestionate și răspunsurile sunt livrate rapid prin utilizarea serverelor edge alături de un sistem de cache local si global.

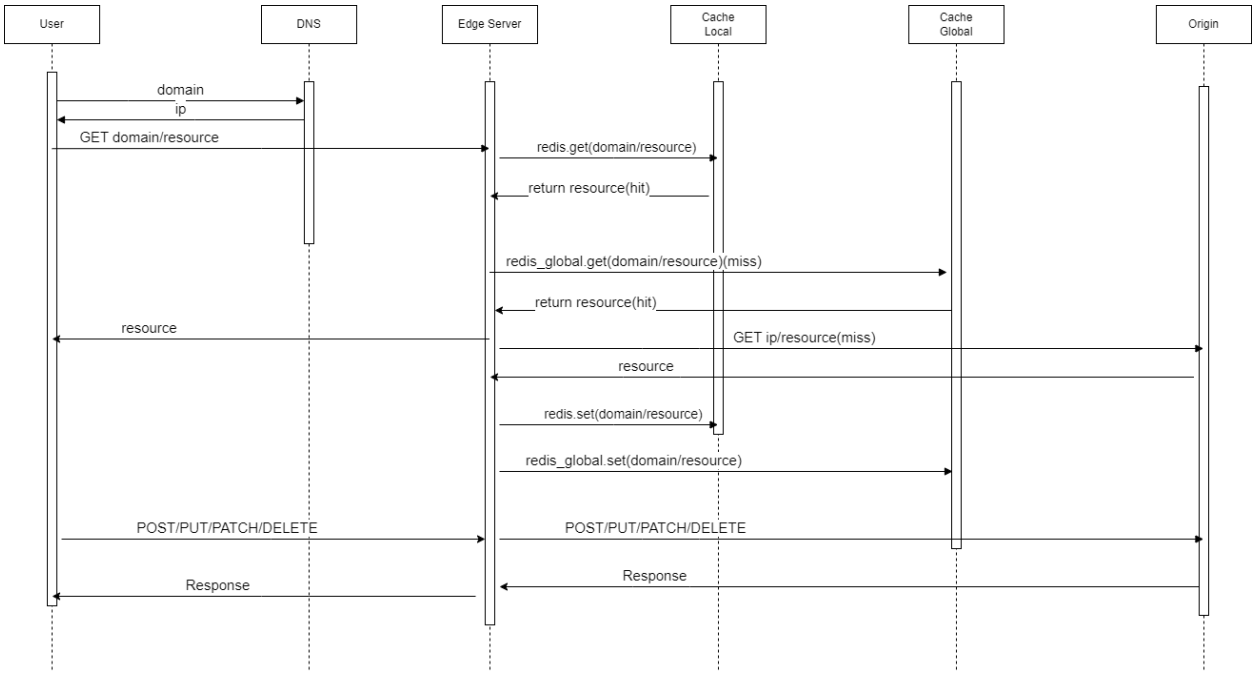

Figura 2.3. Diagrama de secvență

Atunci când un utilizator trimite o cerere către un site web înregistrat pe platforma CDN, procesul începe prin redirecționarea cererii către serverul DNS al platformei. Serverul DNS verifică dacă domeniul solicitat este gestionat de platformă și, dacă este valid, identifică și returnează adresa IP a celui mai apropiat edge server valid. Odată ce cererea ajunge la edge server, acesta verifică dacă domeniul este unul gestionat de platformă și aplică politicile specifice planului utilizatorului.

Edge serverul verifică mai întâi dacă resursa solicitată este disponibilă în cache-ul local, iar dacă resursa este găsită aici este livrată imediat utilizatorului, asigurând un timp de răspuns minim. Când resursa nu este disponibilă în cache-ul local, edge serverul verifică cache-ul global, iar dacă resursa este găsită în cache-ul global, aceasta este preluată de acolo, livrată utilizatorului și stocată în cache-ul local pentru cereri viitoare.

În cazul în care resursa nu este disponibilă nici în cache-ul local, nici în cache-ul global, edge serverul contactează serverul de origine pentru a prelua resursa solicitată. Odată obținută, resursa este livrată utilizatorului și stocată atât în cache-ul local, cât și în cache-ul global, conform politicilor de cache specificate în planul utilizatorului. Aceste politici includ opțiuni pentru tipurile de resurse care pot fi stocate, dimensiunea maximă a fișierelor și timpul de stocare în cache.

Astfel, arhitectura platformei CDN optimizează livrarea conținutului prin utilizarea de edge servere distribuite geografic, reducând latența și îmbunătățind timpii de încărcare. Cererile utilizatorilor sunt gestionate eficient prin cache-uri locale și globale, iar funcția AWS Lambda asigură consistența datelor. Serverul DNS direcționează cererile către cel mai apropiat edge server, prevenind supraîncărcarea și asigurând o performanță ridicată și fiabilă.

2.3. Proiectarea Filtrului DDoS

Filtrul DDoS (*Distributed Denial of Service*) este o componentă critică a platformei CDN, menită să protejeze resursele și să asigure continuitatea serviciului în fața atacurilor cibernetice. Acesta este integrat în arhitectura platformei CDN printr-un sistem de cozi SQS (*Simple Queue Service*). Sistemul monitorizează și analizează traficul pentru a detecta comportamente anormale și pentru a răspunde în timp real la potențialele amenințări.

Figura 2.4 ilustrează arhitectura filtrului DDoS utilizat pentru a proteja platforma CDN împotriva atacurilor de tip DDoS. Această arhitectură implică utilizarea mai multor servere edge, un serviciu de cozi de mesaje (SQS) și un detector de atacuri DDoS.

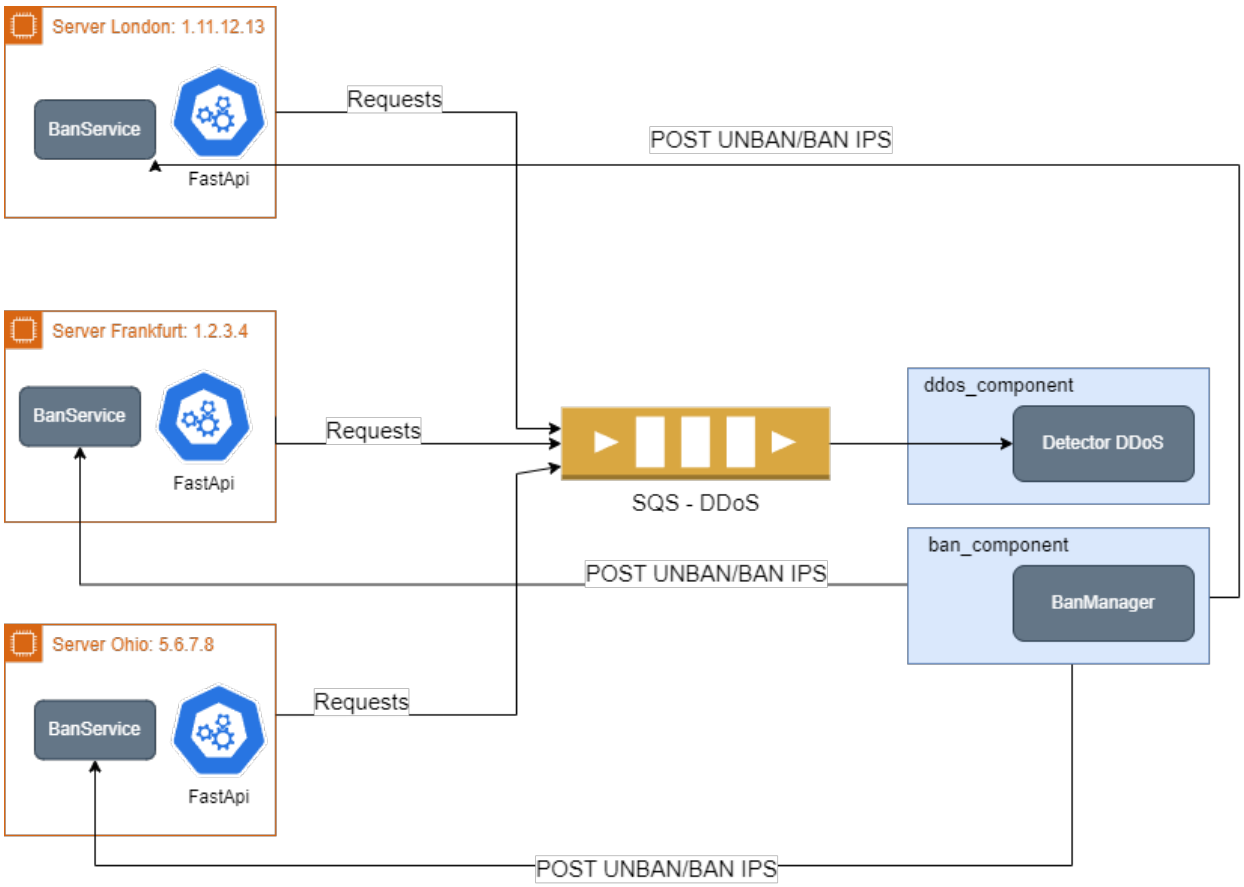

Figura 2.4. Arhitectura Filtrului DDoS

Toate edge serverele adaugă cererile primite în cadrul cozii SQS de tip FIFO (*First In, First Out*), cererile includ informații detaliate despre trafic, cum ar fi adresa IP a clientului, resursa solicitată, și timpul de răspuns. Utilizarea cozii SQS permite scalabilitatea și reziliența sistemului, asigurându-se că toate cererile sunt procesate într-un mod ordonat și eficient.

Componenta dedicată pentru detectarea atacurilor DDoS scoate cererile din coada SQS și le analizează pentru a identifica modele de trafic care pot indica un atac. Această componentă utilizează algoritmi de detectare a anomaliilor pentru a compara traficul actual cu modelele normale și pentru a identifica activitățile suspecte. Cererile sunt analizate pentru a detecta semne de atac

DDoS. Un indicator este un volum neobișnuit de mare de cereri provenind de la aceeași adresă IP.

Când sunt detectate comportamente suspecte, adresele IP ale clienților care au generat aceste cereri sunt identificate și marcate pentru acțiuni ulterioare. Odată identificate adresele IP suspecte, sunt trimise cereri de banare către componenta BanManager care notifică toate edge serverele, asigurând că adresele IP malițioase sunt blocate în întreaga rețea. Fiecare edge server primește cererea de banare și actualizează propriile reguli de firewall pentru a bloca traficul provenit de la adresele IP marcate. Acest lucru se face pentru a preveni accesul continuu al IP-urilor malițioase la resursele platformei, protejând integritatea și performanța serviciilor oferite. Blocarea IP-urilor la nivelul fiecărui edge server asigură o protecție distribuită, reducând riscul ca un atac să afecteze în mod semnificativ performanța sau securitatea rețelei.

2.4. Proiectarea Platformei Web

Platforma Web reprezintă un panou de control prin care utilizatorii își pot crea conturi și încărca site-uri web pentru a livra conținutul acestora prin intermediul platformei CDN. Platforma Web respectă modelul arhitectural REST (*Representational State Transfer*), asigurând un design stateless în care fiecare cerere conține toate informațiile necesare pentru a fi procesată independent. Resursele sunt identificate și accesibile prin URI-uri specifice, iar operațiunile asupra acestora sunt realizate folosind verbele HTTP standard (`GET`, `POST`, `PUT`, `DELETE`). Resursele sunt reprezentate în format standardizat, JSON (*JavaScript Object Notation*), facilitând interacțiunea între client și server.

În figura 2.5 este ilustrată arhitectura platformei web. Aceasta este alcătuită din trei servicii REST: Serviciul de origine, Serviciul de cache și Serviciul de EdgeServere. Pe lângă aceste trei servicii REST, platforma include și un serviciu gRPC de management al identităților (IDM). Toate aceste servicii comunică prin intermediul unui Gateway cu aplicația de frontend.

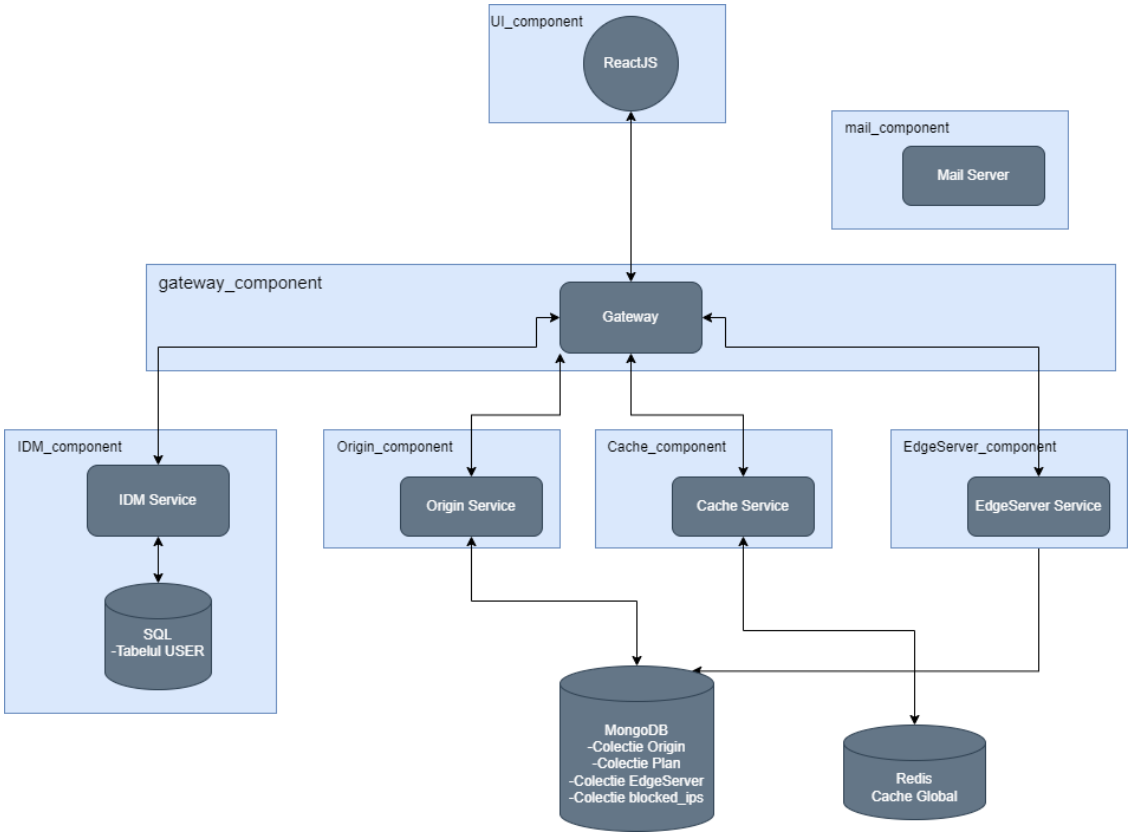

Figura 2.5. Diagrama pe componente Platformă Web

Serviciul de management al identităților are un rol esențial în cadrul platformei, este res-

ponsabil de autentificarea și autorizarea utilizatorilor. Acest serviciu utilizează protocolul gRPC pentru a asigura o comunicare eficientă și securizată între diferitele componente ale sistemului. gRPC permite comunicarea eficientă și rapidă datorită utilizării protocolului HTTP/2 și a formatului de serializare Protobuf (*Protocol Buffers*), oferind suport nativ pentru TLS, asigurând criptarea traficului între client și server. La nivelul acestui serviciu este emis și verificat token-uri JWT (*JSON Web Token*) pentru a asigura că doar utilizatorii autentificați și autorizați au acces la resursele platformei. Serviciul IDM utilizează o bază de date MariaDB pentru a gestiona resursa *„user”*, stocând și administrând informațiile despre utilizatori în mod securizat. Acesta comunică prin intermediul portului `50051`, asigurând astfel o izolare și securitate.

Serviciul de origine gestionează datele despre site-urile web înregistrate de utilizatori. Acest serviciu permite adăugarea, actualizarea și ștergerea site-urilor, precum și configurarea detaliilor specifice pentru fiecare site, cum ar fi planurile de cache și resursele statice. Serviciul utilizează baza de date MongoDB pentru stocarea informațiilor despre site-urile web și planurile asociate, prin două colecții *„origin”* și *„plan”*.

Serviciul de Cache este responsabil pentru gestionarea cache-ului asociat site-urilor web înregistrate pe platformă. Acesta permite utilizatorilor ștergerea resursele proprii din cache, vizualizarea starea acestora și oferă un sistem de bypass cache pentru situațiile în care site-ul este frecvent modificat, astfel încât să se livreze date cât mai actualizate. Serviciul se conectează la serverul Redis pentru stocarea cache-ului global și la baza de date MongoDB pentru gestionarea informațiilor despre site-urile web și utilizatori.

Serviciul de EdgeServere oferă administratorilor de sistem posibilitatea de a adăuga noi servere în baza de date, care vor fi utilizate de celelalte componente ale sistemului. Acesta asigură menținerea centralizată a tuturor serverelor și permite ștergerea și actualizarea acestora. De asemenea, prin intermediul acestui serviciu sunt aduse informații despre serverele din platforma AWS, cum ar fi utilizarea CPU-ului și alte metrici relevante pentru monitorizare.

Serviciul se conectează la baza de date MongoDB pentru gestionarea informațiilor despre EdgeServere prin intermediul colecției edgeserver. Datele despre utilizarea resurselor, inclusiv `CPUUtilization`, `NetworkIn` și `NetworkOut`, sunt extrase utilizând *AWS CloudWatch*. Aceste metrici sunt esențiale pentru monitorizarea performanței și sănătății serverelor edge.

