

## **REFERAT**

### **ANALIZA ARHITECTURII BAZATE PE MICROSERVICII**

**Student:** gr. TI-251M,  
**Perevoznic Vladislav**

**Coordonator: lect. univ., dr. Poștaru Andrei**

**Chișinău, 2025**

## CUPRINS

<b>ABREVIERI .....</b>	<b>3</b>
<b>INTRODUCERE.....</b>	<b>4</b>
<b>1 STUDIU COMPARATIV.....</b>	<b>5</b>
1.1 Definirea arhitecturilor software .....	6
1.2 Analiza comparativă generală .....	11
<b>2 CONCEPTUL ȘI DETALIEREA ARHITECTURII CU MICROSERVICII.....</b>	<b>13</b>
2.1 Problemele rezolvate de microservicii .....	14
2.2 Funcționarea arhitecturii cu microservicii.....	14
<b>3 DOMENIUL APLICATIV.....</b>	<b>16</b>
<b>CONCLUZII.....</b>	<b>17</b>
<b>BIBLIOGRAFIE .....</b>	<b>18</b>

## **ABREVIERI**

**MVP** – minimum viable product (produs minim viabil), versiunea cea mai simplă a unui sistem care conține doar funcționalitățile esențiale pentru a putea fi folosită de primii utilizatori și pentru a obține feedback real;

**API** – interfață de programare a aplicațiilor, un set de reguli și instrumente care permite diferitelor aplicații sau sisteme să comunice între ele, schimbând date și funcționalități într-un mod structurat;

**DevOps** – development + operations (dezvoltare + operațiuni), o metodologie și o cultură de lucru în care echipele de dezvoltare (cei care scriu codul) și echipele de operațiuni (cei care pun sistemul în producție și îl mențin stabil) lucrează împreună, automatizează procesele și livră actualizări rapid și sigur, rezultatul, mai puține erori, deploy-uri mai dese și sisteme mai stabile.

**CI/CD** – continuous integration / continuous delivery (integrare continuă / livrare continuă), un set de practici automate;

**CI** – integrare continua, de fiecare dată când un programator adaugă cod nou, acesta este testat automat și integrat în proiectul principal;

**CD** – livrare sau deployment continuu, codul testat și validat este pus automat în producție (pe serverele reale), uneori de mai multe ori pe zi, rezultatul, actualizări rapide, fără intervenție manuală și cu risc minim de erori;

**DDD** – domain driven design, (proiectare bazată pe domeniul), o abordare de dezvoltare software în care accentul principal se pune pe înțelegerea profundă a domeniului de business (în cazul dat, asistența socială, tichete de suport, fluxuri atas/stas etc.).

## INTRODUCERE

Arhitectura cu microservicii este un mod modern de a construi aplicații software, apărut ca răspuns la creșterea complexității sistemelor din ultimii ani. Pe măsură ce lumea devine tot mai digitalizată, iar companiile dezvoltă aplicații tot mai mari, arhitectura monolică tradițională a început să ridice multe probleme. Un sistem monolic este greu de modificat, greu de testat și dificil de extins, mai ales atunci când cerințele se schimbă rapid. În acest context, microserviciile au apărut ca o soluție care să facă aplicațiile mai flexibile și mai ușor de gestionat.

Microserviciile presupun împărțirea unei aplicații în componente mici, independente, fiecare cu rolul și funcția ei clară. Fiecare microserviciu se poate dezvolta, testa și implementa separat, ceea ce reduce riscurile și accelerează ritmul de lucru al echipei. Această abordare este deosebit de utilă în organizațiile mari, unde mai multe echipe trebuie să lucreze în paralel la aceeași aplicație. Apariția lor este strâns legată de evoluția tehnologiilor cloud, de containerizare și de nevoia de scalare rapidă, elemente care permit fiecarui serviciu să fie pornit, opus sau extins independent de celelalte.

La nivel global, trecerea la microservicii este motivată de ritmul alert al schimbărilor din domeniul IT, de concurența dintre companii și de presiunea de a lansa funcționalități noi cât mai repede. Platformele digitale moderne, comerțul online, aplicațiile de streaming sau rețelele sociale sunt doar câteva exemple de sisteme care trebuie să suporte milioane de utilizatori și un volum mare de date, cerințe pentru care microserviciile sunt o alegere potrivită.

Totuși, această arhitectură aduce și probleme noi. Gestionarea unui număr mare de servicii mici poate fi dificilă, iar comunicarea dintre ele trebuie monitorizată atent. Apărări provocări legate de securitate, sincronizarea datelor, testarea distribuției, precum și necesitatea unor instrumente specializate de monitorizare și logare. În etapa actuală, multe organizații se confruntă cu complexitatea operațională crescută și cu nevoia de personal tehnic bine pregătit pentru a administra astfel de sisteme.

Totuși, arhitectura cu microservicii s-a dezvoltat ca răspuns la limitele sistemelor monolitice și la cerințele tot mai mari ale aplicațiilor moderne. Deși vine cu propriile provocări, ea oferă flexibilitate, viteză și posibilitatea de a adapta rapid aplicațiile la schimbările din lumea reală, motive pentru care este astăzi una dintre arhitecturile preferate în dezvoltarea software.