În cadrul platformei, toate operațiunile sunt protejate prin autentificare JWT, garantând accesul la resurse doar pentru utilizatorii autentificați și autorizați. Validarea token-urilor JWT se realizează prin intermediul unui client gRPC, care comunică cu serviciul de gestionare a identităților. Autorizarea se realizează pe baza rolului utilizatorului, fie că este admin sau proprietar (owner), și verifică apartenența site-ului la utilizatorul autentificat. Fiecare serviciu generează documentația OpenAPI, furnizând o descriere detaliată a fiecărui endpoint și a răspunsurilor posibile, facilitând astfel integrarea și utilizarea eficientă a API-ului de către dezvoltatori.

Comunicarea între serviciile REST și gateway se realizează prin protocolul HTTP, în timp ce serviciul gRPC comunică cu gateway-ul prin intermediul protocolului HTTP/2. Comunicarea dintre aplicația backend și frontend se desfășoară, de asemenea, prin protocolul HTTP.

În figura 2.6 reflectă structura relațională a datelor în platforma CDN. Fiecare entitate și relație este definită clar pentru a asigura integritatea și consistența datelor. Această proiectare ajută la gestionarea eficientă a utilizatorilor, serverelor edge, resurselor și planurilor, asigurând totodată securitatea și performanța optimă a sistemului.

Relația dintre User și Origin este de tipul *one-to-many*, ceea ce înseamnă că un singur utilizator poate deține mai multe originuri. Este de remarcat că relația dintre plan și origin este de asemenea de tipul *one-to-many*, deoarece un plan poate fi asociat cu mai multe site-uri de origine, însă fiecare origin este asociată cu un singur plan. Aceasta structură suportă flexibilitatea în administrarea resurselor.

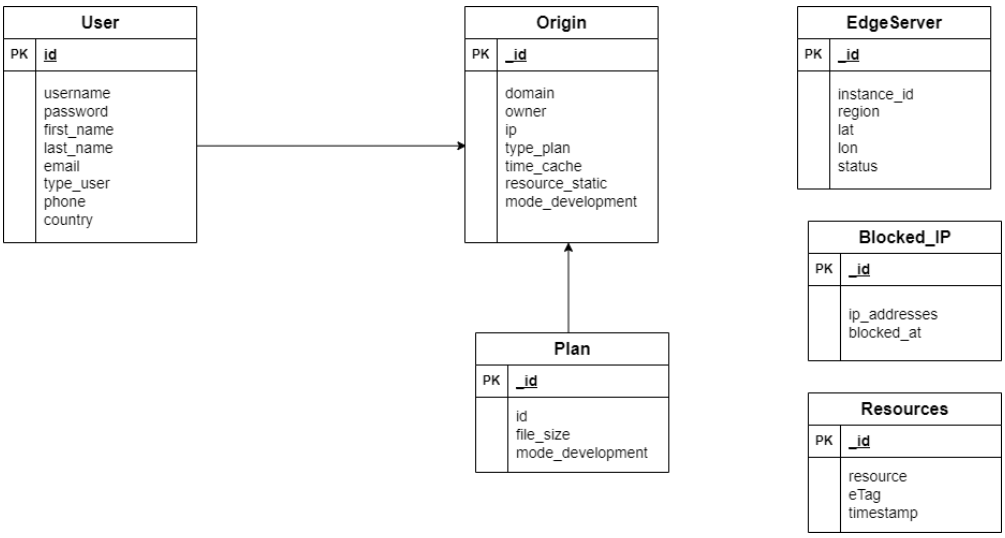

Figura 2.6. Diagrama Entităților

Entitatea User este utilizată pentru a gestiona informațiile despre utilizatorii platformei CDN. Fiecare utilizator are un cont în sistem, iar entitatea User stochează detalii relevante despre acești utilizatori. Platforma dispune de două categorii de utilizatori: admin și user obișnuit.

```
id VARCHAR(255) PRIMARY KEY,
username VARCHAR(255) NOT NULL,
password VARCHAR(255) NOT NULL,
first_name VARCHAR(255),
last_name VARCHAR(255),
email VARCHAR(255) NOT NULL,
type_user VARCHAR(50) NOT NULL,
phone VARCHAR(50),
country VARCHAR(100)
```

Listing 2.1: Structura tabelului USER

Entitatea Origin este utilizată pentru a gestiona informațiile despre site-urile web înregistrate pe platforma CDN. Fiecare site web (origine) are un set de detalii specifice stocate în această entitate. Origin este implementată folosind MongoDB. Câmpul *owner* face referință către utilizatorul proprietar al site-ului, făcând legătura cu entitatea User, iar câmpul *type_plan* face referință către planul pe care îl respectă site-ul. La nivelul acestei entități există indexi pentru optimizarea performanței pe câmpul *domain* si *owner*.

```
"origin":{
    "_id": "ObjectId",
    "owner": "String",
    "domain": "String",
    "ip": "String",
    "type_plan": "String",
    "time_cache": "String",
    "resource_static": [
      "String"
    ],
    "mode_development": "Boolean"
}
```

Listing 2.2: Structura colecției ORIGIN

Entitatea EdgeServer este utilizată pentru a gestiona informațiile despre serverele edge din rețeaua CDN, fiind implementată folosind MongoDB. La nivelul acestei entități există indexi pentru optimizarea performanței pe câmpul *status* și *instance_id*.

```
"edgeserver":{
    "_id": "ObjectId",
    "instance_id": "String",
    "region": "String",
    "lat": "Float",
    "lon": "Float",
    "status": "String",
  }
```

Listing 2.3: Structura colecției EdgeServer

Platforma dispune de un server de mail dedicat trimiterii de emailuri către utilizatori în diverse contexte, cum ar fi confirmarea înregistrării cu succes pe platformă sau notificarea atunci când un server de origine devine indisponibil. Serviciul utilizează biblioteca Python *smtplib* pentru trimiterea emailurilor prin protocolul SMTP (*Simple Mail Transfer Protocol*). Comunicarea securizată este asigurată prin utilizarea modulului SSL/TLS. Serverul de email folosit este Gmail (smtp.gmail.com), configurat pe portul 587 cu TLS activat pentru a garanta securitatea transmisiunilor.

Componenta de frontend a platformei web este construită folosind React [7], un framework popular pentru dezvoltarea interfețelor de utilizator interactive și reactive. React permite crearea de componente reutilizabile și gestionarea eficientă a stării aplicației, asigurând o experiență de utilizator fluidă și dinamică.

În cadrul platformei web, utilizatorii își pot crea un cont și pot adăuga site-uri web pentru a fi servite prin intermediul CDN-ului. Utilizatorii pot alege dintre trei planuri de stocare a datelor, fiecare cu propriile caracteristici și limitări: Basic, Advanced, Enterprise.

Planul Basic este proiectat pentru utilizatori care au nevoie de o soluție simplă pentru stocarea resurselor web de dimensiuni mici. Acest plan este ideal pentru site-uri web simple și aplicații mici. Permisiunile oferite de acest plan includ alegerea tipului de resurse stocate în rețea, alegerea timpului în care resursa să fie stocată în rețea și ștergerea resurselor de pe serverele noastre. Dimensiunea maximă a fișierelor stocate este de 1MB.

Planul Advanced este destinat utilizatorilor care au nevoie de mai multă flexibilitate și control asupra modului în care sunt stocate și servite resursele lor. Acest plan include toate permisiunile din planul Basic și adaugă modul development, care permite dezvoltatorilor să testeze și să actualizeze conținutul mai eficient. Dimensiunea maximă a fișierelor stocate este de 5MB.

Planul Enterprise este conceput pentru afaceri și aplicații mari care necesită stocare și acces rapid la resurse mari. Acest plan oferă toate permisiunile și funcționalitățile din planul Advanced, împreună cu suport și personalizare suplimentară pentru a răspunde cerințelor specifice ale clienților enterprise. Dimensiunea maximă a fișierelor stocate este de 10MB.

În figura 2.7 este ilustrat sistemul de monitorizare a traficului din cadrul platformei CDN. Fiecare edge server plasează mesaje într-o coadă SQS despre cererile care au avut loc, inclusiv adresa IP a clientului, resursa cerută, domeniul și serverul care a procesat cererea. Din coadă, o componentă de analiză extrage informațiile și trimite detaliile către aplicația de frontend, asigurând astfel că administratorul de sistem are acces la aceste date.

Componenta de analiză, scrisă în Python utilizând framework-ul FastAPI, rulează în fundal și extrage mesajele din coada SQS. Aceasta analizează și prelucrează datele, convertind informațiile brute într-un format structurat și ușor de înțeles. Componenta utilizează *boto3* pentru interacțiunea cu AWS SQS și requests pentru obținerea locației IP. Datele prelucrate sunt transmise către aplicația de frontend prin intermediul unui WebSocket. Aplicația de frontend afișează aceste date într-un

format accesibil și ușor de interpretat pentru administratori.

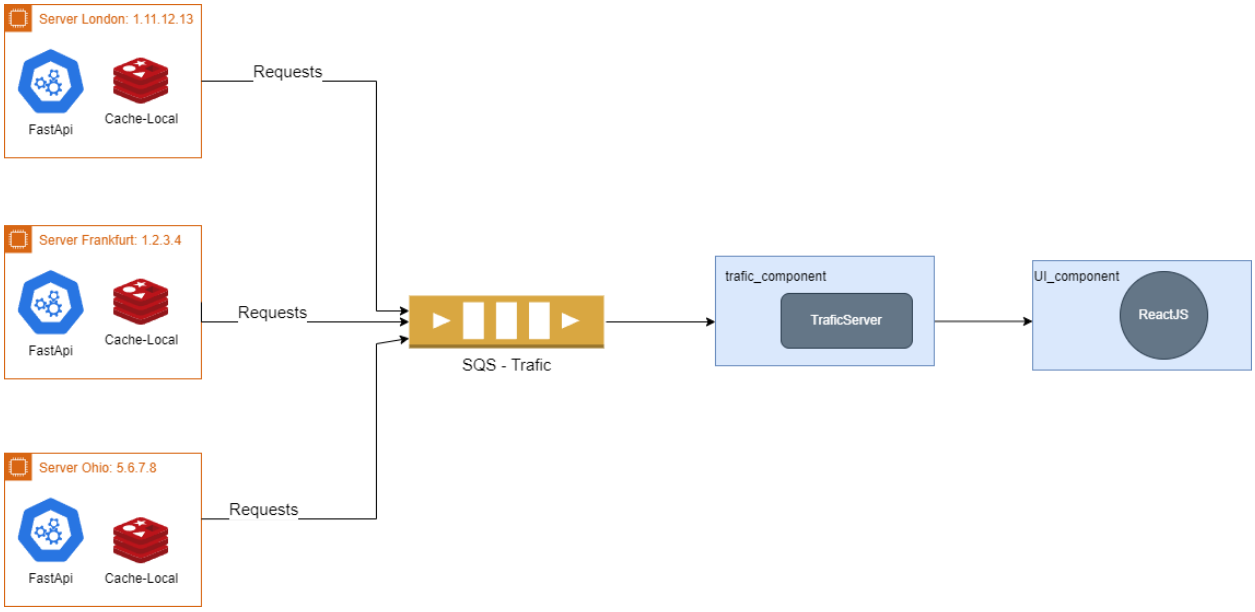

Figura 2.7. Sistem de monitorizare a cererilor

Capitolul 3. Implementarea aplicației

3.1. Server Edge

Edge serverele pot fi conceptualizate ca intermediari care conectează utilizatorii finali și serverele de origine. Ele acționează ca un „man in the middle", intermediind comunicarea între utilizatori și serverele de origine. Această intermediere are rolul de a ascunde identitatea și caracteristicile serverelor de origine, oferindu-le protecție împotriva accesului direct din partea utilizatorilor finali. Funcțional, ele acționează atât ca servere pentru utilizatori, cât și ca interfațe client pentru serverele de origine.

La pornirea serverului Uvicorn, se creează un client HTTP asincron folosind biblioteca *httpx* pentru a gestiona cererile către serverele de origine, și serverul își actualizează starea în baza de date indicând că este activ și gata să proceseze cereri. Uvicorn este configurat să utilizeze 5 workeri, o alegere ce sporește capacitatea de a gestiona un volum mare de cereri simultan fără a compromite performanța. Utilizarea mai multor workeri permite serverului să proceseze cereri în paralel, distribuind sarcina pe mai multe procese și minimizând timpul de răspuns.

Prin specificarea portului `443` în comanda de pornire, serverul Uvicorn este configurat să asculte pe toate interfețele de rețea la portul HTTPS standard, asigurând că este accesibil din orice rețea externă. De asemenea, utilizarea certificatului SSL (ssl_certfile) și a cheii private (ssl_keyfile) înseamnă că toate comunicațiile sunt criptate, oferind un nivel suplimentar de securitate pentru datele transmise între clienți și server.

Certificatele SSL/TLS sunt generate cu ajutorul bibliotecii *cryptography* din Python, așa cum este detaliat în Codul 9. Aceasta oferă instrumentele necesare pentru a crea atât cheia privată, cât și certificatul. Cheia Privată este creată folosind algoritmul RSA, cu o lungime de 2048 biți. Cheia este salvată în format PEM pe disc, fără criptare adițională, permițând utilizarea cheii pentru semnarea certificatului.

Certificatul este construit specificând subiectul și emitentul, care sunt identici, indicând că certificatul este auto-semnat. Certificatul include extensii cum ar fi Numele Alternative ale Subiectului (SAN) pentru a acoperi mai multe domenii. Validitatea este setată pentru un an, cu necesitatea reînnoirii periodice pentru a menține securitatea. Certificatul este semnat folosind cheia privată și algoritmul SHA256, oferind o garanție a autenticității și integrității.

La oprirea serverului, starea acestuia este actualizată în baza de date ca fiind oprită. În acest moment, toate conexiunile cu bazele de date și alte sisteme externe sunt închise corespunzător pentru a elibera resursele și a asigura o oprire curată.

În cadrul serverelor, obținem domeniul pentru care se face cererea pe baza antetului HTTP `Host` și verificăm în baza de date dacă acest domeniu este prezent. În cazul în care este găsit, extragem adresa IP (*Internet Protocol*) a serverului de origine asociat și trimitem cererile către acesta. În caz contrar, se trimite cod-ul de eroare `404 - Not Found`. Iar în cazul în care serverul de origine nu răspunde, se trimite codul `521 - Web server is down` la utilizator, iar către clientul platformei se va trimite un email pentru a rezolva problemele serverul său.

Atunci când un utilizator solicită o resursă web, serverul edge procesează inițial cererea. Dacă obiectul este disponibil în cache sub forma unei copii valide, aceasta este livrată imediat către utilizatori, eliminând necesitatea de a contacta serverul de origine pentru resursă. În caz contrar, dacă o copie a resursei nu este găsită în memoria cache, atunci serverul de origine este contactat pentru a furniza resursa. Aceasta este apoi preluată de către serverul edge și livrată utilizatorului. În același timp, resursa este stocată în cache pentru a putea fi utilizată în viitoarele cereri, îmbunătățind performanța și timpul de răspuns al platformei.

Funcția 3.1 este responsabilă pentru extracția datelor din cache. Aceasta încearcă să obțină conținutul resursei din cache-ul local Redis. Dacă conținutul este găsit, acesta este decodat din

formatul Base64 și returnat. În caz contrar, funcția încearcă să obțină conținutul din cache-ul global Redis. Dacă reușește, conținutul este stocat și în cache-ul local, după care este returnat utilizatorului.

```
1   def get_cached_content(resource_key, site_property):
2       try:
3           cached_content = r.get(resource_key)
4           if cached_content:
5               return base64.b64decode(cached_content)
6       except Exception as e:
7           print(f"Redis local cache error: {e}")
8       try:
9           cached_content = cache.get(resource_key)
10          if cached_content:
11              r.set(resource_key, cached_content,
12                ex=site_property['time_cache'])
13              return base64.b64decode(cached_content)
14      except Exception as e:
15          print(f"Redis global cache error: {e}")
16      return None
```

Listing 3.1. Funcția de extracție a datelor din cache

Înainte de a stoca resursa, se verifică, pe baza extensiei de fișier, dacă acest tip de resursă este configurat de client pentru a fi stocat, iar dimensiunea resursei este conformă cu planul ales de client. În plus, se verifică dacă modul bypass cache este setat. Dacă este setat, mecanismul de caching este evitat, iar resursa este preluată direct de la serverul de origine și livrată utilizatorului. Cererile către serverele de origine se trimit cu ajutorul bibliotecii *httpx*, asigurând comunicarea asincronă.

Funcția *fetch_and_cache*, Codul 7, gestionează trimiterea cererilor către serverul de origine și stochează resursele în cache dacă sunt eligibile. Dacă resursa este considerată potrivită pentru caching, este convertită în format Base64 înainte de a fi plasată în cache-ul global și, ulterior, în cache-ul local (Liniile 27 și 28). În cazul în care headerul `Cache-Control` include directivele `no-store` sau `no-cache`, acestea sunt transmise mai departe către client. Dacă aceste directive nu sunt prezente, `Cache-Control` va fi setat cu un timp de expirare de maxim 1000 de secunde. De asemenea, un header personalizat, `X-Source`, este adăugat pentru a indica sursa originală a răspunsului, facilitând astfel procesele de debugging și monitorizare.

În cadrul unui server edge, utilizarea Base64 pentru codificarea resurselor facilitează gestionarea acestora în diverse contexte care nu suportă date binare brute. Această metodă permite stocarea datelor binare în sisteme textuale fără a pierde informații sau a genera erori de interpretare. Prin prevenirea problemelor de interpretare, codificarea în Base64 asigură integritatea resurselor stocate, prevenind coruperea datelor. Metoda este esențială pentru a garanta comportamente previzibile și corecte la recuperarea și utilizarea resurselor stocate.

Pentru a asigura consistența datelor, atunci când o resursă este stocată în cache, în paralel se salvează un document, în baza de date MongoDB cu numele resursei și header-ul HTTP `ETag`, care servește ca un identificator unic al versiunii resursei(Linia 12). Salvarea are loc doar dacă acest header există,iar datele salvate sunt utilizate de componenta de verificare a cache-ului pentru a valida și gestiona versiunile resurselor, asigurându-se că informațiile din cache sunt actuale și consistente.

Un alt aspect important în gestionarea resurselor statice este compresia. Compresia resurselor statice, cum ar fi fișierele CSS, JavaScript și imaginile, poate reduce semnificativ dimensiunea resurselor, ceea ce duce la îmbunătățirea timpilor de încărcare a paginilor și la reducerea lățimii

de bandă necesare pentru a le transfera către utilizatori. Aceasta este realizată folosind tehnica Gzip pentru a comprima fișierele înainte de a fi trimise către utilizatori, optimizând performanța și eficiența rețelei.

Compresia datelor este realizată cu ajutorul bibliotecii *gzip*. În primă instanță, se verifică dacă datele ajung deja comprimate de la serverul de origine cu ajutorul antetului HTTP `Content-Encoding`. Dacă datele nu sunt deja comprimate, atunci ele vor fi comprimate la nivelul serverului edge.

```
1   def compress_with_gzip(content):
2       buff = BytesIO()
3       with gzip.GzipFile(fileobj=buff,
4           mode='wb') as gz:
5           gz.write(content)
6       compressed_content = buff.getvalue()
7       buff.close()
8       return compressed_content
```

Listing 3.2. Funcția de compresie a resurselor

Principalul scop al Edge Serverelor este gestionarea resurselor statice. Acest lucru implică creare de politici de înlocuire pentru a gestiona eficient capacitatea limitată a memoriei cache și pentru a asigura cele mai bune performanțe de stocare.

La nivel local, stocarea se realizează in-memorie datorită vitezei mari de răspuns, dar și datorită capacității lor de a fi redundante și scalabile. Resursele stocate în cache-ul local sunt înlocuite pe baza unui TTL, adică un interval de timp specificat, pentru a asigura actualitatea informațiilor și pentru a evita utilizarea resurselor expirate. Cache-ul local este vizibil doar pentru serverul edge care rulează pe aceeași instanță EC2, iar datele stocate în acest cache nu pot fi accesate de alte servere edge. Serverul Redis local este configurat automat să elimine resursele prin TTL, asigurând eliminarea resurselor vechi și utilizarea eficientă a spațiului de memorie.

```
maxmemory 512mb
maxmemory-policy volatile-ttl
```

Listing 3.1: Configurarile Redis pentru cache-ul local

Comunicarea cu serverul Redis care găzduiește cache-ul global se realizează prin TLS prin portul 6379, pentru a asigura securitatea comunicațiilor, protejând datele împotriva interceptării și atacurilor de tip man-in-the-middle. Utilizarea TLS asigură criptarea datelor transmise între edge servere și serverul Redis, garantând confidențialitatea și integritatea informațiilor.

Funcția 3.3 folosește biblioteca Python *redis* pentru a stabili și a menține conexiunea cu cache-ul global. Aceasta oferă un client pentru manipularea Redis din cadrul aplicațiilor Python, permițând gestionarea conexiunilor securizate prin TLS.

```
1   def create_redis_connection(instance_id, redis_port=6379):
2      try:
3          public_ip = get_instance_public_ip(instance_id)
4          r = redis.StrictRedis(
5              host=public_ip,
6              port=redis_port,
7              decode_responses=True,
8              ssl=True,
9              ssl_cert_reqs=ssl.CERT_NONE
10         )
11         return r
```

```
    except Exception as e:
        print(str(e))
        return None
```

Listing 3.3. Funcția de conectare la cache-ul global

Funcționalitățile anterioare sunt valabile pentru operații HTTP de tip `GET`, iar pentru restul cererilor, platforma funcționează ca un reverse proxy. Astfel, pentru cererile de tip `POST`, `PUT`, `DELETE` și `PATCH`, serverul edge transmite cererile direct către serverul de origine, fără a le stoca în cache. Platforma acționează ca un intermediar transparent, preluând cererile de la utilizatori, adăugând informații suplimentare necesare și transmițându-le mai departe către serverele de origine. Răspunsurile primite de la serverele de origine sunt apoi returnate utilizatorilor finali.

Codul 3.4 demonstrează modul de gestionare a cererilor HTTP de tip `POST`, `PUT`, `DELETE` și `PATCH` la nivelul serverelor edge. Este utilizată o cerere asincronă pentru a transmite cererile către serverul de origine, preluând răspunsul și redirecționându-l înapoi la client.

```
try:
    response = await http_client.request(
        request.method, PROTOCOL + ip_origin
        + request.url.path, headers=request.headers,
        data=await request.body()
    )
    content = response.content
    headers = {k: v for k, v in response.headers.items()}

    sent_message_to_sqs(queue_url, message_body=message)
    return Response(content=content,
        status_code=response.status_code,
        headers=headers)
except httpx.RequestError as e:
    logger.error(f"Request error for domain {domain}: {e}")
    //tratare exceptii
```

Listing 3.4. Modul de gestionare a cererilor de tip POST, PUT, DELETE, PATCH

Handler-ul de eroare, Funcția 3.5, este responsabil pentru capturarea excepțiilor care pot apărea în timpul preluării și procesării cererilor. Acesta se asigură că utilizatorul primește un răspuns corespunzător, chiar și atunci când ceva nu merge bine. Dacă serverul edge nu poate contacta serverul de origine, handler-ul va returna un cod de eroare specific, cum ar fi `521 - Web Server Is Down`, pentru a informa utilizatorul că serverul de origine nu este disponibil. În cazul în care un server de origine nu răspunde, handler-ul de eroare poate trimite o notificare către administratorul platformei sau proprietarul domeniului, informându-i despre problema întâmpinată. Acest lucru se poate realiza prin trimiterea unui e-mail automatizat, utilizând un client HTTP pentru a contacta un serviciu de mail.

```
async def handle_request_error(domain, document):
 try:
    ip = get_instance_public_ip('i-0de96d646a609bc91')
    id = document.get('owner')
    response = httpx.get(f'http://{ip}/api/cdn/users/mail/{id}')
    data = response.json()
    email = data.get('email')
    data = {
        "domain": domain,
```

```
10          "email": email
11      }
12      async with httpx.AsyncClient() as client:
13      await client
14           .post(f"http://{ip}:8200/api/mail-server/client-origin",
15           json=data)
16   except Exception as e:
17      //tratare exceptii
```

Listing 3.5. Modul de gestionare a erorilor

Fiecare server edge este echipat cu o componentă pentru blocarea și deblocarea adreselor IP în cazul atacurilor, reprezentată de un server FastAPI. Acest server rulează pe aceeași instanță EC2 și ascultă pe portul 80, fiind dedicat pentru primirea cererilor de blocare și deblocare IP.

Aceste funcționalități permit administratorilor să gestioneze IP-urile blocate și să prevină potențiale amenințări. Metodele *handle_ban_ip* și *handle_unban_ip* utilizează *iptables* pentru a gestiona regulile de blocare și deblocare a IP-urilor la nivel de sistem. Acestea sunt implementate asincron și se integrează cu FastAPI pentru a permite administrarea IP-urilor prin intermediul endpoint-urilor expuse.

Pentru a executa comenzi *iptables* din Python, se utilizează biblioteca *subprocess*, ce este folosit pentru a invoca comenzi de sistem, facilitând interacțiunea directă cu shell-ul sistemului de operare.

Metoda *handle_ban_ip*, 3.6, utilizează comanda Linux ,,`sudo iptables -A INPUT -s ip _address -p tcp -dport 443 -j DROP`'', care adaugă (-A) o regulă în lanțul INPUT pentru a bloca (j DROP) traficul TCP provenit de la adresa IP specificată (-s ip_address) pe portul 443 (–dport 443).

```
1   async def handle_ban_ip(ip_address: str, db,logger):
2       try:
3           db.blocked_ips.insert_one({
4               "ip_address": ip_address,
5               "blocked_at": datetime.utcnow()
6           })
7           subprocess.run(['sudo', 'iptables', '-A', 'INPUT', '-s',
            ↪  ip_address, '-p', 'tcp', '--dport', '443', '-j', 'DROP'],
            ↪  check=True)
8           logger.info(f"IP {ip_address} has been blocked.")
9       except subprocess.CalledProcessError as e:
10          logger.error(f"Failed to block IP {ip_address}: {e}")
11          raise HTTPException(status_code=500, detail=str(e))
12      except DuplicateKeyError:
13          logger.warning(f"IP {ip_address} is already blocked.")
14          raise HTTPException(status_code=409, detail="IP address
            ↪  already blocked.")
15      except Exception as e:
16          logger.error(f"Failed to store IP {ip_address}: {e}")
17          raise HTTPException(status_code=500, detail="Failed to store
            ↪  IP address.")
```

Listing 3.6. Funcția handle_ban_ip

Metoda *handle_unban_ip*, 3.7, este utilizată pentru a debloca o adresă IP care a fost anterior blocată folosind *iptables*. Aceasta metodă verifică dacă există o regulă de blocare pentru adresa IP

specificată și, dacă există, o elimină. Comanda `iptables -C INPUT -s ip_address -p tcp -dport 443 -j DROP` este utilizată pentru a verifica dacă există o regulă în lanțul INPUT care blochează traficul TCP de la adresa IP specificată pe portul 443. Dacă regula de blocare există, se rulează comanda `iptables -D INPUT -s ip_address -p tcp -dport 443 -j DROP` pentru a elimina această regulă.

```
1   async def handle_unban_ip(ip_address: str, logger):
2       try:
3           while True:
4               check_rule = subprocess.run(
5                   ['sudo', 'iptables', '-C', 'INPUT', '-s', ip_address,
                     ↪  '-p', 'tcp', '--dport', '443', '-j', 'DROP'],
6                   stdout=subprocess.PIPE, stderr=subprocess.PIPE
7               )
8               if check_rule.returncode == 0:
9                   remove_rule = subprocess.run(
10                      ['sudo', 'iptables', '-D', 'INPUT', '-s',
                         ↪  ip_address, '-p', 'tcp', '--dport', '443',
                         ↪  '-j', 'DROP'],
11                      check=True
12                  )
13                  if remove_rule.returncode == 0:
14                      logger.info(f"IP {ip_address} has been
                         ↪  unblocked.")
15                  else:
16                      logger.error(f"Failed to remove blocking rule for
                         ↪  IP {ip_address}.")
17                      raise HTTPException(status_code=500,
                         ↪  detail=f"Failed to remove blocking rule for
                         ↪  IP {ip_address}.")
18              else:
19                  logger.warning(f"No more blocking rules found for IP
                     ↪  {ip_address}.")
20                  return {"message": f"IP {ip_address} has been
                     ↪  unblocked from all rules."}
21      except subprocess.CalledProcessError as e:
22          logger.error(f"Failed to unblock IP {ip_address}: {e}")
23          raise HTTPException(status_code=500, detail=f"Failed to
             ↪  unblock IP")
```

Listing 3.7. Funcția handle_unban_ip

Componenta de logging este esențială pentru orice aplicație server, permițând monitorizarea comportamentului acesteia și diagnosticarea rapidă a problemelor. În codul 8, componenta de logging este configurată pentru a oferi detalii amănunțite despre activitatea serverului edge, utilizând biblioteca standard *logging* din Python. Funcția *setup_logging* stabilește configurația pentru logging, permitând înregistrarea detaliată a evenimentelor care au loc în cadrul aplicației. Loggerul este configurat pentru a captura mesaje de nivel `DEBUG`, ceea ce înseamnă că toate mesajele de debug, informaționale, de avertizare și de eroare vor fi înregistrate, fiind scrise în fișierul „app.log”.

Prin implementarea acestor servere, se asigură un flux de trafic fără întreruperi, în timp ce se urmărește și îmbunătățirea performanței pentru a crește viteza de procesare.

3.2. Cache-ul Global

La nivel global, resursele sunt stocate pe disc datorită capacității mari a memoriei sale. În acest context, atunci când memoria cache atinge capacitatea sa limitată, resursele sunt înlocuite conform LRU (*Least Recently Used*). Astfel, resursele care nu au fost accesate recent sunt eliminate pentru a face loc pentru noi resurse sau pentru cele care sunt utilizate mai frecvent. Redis oferă suport pentru stocarea pe disc prin două metode principale Redis Database File și Append-Only File.

Metoda RDB salvează instantanee (snapshots) ale bazei de date Redis la intervale specificate de timp. Aceasta creează un fișier de instantanee (snapshot file) care conține o copie binară a datelor din memorie [9].

Metoda AOF înregistrează fiecare operațiune de scriere primită de serverul Redis într-un fișier de jurnal (append-only log file). Acest fișier păstrează un istoric complet al modificărilor, ceea ce permite o recuperare precisă a datelor [9].

În cadrul platformei se utilizează metoda AOF (Append-Only File) deoarece reduce riscul pierderii datelor, asigurând că modificările sunt salvate pe disc aproape în timp real. Fișierul AOF păstrează un istoric complet al operațiunilor, permițând o recuperare exactă a stării bazei de date. De asemenea, fișierul AOF poate fi rescris periodic pentru a reduce dimensiunea acestuia și pentru a îmbunătăți performanța de citire.

```
//comunicare TLS
tls-port 6379
tls-cert-file /etc/redis/cert/redis.crt
tls-key-file /etc/redis/cert/redis.key
//eliminarea resurselor prin LRU
maxmemory 512mb
maxmemory-policy allkeys-lru
//implementarea modului AOF
appendonly yes
appendfilename "appendonly.aof"
appendfsync everysec
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

Listing 3.2: Configurarile Redis TTL

3.2.1. Validarea datelor

Validarea datelor se realizează periodic, la intervale de 24 de ore, pentru a evita suprasolicitarea serverelor de origine cu cereri frecvente și pentru a le menține disponibile pentru solicitările utilizatorilor finali. Această validare periodică este esențială pentru a asigura consistența și actualitatea resurselor cache-uite, fără a compromite performanța serverelor de origine.

Validarea este orchestrată de o funcție AWS Lambda, Codul 10, care este activată de un eveniment AWS EventBridge. EventBridge este configurat să trimită cereri către funcția Lambda la fiecare 24 de ore, iar funcția Lambda se ocupă de verificarea și actualizarea resurselor cache-uite, asigurând că toate datele sunt corecte și actualizate.

Funcția începe prin conectarea la două servicii esențiale: Redis, pentru cache-ul global, și MongoDB, pentru stocarea metadatelor resurselor. Validarea are loc prin parcurgerea tuturor cheilor din Redis și validează fiecare resursă. Procesul implică verificarea existenței resurselor pe serverele de origine și face o comparație între conținutul cache-ului și cel al serverului de origine.

Prima metodă de validare și cea mai eficientă utilizează compararea headerelor ETag. Această tehnică implică trimiterea de cereri `HTTP HEAD` către serverele de origine pentru a verifica dacă resursa a fost modificată. Dacă resursa nu a fost modificată (`304-Not Modified`),

aceasta rămâne în cache. În cazul în care resursa nu mai există (`404 Not Found`), sau resursa a fost modificată, este eliminată atât din cache, cât și din baza de date MongoDB. Această metodă asigură un consum minim de lățime de bandă pentru serverul de origine și permite un timp de verificare rapid.

Dacă o resursă nu dispune de un header `ETag`, se efectuează o cerere HTTP `GET` pentru a recupera resursa direct de la serverul de origine. Conținutul obținut este apoi comparat cu cel din cache utilizând hash-uri *SHA-256* pentru a asigura integritatea datelor. Operațiune se realizează folosind biblioteca *hashlib*, care permite calcularea eficientă a hash-urilor în Python. Dacă hash-urile rezultate nu coincid, înseamnă că resursa din cache este învechită și, ca urmare, aceasta este eliminată din cache. În situația în care resursa nu mai este disponibilă pe serverul de origine, indicat de un cod de stare `404`, aceasta este de asemenea eliminată din cache. Metoda implică un consum suplimentar de lățime de bandă pentru serverul de origine și un timp mai mare de verificare, deoarece fiecare validare necesită recuperarea și compararea conținutului.

3.3. Componenta de monitorizare a sănătății serverelor

Componenta de monitorizare a sănătății serverelor edge, Codul 11, utilizează un script Python asincron pentru a efectua verificări periodice și eficiente ale stării serverelor. Scriptul este proiectat să ruleze continuu, efectuând verificări la intervale de 5 minute, asigurând astfel disponibilitatea și performanța optimă a rețelei de servere edge.

Scriptul utilizează biblioteca *asyncio* pentru execuția asincronă a funcțiilor și biblioteca *httpx* pentru a trimite cereri HTTP către serverele edge. Prin aceste verificări, scriptul evaluează starea de sănătate a fiecărui server, înregistrând timpii de răspuns și verificând codurile de stare HTTP returnate. Dacă un server nu răspunde după trei încercări consecutive, acesta este marcat ca „CRASH”. De asemenea, serverele cu un timp de răspuns mai mare de 1 secundă sunt etichetate ca „SLOW”, indicând performanțe reduse. Această metodă permite monitorizarea în timp real a stării serverelor și intervenția rapidă în caz de probleme, contribuind la menținerea unei rețele de livrare a conținutului fiabile și eficiente.

Scriptul utilizează un client MongoDB pentru a accesa resursa EdgeServer. Funcția *monitor_edge_servers* verifică periodic starea serverelor edge din baza de date care nu sunt oprite (au status diferit de „OFF”). Aceasta extrage serverele din baza de date, obține adresa IP publică a fiecărui server folosind funcția *get_instance_public_ip* și trimite cereri HTTP asincrone pentru a verifica starea fiecărui server folosind funcția *check_https_server_async*. În funcție de răspunsul primit, scriptul actualizează starea serverului în baza de date.

Funcția *check_https_server_async* trimite cereri HTTP asincrone pentru a verifica starea unui server edge. Aceasta utilizează biblioteca *httpx* pentru a trimite cereri de tip HEAD către endpoint-ul `/health` al fiecărui server edge, înregistrând timpul de răspuns pentru fiecare cerere. Funcția încearcă să contacteze serverul de până la trei ori înainte de a renunța.

La fiecare încercare, se înregistrează timpul de început, se trimite o cerere `HTTP HEAD` asincronă către URL folosind *httpx.AsyncClient*, iar timpul de răspuns este măsurat de la momentul trimiterii cererii până la primirea răspunsului. Dacă serverul răspunde cu codul de status `200`, funcția returnează „True” și timpul de răspuns. Dacă serverul răspunde cu un alt cod de status sau dacă cererea eșuează din cauza unei erori de rețea, se afișează un mesaj de eroare și se așteaptă 1 secundă înainte de a încerca din nou.

Dacă toate încercările eșuează, funcția returnează „False” și „None”. Se utilizează cereri de tip `HEAD` pentru a reduce consumul de lățime de bandă pe serverele edge, deoarece aceste cereri nu descarcă corpul răspunsului, ci doar anteturile HTTP.

Monitorizarea continuă și actualizarea în timp real a stării serverelor permite intervenția promptă în caz de defecțiuni, menținând astfel o infrastructură solidă pentru livrarea conținutului.

3.4. Server DNS

În contextul rețelelor de livrare a conținutului, eficiența și rapiditatea în direcționarea utilizatorilor către cel mai apropiat server edge sunt esențiale pentru performanța aplicației. Pentru a facilita acest proces, serverul DNS dinamic este proiectat pentru a direcționa traficul către cel mai apropiat și disponibil edge server din rețeaua de livrare a conținutului.

Cererile DNS sunt inițial gestionate de instanța *DNSServer* din biblioteca *dnslib*. Acest server funcționează pe portul 53, așteptând să primească și să proceseze interogările DNS de la clienți. La primirea unei cereri, *DNSServer* utilizează capabilitățile de decodare ale *dnslib* pentru a analiza și extrage detalii cum ar fi tipul interogării (de exemplu, A, AAAA), numele domeniului vizat, și alte opțiuni DNS relevante.

După decodarea cererii, ea este redirecționată către *DynamicResolver*, clasa responsabilă pentru logistica de alegere a serverului edge adecvat. Codul 12, prezintă cum *DynamicResolver*, care extinde *BaseResolver* din *dnslib*, implementează o logică personalizată pentru a determina cel mai potrivit server edge. Acest proces de selecție se bazează pe informații despre locația geografică a solicitantului și starea curentă a serverelor edge, valorificând astfel datele stocate în cache-uri pentru a optimiza răspunsurile.

Serverul menține un cache pentru a îmbunătăți performanța și a reduce latența. Cache-ul stochează informațiile despre adresele IP ale serverelor edge și răspunsurile DNS anterioare. Acest cache este gestionat cu ajutorul bibliotecii *cachetools*, care oferă funcționalități de expirare a intrărilor bazate pe un timp de viață prestabilit (*TTL*).

```
1   ip_cache = TTLCache(maxsize=100, ttl=600)
2   edgeservers_cache = TTLCache(maxsize=100, ttl=60)
3   dns_response_cache = TTLCache(maxsize=100, ttl=300)
```

Listing 3.8. Configurarea cache-ului

ip_cache stochează adresele IP asociate cu instanțele EC2, atât IPv4, cât și IPv6. Când o cerere pentru o adresă IP este primită, serverul verifică mai întâi acest cache pentru a vedea dacă adresa este deja stocată, reducând astfel necesitatea de a interoga direct serviciile AWS EC2 sau de a efectua un lookup DNS extern. Acest lucru scade latența percepută de utilizator și reduce sarcina pe serviciile externe.

edgeservers_cache stochează informații despre starea și locația geografică a fiecărui server edge activ. Cache-ul este consultat atunci când trebuie să se determine cel mai apropiat server edge de un utilizator pe baza locației sale geografice. Stocarea acestor informații în cache permite serverului DNS să evite interogările frecvente către baza de date internă sau alte servicii backend pentru fiecare cerere DNS primită, accelerând procesul de răspuns.

dns_response_cache este utilizat în mod special pentru a stoca răspunsurile la interogările DNS pentru domeniile care nu sunt gestionate direct de platforma CDN. Când se primește o cerere DNS pentru un domeniu care nu este recunoscut sau administrat de infrastructura CDN, serverul DNS va căuta în acest cache pentru a vedea dacă un răspuns corespunzător a fost generat recent. Dacă un răspuns este disponibil în cache, el poate fi trimis imediat către client, eliminând necesitatea unei interogări suplimentare către serverul DNS al sistemului de operare.

Pentru determinarea adresei IP care urmează să fie trimisă utilizatorului, se utilizează funcția *get_best_server_ip*, Codul 13. Inițial, serverele edge sunt selectate pe baza activării lor, asigurându-se că doar serverele active sunt luate în considerare. Acest lucru garantează că cererile nu sunt direcționate către servere oprite sau supraîncărcate, incapabile să proceseze eficient cererile.

După aceasta, se realizează extragerea informațiilor despre utilizator, obținând poziția geografică, inclusiv longitudinea și latitudinea, prin intermediul unei cereri către API-ul oferit de *ipinfo.io* [20]. Dacă informațiile despre locația clientului nu pot fi obținute, se selectează un server

aleatoriu din cele valide. Biblioteca *requests* este folosită pentru a obține informații geografice despre IP-urile clienților de la *ipinfo.io*.

Calculul celui mai apropiat server se realizează folosind formula *Haversine*, care calculează distanța dintre două puncte pe suprafața unei sfere, utilizând longitudinea și latitudinea acestor puncte.

```
1   def haversine(lat1, lon1, lat2, lon2):
2       R = 6371.0
3       dlat = radians(lat2 - lat1)
4       dlon = radians(lon2 - lon1)
5       a = sin(dlat / 2)**2 + cos(radians(lat1))
6           * cos(radians(lat2))
7           * sin(dlon / 2)**2
8       c = 2 * atan2(sqrt(a), sqrt(1 - a))
9       distance = R * c
10      return distance
```

Listing 3.9. Funcția Haversine

După identificarea serverului edge cel mai apropiat, urmează evaluarea utilizării CPU-ului său, asigurându-se că nu depășește pragul de 70%. Dacă acesta este cazul, se selectează automat următorul server edge cel mai apropiat. Ciclul continuă până când se identifică un server capabil să proceseze cererea în mod eficient sau, în situația în care toate serverele sunt supraîncărcate, se returnează rezultatul None, indicând indisponibilitatea serverelor. Integrația cu serviciile AWS, prin intermediul bibliotecii *boto3*, permite interogarea detaliată a instanțelor EC2 pentru a monitoriza starea acestora, inclusiv utilizarea CPU-ului.

Dacă un domeniu nu este regăsit în baza de date, serverul DNS dinamic inițiază o interogare fallback către serverul DNS al sistemului de operare de pe instanța EC2 pe care rulează. Acest mecanism de fallback asigură că, în cazul în care informațiile necesare nu sunt disponibile în baza de date a serverului DNS dinamic, cererea DNS este redirecționată către un server DNS local de rezervă.

Serverul folosește un executor de tip *ThreadPoolExecutor* pentru a gestiona cererile în paralel. Executorul este inițializat cu un număr maxim de lucrători (thread-uri) care pot procesa cererile simultan. Această implementare îl face capabil să răspundă rapid la cererile utilizatorilor, asigurând o distribuție optimă a traficului către serverele edge.

Serverul DNS poate răspunde prompt la cererile utilizatorilor, optimizând experiența acestora prin reducerea latenței și îmbunătățirea accesibilității conținutului.

3.5. Filtru DDoS

Detector DDoS

Implementarea unui detector DDoS implică monitorizarea traficului de rețea și identificarea tiparelor neobișnuite care ar putea indica un atac.

Detectarea începe prin configurarea unui client SQS pentru a primi mesaje despre cererile HTTP cu ajutorul bibliotecii *boto3*. Mesajele conțin informații despre cererea HTTP, inclusiv adresa IP a clientului (client_ip), domeniul solicitat (domain), identificatorul instanței serverului edge (instance_id), timpul de răspuns (response_time) și timpul la care a fost primită cererea (time).

Mesajele sunt stocate și analizate pe ferestre de timp de un minut. Pentru fiecare cerere, se utilizează câmpul „time” din mesaj pentru a determina intervalul de timp corespunzător. Cererile sunt apoi stocate separat pentru fiecare domeniu și pentru fiecare client. În acest fel, se poate analiza traficul pe fiecare domeniu și se poate detecta dacă un anumit client trimite un număr

neobișnuit de mare de cereri. Această metodă de stocare permite gruparea cererilor în intervale specifice de timp, ceea ce facilitează identificarea tiparelor anormale de trafic.

Funcția *calculate_means*, Codul 3.10, are rolul de a calcula media numărului de cereri atât pentru fiecare domeniu, cât și pentru fiecare client.Valorile sunt apoi utilizate pentru a identifica posibile anomalii în trafic.

```
1   def calculate_means(domain_request_counts, client_request_counts):
2       total_domains = len(domain_request_counts)
3       total_requests = sum(domain_request_counts.values())
4       mean_per_domain = total_requests / total_domains if total_domains
        ↪  > 0 else 0
5
6       total_clients = sum(len(clients) for clients in
        ↪  client_request_counts.values())
7       total_requests_per_client = sum(sum(clients.values()) for clients
        ↪  in client_request_counts.values())
8       mean_per_client = total_requests_per_client / total_clients if
        ↪  total_clients > 0 else 0
9
10      print(f"Total domains: {total_domains}, Total clients:
        ↪  {total_clients}")
11      print(f"Mean per domain: {mean_per_domain}")
12      print(f"Mean per client: {mean_per_client}")
13
14      return mean_per_domain, mean_per_client
```

Listing 3.10. Funcția calculate_means

Media cererilor per domeniu (*mean_per_domain*)(Linia 4) se calculează folosind formula:

$$\text{mean\_per\_domain} = \frac{\sum_{i=1}^{n} \text{requests\_domain}_i}{n}$$

unde n este numărul total de domenii.

Media cererilor per client (*mean_per_client*)(Linia 8):

$$\text{mean\_per\_client} = \frac{\sum_{j=1}^{m} \sum_{k=1}^{c_j} \text{requests\_client}_{j,k}}{m}$$

unde m este numărul total de domenii cu clienți, iar c_j este numărul de clienți pentru domeniul j.

Funcția *analyze_requests*, Codul 16, are rolul de a analiza cererile primite de la diverse domenii și clienți pentru a detecta posibile atacuri DDoS. Aceasta folosește metode statistice pentru a identifica anomalii în traficul de rețea, cum ar fi numărul excesiv de cereri de la un singur client sau către un singur domeniu. În cazul detectării unui astfel de atac, funcția trimite cereri de blocare către un manager de banare pentru a preveni continuarea atacului.

Funcția identifică și listează IP-urile suspicioase și posibilele IP-uri de botnet, oferind indicii asupra sursei atacului. IP-urile care fac parte din botnet sunt identificate prin analiza numărului de cereri per domeniu și per client. Un botnet este indicat atunci când un domeniu are un număr mare de clienți unici, fiecare trimițând un număr semnificativ de cereri. Aceasta sugerează că traficul provine de la o rețea de computere controlate de atacatori, care colaborează pentru a inunda serverul cu cereri și a provoca un atac DDoS.

Când se detectează un potențial atac DDoS, se trimite o cerere către un serviciu de management al interdicțiilor (ban manager) pentru a notifica serverele edge despre IP-urile suspicioase.

Acest serviciu gestionează blocarea IP-urilor suspecte, asigurând protecția rețelei împotriva atacurilor continue. Notificarea trimisă către ban manager permite serverelor edge să reacționeze rapid și să prevină accesul ulterioar al acestor IP-uri, minimizând impactul atacului asupra resurselor și utilizatorilor legitimi.

Procesul de detectare a atacurilor DDoS utilizează două dicționare Python esențiale, *domain_request_counts*, pentru a stoca numărul total de cereri per domeniu, și *client_request_counts*, pentru a stoca numărul total de cereri per client per domeniu. Funcția *analyze_requests* iterează prin cererile stocate în domain_requests și agregă numărul de cereri pentru fiecare domeniu și client (Liniile 18 și 36). Dacă numărul total de cereri este sub un anumit prag (150), funcția nu actualizează mediile globale, deoarece nu există suficiente date pentru o analiză corectă.

Funcția verifică pentru fiecare domeniu dacă numărul de cereri depășește media globală plus jumătate din această medie. Dacă această condiție este îndeplinită, se detectează un posibil atac DDoS. În plus, dacă un domeniu are mai mult de 9 clienți unici și numărul de cereri per client depășește un prag specific (mean_per_client / 10), se detectează un posibil atac de tip botnet (Linia 29).

Funcția verifică, de asemenea, pentru fiecare client dacă numărul de cereri depășește media globală plus jumătate din această medie. Acest lucru ajută la identificarea atacurilor DDoS individuale provenite de la clienți specifici. În final, funcția *analyze_requests* calculează și actualizează mediile globale pentru cereri per domeniu și per client folosind funcția *calculate_means*, asigurând astfel actualizarea continuă a acestor valori pentru detectarea anomaliilor în traficul de rețea.

Biblioteca *asyncio* este utilizată pentru a gestiona operațiile asincrone, permițând funcției să proceseze mesajele din coada SQS și să trimită cereri de banare către serviciul de management al interdicțiilor fără a bloca execuția programului. Biblioteca *math* este folosită pentru a efectua calculele mediilor. În plus, biblioteca *httpx* este utilizată pentru a trimite cereri HTTP asincron către serviciul de management al interdicțiilor (ban manager).

Ban Manager

Ban Managerul este o componentă responsabilă pentru gestionarea blocării și deblocării adreselor IP pe multiple servere edge. Această implementare utilizează FastAPI pentru a expune API-uri care permit administrarea blocării și deblocării IP-urilor prin intermediul unui endpoint HTTP POST.

Funcția *get_instance*, Codul 15, este responsabilă pentru identificarea tuturor serverelor edge active (pornite). Aceasta funcționează prin interogarea bazei de date MongoDB pentru a găsi serverele care au statusul setat la *„ON”* sau *„SLOW”*, indicând că sunt active și disponibile pentru a procesa cereri. Funcția *send_action_to_all_servers*, Codul 14, utilizează informațiile furnizate de *get_instance* pentru a trimite cereri HTTP asincrone către fiecare server edge activ. Aceasta funcție permite trimiterea de cereri pentru a bloca sau debloca adrese IP pe toate serverele edge active.

Această structură permite administrarea eficientă a blocării și deblocării adreselor IP pe întreaga rețea de servere edge, asigurând securitatea și integritatea platformei CDN. Prin utilizarea cererilor HTTP asincrone, sistemul poate scala pentru a gestiona multiple servere edge, menținând performanța și reducând latența.

3.6. Platforma Web

Platforma web reunește toate serviciile prin intermediul unui gateway care servește drept punct central de comunicare între aplicația frontend și serviciile backend. Toate serviciile sunt încapsulate în containere Docker și sunt orchestrate folosind Docker Compose, formând o rețea unificată. Această arhitectură permite gestionarea și scalarea eficientă a componentelor. Serviciul de mail și gateway-ul sunt configurate pentru a fi vizibile din exterior, permițând accesul și interacțiunea directă cu utilizatorii, asigurând comunicarea fluidă și securizată între toate compo-

nentele platformei.

3.6.1. Serviciul pentru gestionarea utilizatorilor

Serviciul gRPC pentru gestionarea utilizatorilor oferă funcționalități esențiale precum autentificarea, verificarea token-urilor, înregistrarea utilizatorilor noi, ștergerea utilizatorilor existenți, actualizarea datelor utilizatorilor și obținerea datelor utilizatorilor. Acesta este implementat în Python și utilizează biblioteca *gRPC* pentru a oferi o interfață de comunicare eficientă și scalabilă între diversele componente ale platformei.

Serviciul pentru gestionarea utilizatorilor este construit prin definirea unui fișier ProtoBuffer, care specifică interfața serviciului, incluzând metodele și mesajele necesare pentru a realiza operațiuni esențiale de autentificare și gestionare a utilizatorilor. Pe baza acestei definiții ProtoBuffer, gRPC generează automat codul necesar pentru server și client. Acest proces produce două componente principale, *IDM_Service_pb2.py*, care conține definițiile claselor pentru mesajele ProtoBuffer, și *IDM_Service_pb2_grpc.py*, care include definițiile claselor și interfețelor pentru serviciul gRPC.

```
syntax="proto3";
service IDMService {
  rpc Login(LoginRequest) returns (LoginResponse);
  rpc VerifyToken(TokenRequest) returns (TokenResponse);
  rpc SignUp(SignUpRequest) returns (SignUpResponse);
  rpc DeleteUser(DeleteUserRequest) returns (DeleteUserResponse);
  rpc UpdateUser(UpdateUserRequest) returns (UpdateUserResponse);
  rpc GetUserData(GetUserRequest) returns (GetUserResponse);
  rpc GetUserMail(GetMailRequest) returns (GetMailResponse);
}
```

Listing 3.3: Interfața Serviciului

Clasa *UserService* este implementată prin moștenirea clasei *IDMServiceServicer* și suprascrierea metodelor definite în această clasă. Acest serviciu comunică cu baza de date MariaDB pentru a gestiona resursa utilizator, realizând operațiuni de autentificare, înregistrare, actualizare și ștergere a utilizatorilor.

Înregistrarea utilizatorilor se realizează prin funcția *SignUp*, care gestionează procesul de adăugare a noilor utilizatori în baza de date. Pentru securitate, parola este hash-uită și i se aplică un salt folosind algoritmul `bcrypt` înainte de a fi stocată. Identificatorul unic al utilizatorului este un *UUID* (*Universally Unique Identifier*) generat cu `uuid.uuid4()`. După înregistrarea utilizatorului, se generează un token JWT pentru a menține utilizatorul autentificat.

Funcționalitatea autentificării utilizatorilor este gestionată prin funcția *Login*, care validează datele de autentificare furnizate de utilizator. Funcția verifică dacă utilizatorul există în baza de date și compară parola furnizată cu cea stocată, utilizând `bcrypt` pentru verificarea hash-ului parolei. Dacă autentificarea este reușită, se generează un token JWT pentru a menține sesiunea utilizatorului.

Tokenul JWT este generat după ce un utilizator este autentificat cu succes, atât la login, cât și la înregistrare. Acesta conține informații despre utilizator și este semnat pentru a asigura integritatea și autenticitatea datelor. Un token este valabil timp de 2 ore și conține un payload cu următoarele informații:

- **iss (issuer)**: Identifică cine a emis tokenul, în acest caz URL-ul serviciului de autentificare.
- **username**: Numele de utilizator al utilizatorului autentificat.
- **role**: Rolul utilizatorului (de exemplu, admin, user).

- **sub (subject)**: ID-ul unic al utilizatorului.
- **exp (expiration)**: Data și ora la care tokenul va expira
- **jti (JWT ID)**: Un identificator unic pentru token pentru a preveni reutilizarea tokenurilor.

3.6.2. Serviciu de origine

Implementarea serviciului de gestionare a serverelor de origine utilizând *FastAPI* este centrată pe gestionarea operațiunilor *CRUD* (Create, Read, Update, Delete) și pe setarea modurilor de bypass cache pentru serverele de origine. Clientul gRPC este utilizat pentru verificarea token-urilor *JWT*. Sunt implementate mai multe endpoint-uri pentru a facilita diferite operațiuni asupra serverelor de origine, ilustrate în anexa 6.1.

Consistența datelor este asigurată prin utilizarea sesiunilor MongoDB pentru a gestiona tranzacțiile și a asigura că operațiunile atomice sunt fie completate, fie anulate în caz de eroare. În loc de a verifica local token-urile JWT, serviciul utilizează un canal gRPC pentru a comunica cu un serviciu extern de management al identității, asigurând astfel o securitate suplimentară și centralizarea autentificării.

Codul 3.11, gestionează autorizarea unei operațiuni într-un serviciu de gestionare a serverelor de origine utilizând FastAPI. Acesta utilizează token-uri JWT pentru a asigura accesul autorizat la diferite endpoint-uri. In linia 1 este obținut antetului de autorizare din cererea HTTP, dacă acest antet nu există sau nu începe cu `"Bearer "`, este returnată o eroare `401`. După obținerea token-ului din antetul de autorizare, acesta este trimis la un serviciu gRPC pentru verificare. Serviciul gRPC validează token-ul prin metoda *VerifyToken*, asigurându-se astfel că token-ul este valid.

Dacă token-ul este valid, se decodează payload-ul acestuia folosind o cheie secretă definită în aplicație (*SECRET_KEY*). Din payload se extrag informații precum *user_id* și *user_type*. În continuare, codul verifică dacă există un proprietar specificat și dacă acesta coincide cu *user_id* extras din token. Dacă proprietarul este diferit, se returnează o eroare `403 Forbidden access`, indicând că utilizatorul nu are permisiunea de a accesa resursa solicitată.

```
1  authorization_header = request.headers.get('Authorization')
2  if authorization_header is None or not
    ↪ authorization_header.startswith('Bearer '):
3      raise HTTPException(status_code=401, detail="Unauthorized
       ↪ access")
4
5  token = authorization_header.split(" ")[1]
6  grpc_stub = grpc_client()
7  response =
    ↪ grpc_stub.VerifyToken(Idm_service_pb2.TokenRequest(token=token))
8  if response.is_valid == "valid":
9      query = {}
10     payload = jwt.decode(token, SECRET_KEY, algorithms=["HS256"])
11     user_id = payload.get('sub')
12     user_type = payload.get('role')
13     if owner is not None:
14         //actiune
15     else:
16         raise HTTPException(status_code=403, detail="Forbidden
           ↪ access")
```

Listing 3.11. Autorizarea unei operațiuni

MongoDB este utilizat pentru stocarea și gestionarea datelor serverelor de origine și a planurilor acestora. Utilizând *pymongo*, conexiunile la baza de date sunt gestionate eficient, iar tranzacțiile sunt folosite pentru a asigura consistența datelor.

3.6.3. Serviciu de caching

Serviciul Cache Server, implementat utilizând FastAPI, gestionează operațiunile de acces și manipulare a datelor cache stocate în Redis. Acest serviciu expune endpoint-uri pentru vizualizarea și ștergerea cheilor din cache, asigurând astfel o gestionare eficientă a resurselor și o actualizare rapidă a datelor cache.

Serviciul utilizează un client Redis cu biblioteca *redis* pentru a se conecta la cache-ul global al platformei. Acesta oferă clienților posibilitatea de a vedea datele disponibile în cache și de a le gestiona în funcție de necesități. În cazul în care utilizatorul nu este mulțumit de prezența anumitor date în cache, acesta le poate șterge prin intermediul clientului Redis, dar numai dacă se dovedește că resursele îi aparțin.

Pentru a asigura securitatea, toate cererile trebuie să includă un token JWT valid. Serviciul utilizează gRPC pentru a verifica aceste token-uri printr-un serviciu extern de management al identității. Aceasta asigură că doar utilizatorii autorizați pot accesa sau manipula datele din cache.

Funcționalitățile expuse de acest serviciu sunt evidențiate în Anexa 6.2.

3.6.4. Serviciu de EdgeServere

Serviciul Edge Server Management este implementat utilizând FastAPI și oferă funcționalități pentru gestionarea serverelor edge, inclusiv operațiuni de creare, citire, actualizare și ștergere (CRUD). În plus, include funcționalități pentru gestionarea adreselor IP blocate și pentru obținerea metricilor de performanță ale serverelor edge utilizând AWS CloudWatch. Toate cererile trebuie să includă un token JWT valid, verificat printr-un serviciu gRPC. Utilizatorii trebuie să fie autentificați și să aibă rolul de admin pentru a accesa majoritatea funcționalităților.

MongoDB este utilizat pentru stocarea și gestionarea informațiilor despre serverele edge și adresele IP blocate, iar conexiunea este gestionată prin *pymongo*. gRPC este folosit pentru verificarea token-urilor JWT printr-un serviciu extern de management al identității, asigurând securitate suplimentară prin centralizarea autentificării. *jwt* este biblioteca folosită pentru decodarea și validarea token-urilor JWT, ajutând la autentificarea utilizatorilor și autorizarea accesului la resursele gestionate. boto3 este utilizat pentru obținerea metricilor de performanță ale instanțelor EC2 prin AWS CloudWatch, iar Uvicorn este serverul ASGI utilizat pentru a rula aplicația FastAPI.

Funcționalitățile expuse de acest serviciu sunt evidențiate Anexa 6.3..

3.6.5. Serviciu de mail

Serviciul Mail Server este utilizat pentru trimiterea notificărilor și este implementat în Python utilizând framework-ul FastAPI. Acest serviciu utilizează metode POST pentru a trimite emailuri de confirmare a creării contului și pentru notificarea problemelor la serverele de origine.

Metoda *send_email*, 3.12, este responsabilă pentru trimiterea unui email utilizând protocolul SMTP prin intermediul serverului de email Gmail. Această funcție primește ca parametri adresa de email a expeditorului, parola, adresa de email a destinatarului, subiectul emailului și corpul mesajului. Biblioteca *smtplib* este utilizată pentru a gestiona comunicarea cu serverul SMTP.

Funcția începe prin configurarea serverului SMTP Gmail și a portului acestuia (`587`) pentru conexiuni TLS. Este creat un context SSL utilizând biblioteca *ssl* pentru a asigura securitatea conexiunii. Mesajul email este construit folosind *email.mime.multipart.MIMEMultipart* și *email.mime.text.MIMEText* pentru a atașa corpul mesajului în format text simplu. Conectarea la serverul `SMTP` se realizează prin *smtplib.SMTP*, urmată de inițializarea conexiunii TLS pentru securitate. Autentificarea la serverul SMTP se efectuează folosind adresa de email a expeditorului și

parola acestuia. După autentificare, emailul este trimis utilizând metoda *sendmail* din *smtplib*.

```
1  def send_email(sender_email, password, receiver_email, subject,
   ↪ body):
2      smtp_server = "smtp.gmail.com"
3      port = 587
4      context = ssl.create_default_context()
5      message = MIMEMultipart()
6      message["From"] = sender_email
7      message["To"] = receiver_email
8      message["Subject"] = subject
9      message.attach(MIMEText(body, "plain"))
10     try:
11         server = smtplib.SMTP(smtp_server, port)
12         server.starttls(context=context)
13         server.login(sender_email, password)
14         server.sendmail(sender_email, receiver_email,
           ↪ message.as_string())
15     except smtplib.SMTPAuthenticationError as auth_error:
16         raise HTTPException(status_code=500,
17                             detail="SMTP authentication error. Please
                               ↪ check your email address and
                               ↪ password.")
18     except smtplib.SMTPException as smtp_error:
19         raise HTTPException(status_code=500, detail="SMTP error.
           ↪ Please check SMTP server settings.")
20     except Exception as e:
21         raise HTTPException(status_code=500, detail="Error sending
           ↪ email.")
22     finally:
23         server.quit()
```

Listing 3.12. Funcția send_mail

3.6.6. Componenta de analiză a traficului

Componenta de analiză a traficului din cadrul platformei CDN, implementată utilizând FastAPI, are scopul de a primi, procesa și transmite în timp real mesajele dintr-un AWS SQS către clienți prin intermediul WebSocket-urilor. Aceasta include funcționalități pentru extragerea locației geografice a clienților, trimiterea mesajelor prin WebSocket și gestionarea cererilor primite de edge servere din SQS.

Funcția *get_sqs_messages*, Codul 17, este responsabilă pentru primirea mesajelor din queue-ul AWS SQS. Utilizând biblioteca *boto3*, aceasta recepționează mesajele și le șterge din queue după ce au fost procesate.

```
{
    "message_id": "12345678-1234-1234-1234-1234567890ab",
    "client_ip": "1.2.3.4",
    "edge_server": "i-0123456789abcdef0",
    "domain": "example.com",
    "resource": "/index.html",
    "state": "success",
    "time": "2023-06-26T14:30:45.123456Z"
}
```

Listing 3.13: Structura mesajelor primite din AWS SQS

Endpoint-ul `/sqs-messages` definește un WebSocket care transmite mesajele primite de la SQS către client în timp real. Conexiunea WebSocket este menținută deschisă, iar mesajele sunt trimise periodic pentru a asigura actualizarea continuă a datelor.

Funcția *get_region* utilizează API-ul `ipinfo.io` pentru a obține locația geografică a unui client pe baza adresei IP. Aceasta asigură că mesajele includ informații precise despre locația geografică a clienților, îmbogățind astfel datele transmise.

Această componentă este esențială pentru vizualizarea cererilor din cadrul platformei CDN de către administratori. Aceștia pot observa de unde vin cererile și care servere edge le rezolvă, asigurând astfel o monitorizare și gestionare eficientă a traficului.

3.6.7. Componenta de frontend

Componenta de frontend este implementată în React și rulează pe portul `3000`. Aplicația oferă utilizatorilor o interfață interactivă și intuitivă. Aceasta utilizează biblioteca *Axios* pentru a trimite cereri HTTP către Gateway-ul platformei web, asigurând astfel comunicarea eficientă cu backend-ul. Prin intermediul *Axios*, componentele React pot obține și trimite date, permițând actualizarea stării în timp real și îmbunătățind experiența utilizatorilor. În combinație cu diverse hook-uri React, precum *useParams*, *useNavigate*, *useState*, și *useEffect*, componenta gestionează starea aplicației și efectuează cereri asincrone pentru a obține și actualiza datele.

La autentificarea unui utilizator, aplicația React stochează în local storage mai multe informații esențiale: ID-ul utilizatorului, tipul acestuia (user sau admin) și tokenul JWT. Aceste informații sunt importante pentru gestionarea sesiunii utilizatorului și pentru a controla accesul la diverse resurse și funcționalități ale aplicației.

În funcție de tipul utilizatorului stocat în local storage, aplicația React oferă acces la pagini web diferite. Utilizatorii obișnuiți și administratorii au acces la seturi diferite de funcționalități și resurse.

Pentru fiecare cerere HTTP trimisă către serverul backend, tokenul JWT este inclus în antetul cererii pentru a asigura autentificarea și autorizarea corectă a utilizatorului. Aceasta permite serverului să verifice identitatea utilizatorului și să determine dacă acesta are drepturi suficiente pentru a accesa resursa solicitată.

```
headers: {
    Authorization: `Bearer ${token}`,
}
```

Listing 3.14. Header Authorization

ID-ul utilizatorului stocat în local storage este utilizat pentru a solicita resurse specifice utilizatorului de la backend. Acest ID permite serverului să returneze date personalizate pentru utilizatorul respectiv, cum ar fi profilul sau site-urile înregistrate.

Pentru a preveni atacurile de tip Cross-Site Scripting (XSS), aplicația React implementează un mecanism de validare a datelor de intrare. Acest mecanism utilizează o expresie regulată pentru a interzice caracterele sensibile precum „<”, „>”, „&” și „#”. În 3.15, se asigură că aceste caracter nu sunt permise în datele de intrare, prevenind astfel posibilele exploatări de tip XSS și protejând integritatea aplicației și securitatea utilizatorilor.

```
1   const sensitiveCharsRegex = /[<>&#]/;
2       if (sensitiveCharsRegex.test(instance_id)) {
3         setError("Datele nu poate contine caractere sensibile (<, >, &
            ↪  sau #).");
4         return;
5       }
6       if (sensitiveCharsRegex.test(region)) {
7         setError("Datele nu poate contine caractere sensibile (<, >, &
            ↪  sau #).");
8         return;
9       }
```

Listing 3.15. Verificare XSS

Capitolul 4. Testarea aplicației și rezultate experimentale

Pentru a demonstra funcționalitatea platformei, am utilizat site-uri web dezvoltate în cadrul altor proiecte universitare. Aceste site-uri, construite în NodeJS și reprezentând magazine online, au fost containerizate și încărcate pe instanțe EC2, ceea ce le permite să fie accesibile în rețea prin intermediul adreselor IP ale instanțelor respective. Magazinele online includ resurse statice, cum ar fi fișiere CSS, JS, imagini și videoclipuri. Această abordare asigură că site-urile pot fi accesate și testate eficient în mediul de rețea configurat pentru platformă, demonstrând capacitatea de a gestiona și securiza traficul web în condiții reale.

Pentru simularea utilizatorului, am folosit o mașină virtuală Ubuntu configurată pentru a acționa ca un server DNS. În această configurație, serverul DNS a fost setat să redirecționeze cererile către serverele edge din platforma noastră. Figura 4.1 ilustrează faptul că mașina virtuală a fost configurată cu adresa IP 54.93.223.140, care este adresa IP a serverului DNS.

```
1   cosmin@cosmin-virtual-machine:~/Desktop$ cat /etc/resolv.conf
2       nameserver 54.93.223.140 //ip server dns
3       nameserver 127.0.0.53
```

Listing 4.1: Configurarea fișierului /etc/resolv.conf

Pentru testele următoare se vor utiliza trei edge servere. Cel mai apropiat server este localizat în Frankfurt și are adresa IP 3.74.161.3. Următorul server este situat în Londra, având adresa IP 13.42.105.34. Al treilea server se află în Ohio, cu adresa IP 3.14.245.69.

Pentru a simula clientul, am creat un cont pe platforma web folosind următoarele date: username CosminSergiu, numele de familie Sergiu, prenumele Cosmin, telefon 0727892345, Email: cosmin-sergiu@gmail.com și țara: România.

În cadrul acestui utilizator, am încărcat două site-uri web care urmează să fie deservite de platforma CDN. Site-ul www.idoc.ro a fost configurat pentru a implementa planul Enterprise, iar site-ul www.ielectro.ro a fost configurat pentru a implementa planul Basic. În figura 4.3 se observă diferențele dintre planurile alese, cum ar fi dimensiunea maximă a fișierului permisă sau absența Modul Dezvoltare pentru site-ul www.ielectro.ro. Aceste diferențe evidențiază avantajele și limitările fiecărui plan în contextul gestionării și optimizării traficului web.

Figura 4.1. Site-urile inregistrate

Se observă că adresele IP configurate pentru site-uri sunt identice atunci când se execută comanda nslookup asupra celor două domenii. În 4.1, se poate vedea că mașina virtuală nu returnează adresa IP a serverelor de origine, ci aceeași adresă IP pentru ambele domenii. Aceasta se datorează faptului că, datorită locației clientului, cererile vor fi deservite de același edge server.

```
cosmin@cosmin-virtual-machine:~/Desktop$ nslookup www.idoc.ro
    Server:    54.93.223.140
    Address:  54.93.223.140#53

    Name:   www.idoc.ro
    Address: 3.74.161.3

cosmin@cosmin-virtual-machine:~/Desktop$ nslookup www.ielectro.ro
    Server:    54.93.223.140
    Address:  54.93.223.140#53

    Name:   www.ielectro.ro
    Address: 3.74.161.3
```

Listing 4.2: nslookup în mașina virtuală

La următoarea cerere nslookup, o vom executa pe o instanță EC2 care este în cloudul AWS, pe o mașină virtuală situată în California. Acest experiment are rolul de a verifica modul în care cererile DNS sunt redirecționate către edge serverele cele mai apropiate de locația geografică a clientului. Se observă că adresa IP s-a schimbat pentru domenii, dar a rămas identică pentru ambele domenii. Aceasta indică faptul că, deși platforma a selectat un alt edge server datorită locației diferite a clientului, ambele domenii sunt deservite de același server edge din acea regiune.

```
ubuntu@ip-172-31-19-226:~$ nslookup www.idoc.ro 54.93.223.140
    Server:    54.93.223.140
    Address:  54.93.223.140#53

    Name:   www.idoc.ro
    Address: 3.14.245.69

ubuntu@ip-172-31-19-226:~$ nslookup www.ielectro.ro 54.93.223.140
    Server:    54.93.223.140
    Address:  54.93.223.140#53

    Name:   www.ielectro.ro
    Address: 3.14.245.69
```

Listing 4.3: nslookup în ec2 California

Pe mașina virtuală configurată, vom utiliza browserul Mozilla Firefox pentru a trimite o cerere către domeniul www.idoc.ro. Observăm că primim un răspuns de la server, confirmând că cererea a fost procesată cu succes și că serverul edge selectat pentru această regiune funcționează corect, furnizând conținutul solicitat. Aceasta demonstrează capacitatea platformei CDN de a redirecționa eficient cererile în funcție de locația geografică a clientului și de a deservi conținutul web prin intermediul serverelor edge. În figura 4.3, se observă conținutul livrat de serverul de origine prin adresa IP și conținutul livrat prin intermediul edge serverelor. Acest conținut este identic, evidențiind consistența platformei CDN în distribuirea conținutului.

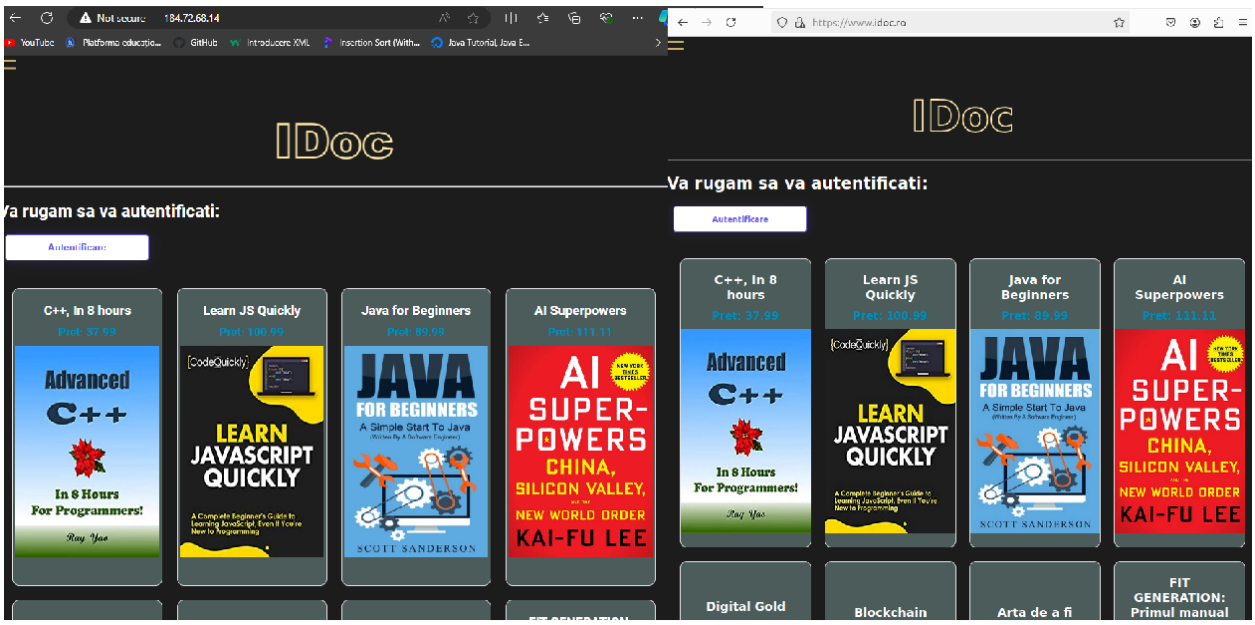

Figura 4.2. www.idoc.ro

Pentru a verifica capacitatea platformei de a observa și reacționa la indisponibilitatea serverului de origine, serverul de origine a fost oprit și s-a trimis o cerere către domeniul www.idoc.ro. După trimiterea cererii, se observă că răspunsul primit conține codul de eroare 521. Acest cod indică faptul că serverul web de origine este inaccesibil, ceea ce demonstrează eficiența sistemului de monitorizare al platformei în identificarea și raportarea problemelor legate de disponibilitatea serverelor de origine.

Figura 4.3. www.idoc.ro

Pe partea de client a platformei, se trimite un e-mail automat către proprietarul site-ului în momentul în care serverul de origine devine indisponibil. Acest e-mail conține informații detaliate despre problema întâmpinată, precum și recomandări pentru remedierea rapidă a acesteia. Această funcționalitate asigură o comunicare eficientă între platformă și proprietarii de site-uri, permițându-le acestora să fie informați în timp real despre orice problemă care ar putea afecta accesibilitatea și performanța site-ului lor.

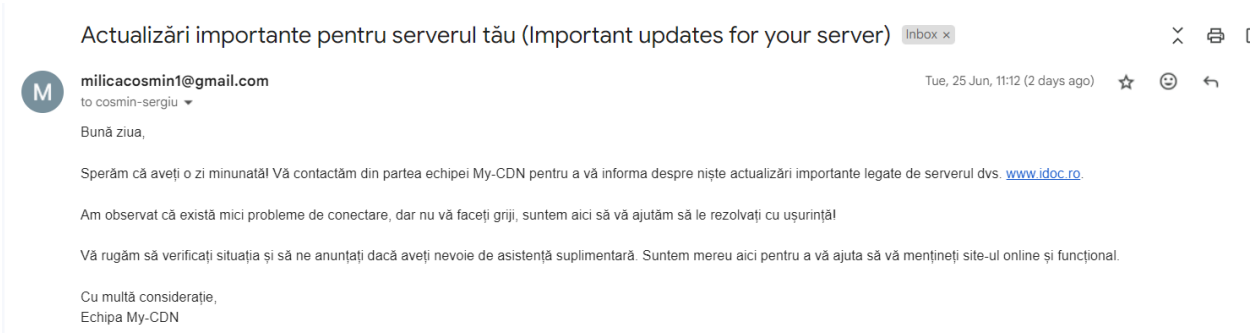

Figura 4.4. Email către owner

Pentru a testa funcționalitatea serverului DNS și capacitatea acestuia de a detecta inactivitatea unui server, serverul edge din Frankfurt va fi oprit, iar apoi se va executa comanda nslookup pe mașina virtuală. În Listing 4.6, se poate observa că domeniul este acum deservit de edge serverul din Londra, acesta fiind cel mai apropiat de client.

```
1  cosmin@cosmin-virtual-machine:~/Desktop$ nslookup www.idoc.ro
2       Server:     54.93.223.140
3       Address:  54.93.223.140#53
4
5       Name:  www.idoc.ro
6       Address: 13.42.105.34
```

Listing 4.4: nslookup după oprirea serverului din Frankfurt

Pentru a testa filtrul DDoS al platformei, se utilizează comanda `ab -n 1000 -c 100 https://www.idoc.ro/stil.css`. Această comandă utilizează ApacheBench (ab) pentru a trimite 1000 de cereri concurente (simultane) la resursa specificată (stil.css) de pe site-ul www.idoc.ro, cu o concurență de 100. ApacheBench este un instrument de benchmarking care ne permite să simulăm un volum mare de trafic către server pentru a testa performanța acestuia sub sarcină și pentru a observa cum gestionează cererile multiple simultan.

În timpul testului, ApacheBench a reușit să finalizeze 600 de cereri înainte de a începe să raporteze erori de închidere a conexiunii SSL (SSL read failed). Acest lucru indică faptul că serverul a început să refuze conexiunile, cel mai probabil ca răspuns la detecția unui posibil atac DDoS.

Partea dreaptă a imaginii arată că scriptul de filtrare DDoS a detectat acest volum mare de trafic ca fiind un posibil atac DDoS. Detectează un atac DDoS potențial de la clientul cu IP-ul 79.112.79.141 pe domeniul www.idoc.ro cu un număr total de 471 de cereri pe minut. Detectorul de DDoS a trimis cerere către ban manager, care a trimis la rândul său cererea către edge servere și se observă că clientul a fost banat instant. Acest lucru indică faptul că platforma a fost capabilă să identifice rapid și să reacționeze eficient la un volum mare de trafic suspect, protejând astfel resursele serverului.

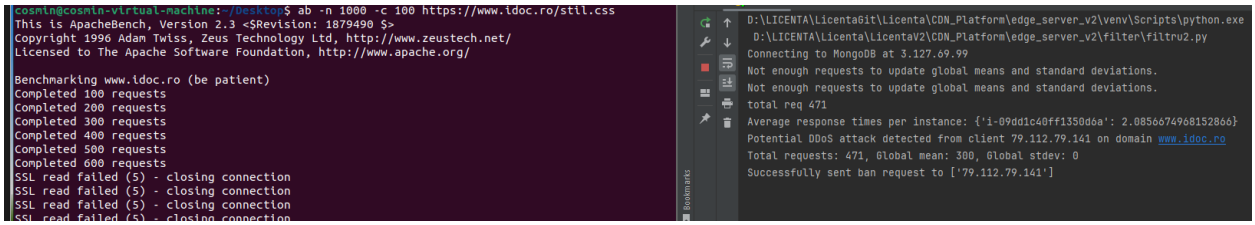

Figura 4.5. Utilizarea filtrului DDoS

După ce am trimis cererile, am accesat din nou browserul și am observat că nu se mai poate stabili conexiunea cu acel domeniu. Acest lucru confirmă faptul că IP-ul a fost blocat cu succes de către edge servere, prevenind astfel accesul acestuia la resursele site-ului www.idoc.ro. Blocarea automată și rapidă a IP-urilor suspecte demonstrează eficacitatea sistemului de filtrare DDoS implementat, asigurând protecția continuă a resurselor și menținerea performanței serverelor în fața atacurilor cibernetice.

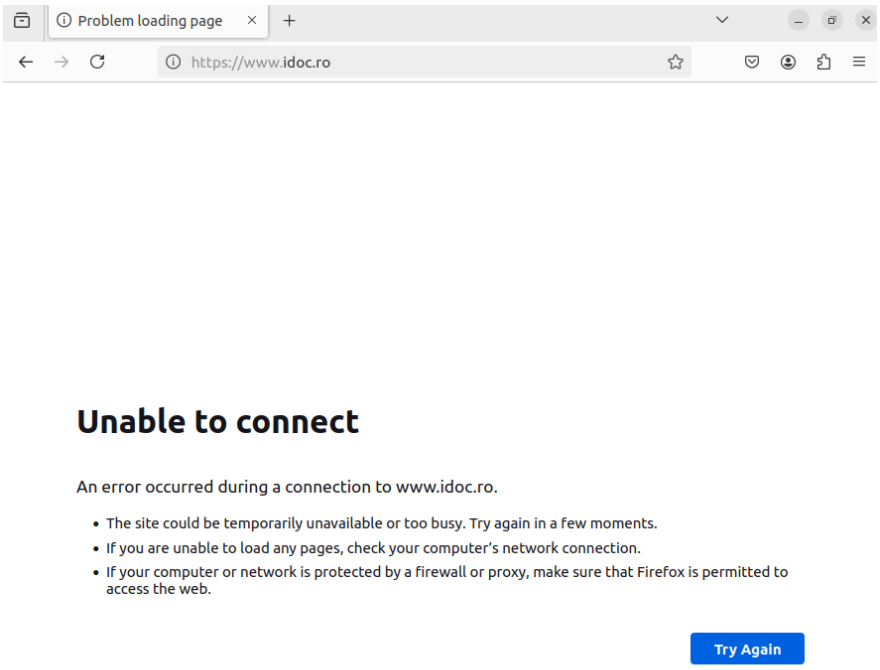

Figura 4.6. Rezultat banare

Pentru a testa componenta de monitorizare a sănătății serverelor edge, dezactivăm filtrul DDoS. Inițial, pornim două servere edge și verificăm statusul acestora în interfața grafică, conform ilustrației.

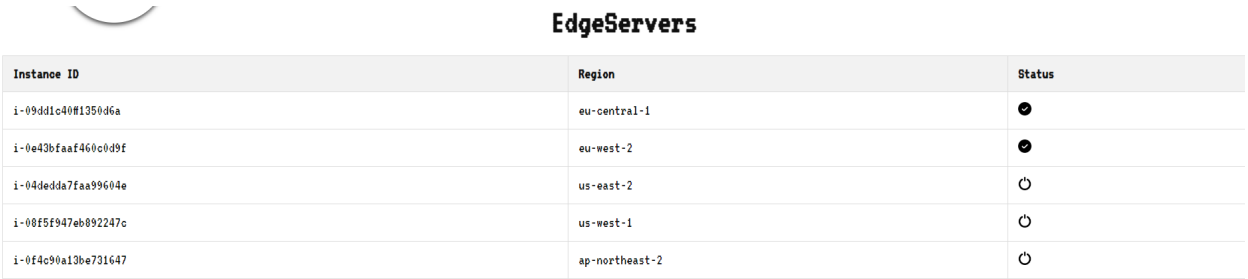

Figura 4.7. Status ON

Executăm nslookup pe mașina virtuală și observăm că cererile sunt direcționate către IP-ul 3.74.161.3. Pentru a încărca un server, simulăm trafic similar testelor pentru filtrul DDoS și activăm componenta de monitorizare a sănătății.

```
cosmin@cosmin-virtual-machine:~/Desktop$ nslookup www.idoc.ro
    Server:     54.93.223.140
    Address:  54.93.223.140#53

    Name:   www.idoc.ro
    Address: 3.74.161.3
```

Listing 4.5: nslookup înainte de CRASH

În figura 4.8, observăm că serverul care a gestionat traficul a fost marcat ca „CRASH” deoarece traficul simulat a afectat capacitatea sa de a răspunde la cererile componentei de monitorizare a sănătății.

Instance ID	Region	Status
i-09dd1c40ff1350d6a	eu-central-1	
i-0e43bfaaf460c0d9f	eu-west-2	
i-04dedda7faa99604e	us-east-2	
i-08f5f947eb892247c	us-west-1	
i-0f4c90a13be731647	ap-northeast-2	

Figura 4.8. Status CRASH

Pentru a verifica dacă platforma poate răspunde, vom executa din nou un nslookup. Observăm că serverul DNS redirecționează cererile către un alt server edge, cu IP-ul 13.42.105.34. Acest test demonstrează că platforma poate răspunde la cereri chiar dacă un server este afectat de trafic intens și devine inoperabil.

```
cosmin@cosmin-virtual-machine:~/Desktop$ nslookup www.idoc.ro
    Server:     54.93.223.140
    Address:  54.93.223.140#53

    Name:  www.idoc.ro
    Address: 13.42.105.34
```

Listing 4.6: nslookup după CRASH

După efectuarea testelor, se poate concluziona că platforma demonstrează o capacitate robustă de gestionare a traficului și de securizare a acestuia. Sistemul CDN reușește să redirecționeze cererile eficient către serverele edge în funcție de locația geografică a utilizatorului, asigurând o distribuție uniformă și rapidă a conținutului. Funcționalitatea de monitorizare a sănătății serverelor și filtrul DDoS au reacționat corespunzător, protejând resursele și menținând performanța în fața traficului intens și a posibilelor atacuri.

Concluzii

Prezenta lucrare a avut ca obiectiv dezvoltarea și implementarea unei platforme CDN capabile sa optimizeze livrarea continutului si securitatea site-urilor web. Lucrarea a demonstrat cu succes capacitatea platformei de a distribui eficient conținut web prin intermediul serverelor edge, adaptându-se la locația geografică a utilizatorilor pentru a optimiza performanța și a reduce latența. Prin integrarea serviciilor cloud AWS, am reușit să construim o platformă scalabilă și flexibilă, capabilă să răspundă cerințelor dinamice ale utilizatorilor.

Testele au arătat că platforma poate servi cererile în mod eficient, direcționându-le către serverele edge cele mai apropiate din punct de vedere geografic. Aceasta îmbunătățește timpii de răspuns și optimizează livrarea conținutului.

Platforma a demonstrat capacitatea de a direcționa eficient traficul către cel mai apropiat server edge față de client. Aceasta asigură timpi de răspuns optimizați și o experiență de utilizare îmbunătățită. Atunci când un server edge a devenit inoperabil sau a fost oprit, platforma a redirecționat automat cererile către un alt server disponibil. Acest mecanism de failover asigură continuitatea serviciilor și minimizarea impactului asupra utilizatorilor finali.

Filtrul DDoS a fost capabil să detecteze și să răspundă prompt la un volum mare de trafic suspect, identificând și blocând IP-urile potențial periculoase. Aceasta protejează resursele platformei și menține performanța în fața atacurilor cibernetice.

Sistemul de monitorizare a sănătății serverelor edge a funcționat corect, identificând serverele supraîncărcate sau inoperabile și redirecționând cererile către servere alternative. Acest lucru demonstrează capacitatea platformei de a menține un nivel ridicat de disponibilitate și performanță.

Prin utilizarea serviciilor AWS și a containerizării aplicațiilor, platforma poate fi scalată rapid pentru a răspunde cerințelor crescânde ale utilizatorilor. Această flexibilitate permite adaptarea rapidă la schimbările din mediul online și asigură disponibilitatea continuă a serviciilor, indiferent de volumul de trafic.

Direcții viitoare de dezvoltare

Direcțiile viitoare de dezvoltare includ optimizarea algoritmilor de detecție a atacurilor și extinderea funcționalităților pentru a răspunde mai bine nevoilor utilizatorilor.

În viitor, platforma va integra și resurse HTML pentru a spori eficiența livrării conținutului. De asemenea, se dorește utilizarea resurselor dinamice pentru a oferi o experiență optimizată și personalizată utilizatorilor finali. Prin detectarea conținutului static din resursele dinamice, de exemplu, o pagină HTML care prezintă un cronometru ce se actualizează la fiecare secundă, se pot trimite cereri către serverul de origine doar pentru cronometru, în timp ce restul paginii va fi returnată din cache.

Pe partea de securitate, se dorește implementarea unui sistem de cozi, câte una pentru fiecare domeniu, pentru a detecta mai rapid atacurile de tip DDoS și pentru a putea răspunde într-un mod mai eficient și precis. Aceasta va permite o analiză mai detaliată a traficului specific fiecărui domeniu și va facilita identificarea rapidă a surselor de trafic suspect. De asemenea, se intenționează banarea adreselor IP pe o durată de timp variabilă, astfel încât, cu cât persistă atacurile din partea unui atacator, cu atât să se crească timpul de banare.

Pe partea de monitorizare, se dorește implementarea unui sistem care să ofere informații în timp real despre starea serverelor și timpii de răspuns la cereri. Acest sistem va include multiple componente și funcționalități pentru a asigura performanța și securitatea rețelei de livrare a conținutului. Metoda va permite distribuirea cererilor pe baza timpilor de răspuns anteriori ai serverelor edge, facilitând astfel răspunsuri cât mai rapide.

Lecții învățate pe parcursul dezvoltării lucrării de diplomă

Pe parcursul dezvoltării acestei lucrări, am dobândit cunoștințe despre sistemele distribuite și cloud computing, inclusiv despre funcționarea serverelor DNS, cozilor de mesaje și load balancing. Unul dintre aspectele cheie pe care le-am explorat a fost fluxul unei cereri într-un mediu online. Am învățat cum o cerere este direcționată către serverul DNS, cum este procesată și cum se asigură livrarea conținutului către utilizatorul final. Acest proces implică multiple etape, de la rezolvarea domeniului până la distribuirea încărcării pe servere edge și optimizarea timpilor de răspuns prin caching.

De asemenea, am învățat despre tehnologia CDN și cum aceasta contribuie la îmbunătățirea performanței livrării conținutului. CDN-urile reduc latența și cresc viteza de încărcare a paginilor web prin distribuirea conținutului pe multiple servere localizate geografic aproape de utilizatorii finali.

Un alt aspect important pe care l-am învățat a fost dezvoltarea unei arhitecturi capabile să gestioneze cererile în mod dinamic, o arhitectură care se poate scala pe orizontală pentru a satisface cerințele crescânde.

Bibliografie

[1] Amazon Web Services. https://aws.amazon.com/.

[2] Google Cloud CDN. https://cloud.google.com/cdn?hl=ro.

[3] Docker. https://docs.docker.com/.

[4] MongoDB. https://www.mongodb.com/.

[5] MariaDB. https://mariadb.org/.

[6] Uvicorn. https://www.uvicorn.org/.

[7] React. https://react.dev/.

[8] FastAPI. https://fastapi.tiangolo.com/.

[9] Redis. https://redis.io/.

[10] Redis - LRU CACHE. https://redis.io/glossary/lru-cache/.

[11] What is a CDN edge server? https://www.cloudflare.com/learning/cdn/glossary/edge-server. 2024.

[12] What is load balancing? | How load balancers work https://www.cloudflare.com/learning/performance/what-is-load-balancing/.

[13] What is DNS-based load balancing? https://www.cloudflare.com/learning/performance/what-is-dns-load-balancing/.

[14] What is a DDoS attack? https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/.

[15] Cloudflare CDN. https://developers.cloudflare.com/cache/.

[16] Google Cloud CDN. https://cloud.google.com/cdn/docs/.

[17] Chapter 11 - Managing Load https://sre.google/workbook/managing-load/.

[18] What is a content delivery network (CDN)? https://www.cloudflare.com/learning/cdn/what-is-a-cdn.

[19] What is DNS? | How DNS works. https://www.cloudflare.com/learning/dns/what-is-dns/.

[20] IPinfo. https://ipinfo.io/.

[21] What is a WAF? | Web Application Firewall explained. https://www.cloudflare.com/learning/ddos/glossary/web-application-firewall-waf.

[22] Archip, A. et al. "RESTful Web Services - A Question of Standards". În: Proceedings Of The 22nd International Conference On System Theory, Control And Computing (ICSTCC). 2018, pp. 677-682. DOI: https://doi.org/10.1109/ICSTCC.2018.8540763.

[23] Fielding, R. T. "Architectural styles and the design of network-based software architectures". Teză de doctorat. Universitatea California, Irvine, 2000.

[24] Zeng, D. et al. "Efficient web content delivery using proxy caching techniques". În: IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 34.3 (2004), pp. 270-280. DOI: 10.1109/TSMCC.2004.829261.

[25] Mulerikkal, J. P. și Khalil, I. "An Architecture for Distributed Content Delivery Network". În: 2007 15th IEEE International Conference on Networks, 2007, pp. 359-364. DOI: 10.1109/ICON.2007.4444113.

[26] Bakonyi, P. et al. "Classification Based Load Balancing in Content Delivery Networks". În: 2020 43rd International Conference on Telecommunications and Signal Processing (TSP), 2020, pp. 621-626. DOI: 10.1109/TSP49548.2020.9163470.

[27] Ghaznavi, M. et al. "Content Delivery Network Security: A Survey". În: IEEE Communications Surveys Tutorials 23.4 (2021), pp. 2166-2190. DOI: 10.1109/COMST.2021.3093492.

[28] Hasslinger, G. et al. "Comparing Web Cache Implementations for Fast O(1) Updates Based on LRU, LFU and Score Gated Strategies". În: 2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), pp. 1-7. DOI: 10.1109/CAMAD.2018.8514951.

Anexe

Anexa 1. EdgeServer

```
async def fetch_and_cache(request, resource_key, site_property, plan_detail):
    header = {k: v for k, v in request.headers.items() if k.lower() not in ["content-range", "range"]}
    response = await http_client.request(
        request.method, PROTOCOL + site_property['ip_origin'] + request.url.path, headers=header
    )
    content = response.content
    headers = {k: v for k, v in response.headers.items() if k.lower() not in ["cache-control"]}

    if response.status_code == 304:
        return Response(status_code=304)

    if "ETag" in response.headers:
        db.resources.insert_one({'resource': resource_key, 'eTag': headers['etag'], 'timestamp': datetime.now()})

    cache_control = response.headers.get('cache-control', '')

    if 'no-store' in cache_control or 'no-cache' in cache_control:
        headers['Cache-Control'] = cache_control
    else:
        headers['Cache-Control'] = f'public, max-age=10'
    headers['X-Source'] = site_property['ip_origin']
    size = response.headers['content-length']

    if bytes_to_megabytes(size) <= int(plan_detail.get("file_size", '')):
        content_cache = base64.b64encode(content)
        try:
            cache.set(resource_key, content_cache)
            r.set(resource_key, content_cache, ex=site_property['time_cache'])
        except Exception as e:
            print('Eroare la redis cache')
    if 'content-encoding' not in headers:
        content_compressed = compress_with_gzip(content)
        content = content_compressed
        headers['Content-Encoding'] = 'gzip'
        if 'content-length' in headers:
            headers.pop('content-length')

    return Response(content=content, status_code=response.status_code, headers=headers)
```

Listing 7. Funcția pentru extragerea datelor de la origine

```
def setup_logging(log_file='app.log'):
    logger = logging.getLogger('edge_server')
    logger.setLevel(logging.DEBUG)

    formatter = logging.Formatter('%(asctime)s - %(name)s -
    ↪  %(levelname)s - %(message)s')
    file_handler = logging.handlers.RotatingFileHandler(
        log_file, maxBytes=5*1024*1024, backupCount=2
    )
    file_handler.setFormatter(formatter)

    logger.addHandler(file_handler)

    return logger
```

Listing 8. Funcția de logging

```
from datetime import datetime, timedelta, timezone
from cryptography import x509
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.x509.oid import NameOID
import os

from utile.mongo_connection import connect_to_mongodb

db, client = connect_to_mongodb()
document = db.origin.find({}, {"domain": 1, "_id": 0})
domains_list = []
if document:
    for doc in document:
        domain = doc.get("domain")
        if domain:
            domains_list.append(domain)
def generate_proxy_certificate(proxy_dir, domains):

    key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )

    key_file = os.path.join(proxy_dir, "proxy.key")
    with open(key_file, "wb") as f:
        f.write(key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.TraditionalOpenSSL,
            encryption_algorithm=serialization.NoEncryption()
        ))

```

```
35      subject = issuer = x509.Name([
36          x509.NameAttribute(NameOID.COMMON_NAME, domains[0]),
37          x509.NameAttribute(NameOID.ORGANIZATION_NAME, "CosminCDN")
38      ])
39
40      alt_names = [x509.DNSName(domain) for domain in domains]
41      san = x509.SubjectAlternativeName(alt_names)
42
43      cert = x509.CertificateBuilder().subject_name(
44          subject
45      ).issuer_name(
46          issuer
47      ).public_key(
48          key.public_key()
49      ).serial_number(
50          x509.random_serial_number()
51      ).not_valid_before(
52          datetime.now(timezone.utc)
53      ).not_valid_after(
54          datetime.now(timezone.utc) + timedelta(days=365)
55      ).add_extension(
56          san, critical=False
57      ).sign(
58          private_key=key,
59          algorithm=hashes.SHA256(),
60          backend=default_backend()
61      )
62
63      cert_file = os.path.join(proxy_dir, "proxy.crt")
64      with open(cert_file, "wb") as f:
65          f.write(cert.public_bytes(serialization.Encoding.PEM))
66
67      return key_file, cert_file
68
69
70  generate_proxy_certificate(".", domains_list)
```

Listing 9. Generarea certificatelor

Anexa 2. Funcția Lambda responsabilă de validarea cache-ului

```
import base64
import boto3
from botocore.exceptions import NoCredentialsError,
 ↪  PartialCredentialsError, ClientError
import redis
import httpx
import hashlib
import ssl
from pymongo import MongoClient
from pymongo.errors import ConnectionFailure

def get_instance_public_ip(instance_id, region='eu-central-1',
 ↪  public_ip=True):
    ec2 = boto3.client('ec2', region_name=region)
    try:
        response = ec2.describe_instances(InstanceIds=[instance_id])
        for reservation in response['Reservations']:
            for instance in reservation['Instances']:
                if instance['State']['Name'] == 'running':
                    return instance['PublicIpAddress'] if public_ip
                     ↪  else instance['PrivateIpAddress']
    except (NoCredentialsError, PartialCredentialsError):
        print("Credențialele nu sunt configurate corect.")
    except ClientError as e:
        if e.response['Error']['Code'] == 'UnauthorizedOperation':
            print(f"Eroare de autorizare:
             ↪  {e.response['Error']['Message']}")
        else:
            print(f"A aparut o eroare: {e}")
    except Exception as e:
        print(f"A aparut o eroare neasteptata: {e}")
    return None


REDIS_PORT = 6379
def connect_to_redis():
    try:
        REDIS_HOST =
         ↪  get_instance_public_ip(instance_id='i-08777038d1546da66')

        return redis.StrictRedis(host=REDIS_HOST,
                                 port=REDIS_PORT,
                                 decode_responses=True,
                                 ssl=True,
                                 ssl_cert_reqs=ssl.CERT_NONE
                                 )
    except Exception as e:
        print(str(e))
        return None

def connect_to_mongodb():
    try:
```

```
48          MONGO_HOST =
             ↪ get_instance_public_ip(instance_id='i-0bb9483c3a293294d')
49          client =
             ↪ MongoClient(f"mongodb://cosmin:cosmin@{MONGO_HOST}:27017")
50          db = client["mongodb"]
51          return db, client
52      except ConnectionFailure:
53          print("Failed to connect to the database. Please make sure
             ↪ MongoDB is running.")
54          return None, None
55  #
56  def lambda_handler(event, context):
57      cache = connect_to_redis()
58      if cache is None:
59          return {"message": "Failed to connect to Redis."}
60
61      db, client = connect_to_mongodb()
62      if db is None:
63          return {"message": "Failed to connect to MongoDB."}
64
65      all_keys = cache.keys('*')
66      if all_keys:
67          with httpx.Client() as http_client, client.start_session() as
             ↪ session:
68              try:
69                  for key in all_keys:
70                      parts = key.split("/", 1)
71                      domain, resource = parts if len(parts) > 1 else
                         ↪ (parts[0], "")
72                      ip_record = db.origin.find_one({'domain':
                         ↪ domain}, {'_id': 0, 'ip': 1},
                         ↪ session=session)
73                      if not ip_record:
74                          cache.delete(key)
75                          continue
76                      ip = ip_record['ip']
77                      etag = db.resources.find_one({'resource': key},
                         ↪ {'_id': 0, 'eTag': 1}, session=session)
78                      if etag:
79                          headers = {'If-None-Match': etag['eTag']}
80                          response =
                             ↪ http_client.head(f'http://{ip}/{resource}',
                             ↪ headers=headers)
81                          if response.status_code == 304:
82                              continue
83                          elif response.status_code == 404:
84                              cache.delete(key)
85                              db.resources.delete_one({'resource':
                                 ↪ key}, session=session)
86                              continue
87                          else:
88                              cache.delete(key)
89                              db.resources.delete_one({'resource':
                                 ↪ key}, session=session)
```

```
90                              continue
91                          else:
92                              response =
                                ↪ http_client.get(f'http://{ip}/{resource}')
93                              if response.status_code == 200:
94                                  content = response.content
95                                  hash_content =
                                    ↪ hashlib.sha256(content).hexdigest()
96                                  value = cache.get(key)
97
98                                  if not value:
99                                      cache.delete(key)
100                                     continue
101                                 decoded_value = base64.b64decode(value)
102                                 hash_cache =
                                    ↪ hashlib.sha256(decoded_value).hexdigest()
103
104                                 if hash_content != hash_cache:
105                                     cache.delete(key)
106                                     continue
107                             elif response.status_code == 404:
108                                 cache.delete(key)
109             except Exception as e:
110                 print(f"An error occurred: {e}")
111     client.close()
112     return {"message": "Cache verification complete."}
```

Listing 10. Funcția pentru validarea datelor din cache

Anexa 3. Componenta de monitorizare a sănătății

```python
import asyncio
import httpx
import time
from utile.mongo_connection import connect_to_mongodb
from utile.boto_aws import get_instance_public_ip

db, client = connect_to_mongodb()

def update_server_status(instance_id, new_status):
    db.edgeserver.update_one({"instance_id": instance_id}, {"$set": {"status": new_status}})

async def check_https_server_async(url, retries=3):
    for attempt in range(retries):
        try:
            async with httpx.AsyncClient(verify=False) as client:
                start_time = time.time()
                response = await client.head(url)
                elapsed_time = time.time() - start_time

                if response.status_code == 200:
                    print(f"Server responded with status code: {response.status_code}")
                    print(f"Response time: {elapsed_time:.2f} seconds")
                    return True, elapsed_time
                else:
                    print(f"Server returned error code: {response.status_code}")
                    return False, elapsed_time
        except httpx.RequestError as e:
            print(f"Request failed: {e}")
        await asyncio.sleep(1)
    return False, None

async def monitor_edge_servers(db):
    while True:
        cursor = db.edgeserver.find({"status": {"$ne": "OFF"}})
        servers = list(cursor)
        tasks = []
        for server in servers:
            ip_address = get_instance_public_ip(server['instance_id'], server['region'])
            if ip_address:
                url = f"https://{ip_address}/health"
                tasks.append(check_https_server_async(url))

        results = await asyncio.gather(*tasks)
        for server, (is_alive, response) in zip(servers, results):
            if not is_alive:
```

```
46                  print(f"Server {server['instance_id']} is down after
                     ↪  multiple attempts. Response time: {response}
                     ↪  seconds")
47                  update_server_status(server['instance_id'], 'CRASH')
48              else:
49                  print(f"Server {server['instance_id']} is up.
                     ↪  Response time: {response:.2f} seconds")
50                  if response > 1:
51                      update_server_status(server['instance_id'],
                         ↪  'SLOW')
52                  else:
53                      update_server_status(server['instance_id'], 'ON')
54          await asyncio.sleep(300)
55
56  async def main():
57      await monitor_edge_servers(db)
58
59  if __name__ == "__main__":
60      asyncio.run(main())
```

Listing 11. Script de monitorizare a sănătății

Anexa 4. Server DNS

```
class DynamicResolver(BaseResolver):
    def resolve(self, request, handler):
        client_ip = handler.client_address[0]
        print(f"Client IP: {client_ip}")

        best_server_ip, best_server_ipv6 =
         ↪ get_best_server_ip(client_ip)

        reply = request.reply()
        qname = request.q.qname
        qtype = request.q.qtype
        domain = str(qname).rstrip('.')
        if db.origin.find_one({'domain': domain}) is not None:
            print('database')
            if qtype == QTYPE.A and best_server_ip is not None:
                reply.add_answer(RR(qname, QTYPE.A,
                 ↪ rdata=A(best_server_ip), ttl=300))
            elif qtype == QTYPE.AAAA:
                if best_server_ipv6 is not None:
                    reply.add_answer(RR(qname, QTYPE.AAAA,
                     ↪ rdata=AAAA(best_server_ipv6), ttl=300))
                else:
                    print("Nu a fost gasita adresa IPv6 pentru cel
                     ↪ mai bun server.")
            else:
                reply.header.rcode = RCODE.NXDOMAIN
        else:
            print('Domeniu nu a fost gasit, interogare DNS
             ↪ implicita')
            dns_response = self.fallback_dns_query(request.pack())
            if dns_response:
                return DNSRecord.parse(dns_response)
        return reply

    def fallback_dns_query(self, data):
        cache_key = hashlib.sha256(data).hexdigest()
        if cache_key in dns_response_cache:
            return dns_response_cache[cache_key]

        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sock.settimeout(2)
        try:
            sock.sendto(data, ('127.0.0.1', 53))
            response, _ = sock.recvfrom(512)
            dns_response_cache[cache_key] = response
            return response
        except socket.timeout:
            print("Interogare DNS a expirat")
            return None
        finally:
            sock.close()
```

Listing 12. Clasa DynamicResolve

```
def get_best_server_ip(client_ip):
    client_lat, client_lon = get_location_from_ip(client_ip)
    instances_info = get_edgeservers_info()
    if client_lat is None or client_lon is None:
        # random in caz de nu obtinem pozitia clientului
        random_region = random.choice(list(instances_info.keys()))
        instance_id = instances_info[random_region]['instance_id']
        best_server_ip, best_server_ipv6 =
         ↪  get_ec2_instance_ip(random_region, instance_id)
        return best_server_ip, best_server_ipv6
    # vectorul de instante in ordinea distantei de client
    sorted_servers = sorted(instances_info.items(),
                            key=lambda item: haversine(client_lat,
                             ↪  client_lon, float(item[1]['lat']),
                             ↪  float(item[1]['lon'])))

    futures = [executor.submit(get_ec2_instance_ip, region,
     ↪  info['instance_id']) for region, info in sorted_servers]
    # returnam cel mai aporpiat server functional
    for future in as_completed(futures):
        best_server_ip, best_server_ipv6 = future.result()
        if best_server_ip:
            print(f"Server gasit: {best_server_ip}")
            return best_server_ip, best_server_ipv6

    return None, None
```

Listing 13. Funcția getbestserverip

Anexa 5. Filtru DDoS

```python
async def send_action_to_all_servers(ips, action):
    if action not in ["ban", "unban"]:
        raise ValueError("Invalid action. Must be 'ban' or 'unban'.")

    instance_id = get_instance()
    headers = {
        'Content-Type': 'application/json'
    }
    data = {
        "ip_addresses": ips
    }
    async with httpx.AsyncClient(verify=False) as client:
        for key, value in instance_id.items():
            ip = get_instance_public_ip(value['instance_id'], key)
            if ip:
                url = f'https://{ip}/{action}'
                try:
                    response = await client.post(url,
                     ↪  headers=headers, json=data)
                    if response.status_code == 200:
                        print(f"Successfully sent ban request to
                         ↪  {ip}")
                    else:
                        print(f"Failed to send ban request to {ip},
                         ↪  status code: {response.status_code}")
                except Exception as e:
                    print(f"Error sending ban request to {ip}: {e}")
```

Listing 14. Funcția send_action_to_all_servers

```python
def get_instance():
    edgeservers = db.edgeserver.find({'status': {'$in': ['ON',
     ↪  'SLOW']}})
    instances_info = {}
    for edgeserver in edgeservers:
        region = edgeserver['region']
        instances_info[region] = {
            'instance_id': edgeserver['instance_id']
        }
    return instances_info
```

Listing 15. Funcția get_instance

```python
async def analyze_requests():
    global MEAN_PER_DOMENIU, MEAN_PER_CLIENT

    domain_request_counts = defaultdict(int)
    client_request_counts = defaultdict(lambda: defaultdict(int))
    total_requests = 0

    for domain, clients in domain_requests.items():
        domain_total = sum(clients.values())
        domain_request_counts[domain] = domain_total
        for client, count in clients.items():
            client_request_counts[domain][client] = count
            total_requests += count

    if total_requests < 150:
        print("Not enough requests to update global means and standard deviations.")
        return
    print(f"total req {total_requests}")

    average_response_times = calculate_average_response_times()
    print(f"Average response times per instance: {average_response_times}")

    for domain, clients in domain_requests.items():
        domain_total = sum(clients.values())
        if (domain_total > (1/2)*MEAN_PER_DOMENIU ):
            print(f"Potential DDoS attack detected for domain {domain}")
            print(
                f"Total requests: {domain_total}, Global mean: {MEAN_PER_DOMENIU}")

            suspicious_ips = [ip for ip, count in clients.items() if count > MEAN_PER_CLIENT]
            print(f"Suspicious IPs: {suspicious_ips}")
            await send_ban_request(suspicious_ips)

            if len(clients) > 9:
                botnet_ips = [ip for ip, count in clients.items() if count > MEAN_PER_CLIENT / 10]
                if botnet_ips:
                    print(f"Potential Botnet DDoS attack detected for domain {domain}")
                    print(f"Botnet IPs: {botnet_ips}")
                    await send_ban_request(botnet_ips)
                return
        for client, count in clients.items():
            if count > MEAN_PER_CLIENT + (1/2) *MEAN_PER_CLIENT:
                print(f"Potential DDoS attack detected from client {client} on domain {domain}")
```

```
44                  print(f"Total requests: {count}, Global mean:
                     ↪  {MEAN_PER_CLIENT}")
45                  suspicious_ips = [client]
46                  await send_ban_request(suspicious_ips)
47                  return
48      mean_per_domain, mean_per_client =
         ↪  calculate_means(domain_request_counts, client_request_counts)
49      MEAN_PER_DOMENIU = (MEAN_PER_DOMENIU + mean_per_domain) / 2
50      MEAN_PER_CLIENT = (MEAN_PER_CLIENT + mean_per_client) / 2
51
52      print(f"Updated global mean requests per domain:
         ↪  {MEAN_PER_DOMENIU}")
53      print(f"Updated global mean requests per client:
         ↪  {MEAN_PER_CLIENT}")
54
55      historical_data['mean_per_domain'].append(mean_per_domain)
56      historical_data['mean_per_client'].append(mean_per_client)
57
58      domain_requests.clear()
59      response_time_cache.clear()
```

Listing 16. Funcția analyze_requests

Anexa 6. Platforma Web

6.1. API-ul Serviciului de Origine

- **GET /api/origin_service/plan/domain**:
  - **Descriere**: Returnează planul asociat unui domeniu specific.
  - **Coduri de răspuns**:
    * 200 - Success: Planul a fost returnat cu succes.
    * 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    * 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a accesa planul.
    * 404 - Resource Not Found: Domeniul nu a fost găsit.
    * 500 - Internal server error: Eroare internă a serverului.
- **GET /api/origin_service**:
  - **Descriere**: Returnează toate serverele de origine pentru un proprietar specificat.
  - **Parametri**: `owner` (optional) - Proprietarul serverului de origine.
  - **Coduri de răspuns**:
    * 200 - Success: Serverele de origine au fost returnate cu succes.
    * 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    * 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a accesa serverele de origine.
    * 404 - Resource Not Found: Nu au fost găsite servere de origine.
    * 500 - Internal server error: Eroare internă a serverului.
    * 503 - Database connection error: Eroare de conexiune la baza de date.
- **GET /api/origin_service/domain**:
  - **Descriere**: Returnează detaliile serverului de origine pentru un domeniu specific.
  - **Coduri de răspuns**:
    * 200 - Success: Detaliile serverului de origine au fost returnate cu succes.
    * 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    * 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a accesa serverul de origine.
    * 404 - Resource Not Found: Domeniul nu a fost găsit.
    * 500 - Internal server error: Eroare internă a serverului.
    * 503 - Database connection error: Eroare de conexiune la baza de date.
- **POST /api/origin_service**:
  - **Descriere**: Creează un nou server de origine.
  - **Coduri de răspuns**:
    * 201 - Success: Serverul de origine a fost creat cu succes.
    * 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    * 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a crea serverul de origine.

    - 409 - Issue or Conflicts: Domeniul există deja.
    - 422 - Unprocessable Entity: Datele trimise nu sunt valide.
    - 500 - Internal server error: Eroare internă a serverului.
    - 503 - Database connection error: Eroare de conexiune la baza de date.

- **PUT /api/origin_service/domain**:
  - **Descriere**: Actualizează detaliile unui server de origine.
  - **Coduri de răspuns**:
    - 200 - Success: Detaliile serverului de origine au fost actualizate cu succes.
    - 201 - Created: Un nou server de origine a fost creat, deoarece nu a fost găsit unul existent.
    - 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    - 404 - Resource Not Found: Serverul de origine nu a fost găsit.
    - 500 - Internal server error: Eroare internă a serverului.
    - 503 - Database connection error: Eroare de conexiune la baza de date.

- **PATCH /api/origin_service/mode_dev/domain**:
  - **Descriere**: Setează modul de dezvoltare pentru un server de origine.
  - **Coduri de răspuns**:
    - 200 - Success: Modul de dezvoltare a fost setat cu succes.
    - 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    - 404 - Resource Not Found: Domeniul nu a fost găsit.
    - 500 - Internal server error: Eroare internă a serverului.
    - 503 - Database connection error: Eroare de conexiune la baza de date.

- **PATCH /api/origin_service/mode_offline/domain**:
  - **Descriere**: Setează modul offline pentru un server de origine.
  - **Coduri de răspuns**:
    - 200 - Success: Modul offline a fost setat cu succes.
    - 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    - 404 - Resource Not Found: Domeniul nu a fost găsit.
    - 500 - Internal server error: Eroare internă a serverului.
    - 503 - Database connection error: Eroare de conexiune la baza de date.

- **PATCH /api/origin_service/resource_static/domain**:
  - **Descriere**: Setează resursele statice pentru un server de origine.
  - **Coduri de răspuns**:
    - 200 - Success: Resursele statice au fost setate cu succes.
    - 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    - 404 - Resource Not Found: Domeniul nu a fost găsit.
    - 500 - Internal server error: Eroare internă a serverului.
    - 503 - Database connection error: Eroare de conexiune la baza de date.

- **DELETE /api/origin_service/domain**:
  - **Descriere**: Șterge un server de origine.
  - **Coduri de răspuns**:
    * 200 - Successfully deleted domain: Domeniul a fost șters cu succes.
    * 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    * 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a șterge serverul de origine.
    * 404 - Domain not found: Domeniul nu a fost găsit.
    * 500 - Internal server error: Eroare internă a serverului.

6.2. API-ul Serviciului de Cache

- **GET /api/cache_service/cache_redis**:
  - **Descriere**: Returnează toate cheile din cache-ul Redis.
  - **Coduri de răspuns**:
    * 200 - Successfully retrieved cache keys: Cheile au fost returnate cu succes.
    * 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    * 500 - Internal server error: Eroare internă a serverului.
- **GET /api/cache_service/cache_redis/domain**:
  - **Descriere**: Returnează toate cheile din cache-ul Redis pentru un anumit domeniu.
  - **Coduri de răspuns**:
    * 200 - Successfully retrieved cache keys: Cheile au fost returnate cu succes.
    * 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    * 500 - Internal server error: Eroare internă a serverului.
- **DELETE /api/cache_service/cache_redis/domain**:
  - **Descriere**: Șterge toate cheile din cache-ul Redis pentru un anumit domeniu.
  - **Coduri de răspuns**:
    * 200 - Successfully deleted domain cache: Cheile au fost șterse cu succes.
    * 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    * 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a șterge cache-ul domeniului.
    * 500 - Internal server error: Eroare internă a serverului.
- **DELETE /api/cache_server/cache_redis/resource/resource**:
  - **Descriere**: Șterge o resursă specifică din cache-ul Redis.
  - **Coduri de răspuns**:
    * 200 - Successfully deleted resource: Resursa a fost ștersă cu succes.
    * 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    * 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a șterge resursa specificată.
    * 500 - Internal server error: Eroare internă a serverului.

6.3. API-ul Serviciului de EdgeServere

- **GET /api/edgeserver_service/banip**:
  - **Descriere**: Returnează toate adresele IP blocate.
  - **Coduri de răspuns**:
    - 200 - Successfully retrieved ban IPs: Adresele IP blocate au fost returnate cu succes.
    - 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    - 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a accesa resursele.
    - 500 - Internal server error: Eroare internă a serverului.
- **GET /api/edgeserver_service**:
  - **Descriere**: Returnează toate edge serverele.
  - **Coduri de răspuns**:
    - 200 - Success: Edge serverele au fost returnate cu succes.
    - 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    - 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a accesa resursele.
    - 500 - Internal server error: Eroare internă a serverului.
- **GET /api/edgeserver_service/instance_id**:
  - **Descriere**: Returnează detaliile unui edge server specific.
  - **Coduri de răspuns**:
    - 200 - Success: Detaliile edge serverului au fost returnate cu succes.
    - 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    - 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a accesa resursele.
    - 404 - Resource Not Found: Edge serverul nu a fost găsit.
    - 500 - Internal server error: Eroare internă a serverului.
- **POST /api/edgeserver_service**:
  - **Descriere**: Creează un nou edge server.
  - **Coduri de răspuns**:
    - 201 - Success: Edge serverul a fost creat cu succes.
    - 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    - 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a crea resursele.
    - 409 - Issue or Conflicts: Edge serverul există deja.
    - 422 - Unprocessable Entity: Datele trimise nu sunt valide.
    - 500 - Internal server error: Eroare internă a serverului.
    - 503 - Database connection error: Eroare de conexiune la baza de date.
- **PUT /api/edgeserver_service/instance_id**:

  - **Descriere**: Actualizează detaliile unui edge server specific.
  - **Coduri de răspuns**:
    * 200 - Success: Detaliile edge serverului au fost actualizate cu succes.
    * 201 - Created: Un nou edge server a fost creat, deoarece nu a fost găsit unul existent.
    * 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    * 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a accesa resursele.
    * 404 - Resource Not Found: Edge serverul nu a fost găsit.
    * 500 - Internal server error: Eroare internă a serverului.
    * 503 - Database connection error: Eroare de conexiune la baza de date.

- **DELETE /api/edgeserver_service/instance_id**:
  - **Descriere**: Șterge un edge server specific.
  - **Coduri de răspuns**:
    * 200 - Success: Edge serverul a fost șters cu succes.
    * 401 - Unauthorized access: Acces neautorizat, token invalid sau lipsă.
    * 403 - Forbidden access: Acces interzis, utilizatorul nu are permisiuni pentru a șterge resursele.
    * 404 - Resource Not Found: Edge serverul nu a fost găsit.
    * 500 - Internal server error: Eroare internă a serverului.
    * 503 - Database connection error: Eroare de conexiune la baza de date.

```
async def get_sqs_messages():
    all_messages = []
    while True:
        response = sqs_client.receive_message(
            QueueUrl=queue_url,
            MaxNumberOfMessages=10,
            WaitTimeSeconds=20,
        )
        messages = response.get('Messages', [])
        if not messages:
            break
        for msg in messages:
            try:
                body_parts = msg['Body'].split('|')
                if len(body_parts) != 6:
                    raise ValueError("Unexpected message format")
                client_ip, edge_server, domain, resource, state,
                 ↪  time_str = body_parts
                message_time =
                 ↪  datetime.strptime(time_str.strip(), '%Y-%m-%d
                 ↪  %H:%M:%S.%f')
                location = get_region(client_ip.strip())
                all_messages.append({
                    "message_id": msg['MessageId'],
                    "client_ip": location,
                    "edge_server": edge_server.strip(),
```

```
                    "domain": domain.strip(),
                    "resource": resource.strip(),
                    "state": state.strip(),
                    "time": message_time.isoformat()
                })
                sqs_client.delete_message(
                    QueueUrl=queue_url,
                    ReceiptHandle=msg['ReceiptHandle']
                )
            except Exception as e:
                print(f"Error processing message: {e}")
                continue
        return all_messages
```

Listing 17. Funcția get_sqs_messages