## 1 STUDIU COMPARATIV

De-a lungul timpului, aplicațiile software au devenit tot mai complexe, iar modul în care sunt construite a evoluat în același ritm. Două dintre cele mai cunoscute abordări în dezvoltarea unui sistem sunt arhitectura monolică și arhitectura bazată pe microservicii. Aceste două modele sunt diferite atât ca structură, cât și ca mod de operare, iar alegerea uneia dintre ele depinde de mai mulți factori: dimensiunea aplicației, resursele disponibile, viteza de dezvoltare dorită, tipul echipei și cerințele pe termen lung. De aceea, un studiu comparativ devine important pentru a înțelege clar avantajele și limitele fiecărei arhitecturi.

Arhitectura monolică a fost mult timp varianta tradițională și preferată în dezvoltarea software. Ea presupune construirea întregii aplicații ca un singur bloc, în care toate funcțiile – interfață, logică de business, procesare, acces la date – sunt integrate într-o structură unitară. Monolitul este ușor de înțeles la început și simplu de lansat în producție, dar pe măsură ce aplicația crește, devine tot mai greu de actualizat, testat și întreținut. Orice schimbare, chiar și minoră, poate necesita recompilarea întregului sistem, ceea ce crește riscul de erori și încetinește ritmul de dezvoltare.

În contrast, arhitectura cu microservicii împarte aplicația în componente mici, independente, fiecare responsabilă pentru o anumită funcționalitate. Această abordare a apărut ca răspuns la problemele întâlnite în monolit și este folosită tot mai des în companii mari, care au nevoie de flexibilitate, scalare rapidă și dezvoltare paralelă. Fiecare microserviciu poate fi implementat, actualizat și scalat individual, ceea ce reduce dependențele și oferă echipelor libertate mai mare în procesul de dezvoltare. Totuși, odată cu această libertate, cresc și provocările: comunicarea dintre servicii devine mai complexă, monitorizarea este mai dificilă, iar infrastructura necesară este mai avansată.

În prezent, tranziția către microservicii este influențată de mai multe schimbări la nivel global: creșterea volumului de date, numărul mare de utilizatori simultani, necesitatea lansării rapide de funcționalități noi și apariția tehnologiilor moderne precum cloud computing, containerele și orchestrarea cu Kubernetes. Toate acestea au făcut ca microserviciile să devină o soluție atractivă pentru sistemele mari și dinamice. Cu toate acestea, arhitectura monolică nu a dispărut. Ea rămâne o opțiune viabilă pentru proiectele mici și medii, unde complexitatea suplimentară a microserviciilor nu este justificată.

Prin urmare, scopul acestui studiu comparativ este de a analiza în mod echilibrat cele două arhitecturi, evidențiind atât punctele lor forte, cât și dificultățile întâlnite în practică. Înțelegerea acestor diferențe ajută la alegerea unei arhitecturi potrivite pentru nevoile reale ale unui proiect, evitând costurile inutile sau problemele tehnice pe termen lung. Astfel, putem observa o imagine clară asupra modului în care cele două modele se raportează la scalabilitate, performanță, menenanță, viteza de dezvoltare și complexitatea operațională.

## 1.1 Definirea arhitecturilor software

În dezvoltarea sistemelor informatici moderne, alegerea unei arhitecturi software reprezintă una dintre cele mai importante decizii strategice, deoarece influențează direct modul în care aplicația va fi construită, întreținută și extinsă în timp. Arhitectura software stabilește structura logică a sistemului, relațiile dintre componente și principiile care guvernează interacțiunea acestora. Alegerea între o arhitectură monolică și una bazată pe microservicii determină nu doar abordarea tehnică, ci și modul de organizare a echipei, gestionarea resurselor și complexitatea procesului de implementare.

**Arhitectura monolică** este considerată forma tradițională de construcție a aplicațiilor software. Într-un sistem monolic, toate componentele aplicației, interfața utilizator, logica de business și accesul la bazele de date, sunt integrate într-o singură unitate compactă, situate într-o singură soluție și funcționează împreună ca un întreg. Toate funcționalitățile rulează în același spațiu de execuție, iar comunicarea dintre module se realizează intern, fără interfețe externe. Comunicarea dintre componente se face prin apeluri directe de funcții. Acest stil de dezvoltare a aplicațiilor simplifică foarte mult lucrurile, în special în primele faze de dezvoltare. Principiul de bază al arhitecturii monolitice constă în unitatea structurală și logică, aplicația este construită, testată și implementată ca un întreg. Acest model este ideal pentru proiecte mici sau medii, în care complexitatea sistemului este limitată, iar modificările sunt rare [1].

În figura 1.1, este reprezentată arhitectura monolică, cu rolul fiecărei componente și modul în care ele comunică. Ca exemplu, este modul în care funcționează o aplicație de tip E-Commerce construită în arhitectura monolică. În acest model, toate funcționalitățile aplicației sunt incluse într-un singur bloc software, care se execută ca o unitate unică, aceasta înseamnă că orice modificare sau actualizare a aplicației necesită modificarea și redistribuirea (redeploying) întregului monolit.

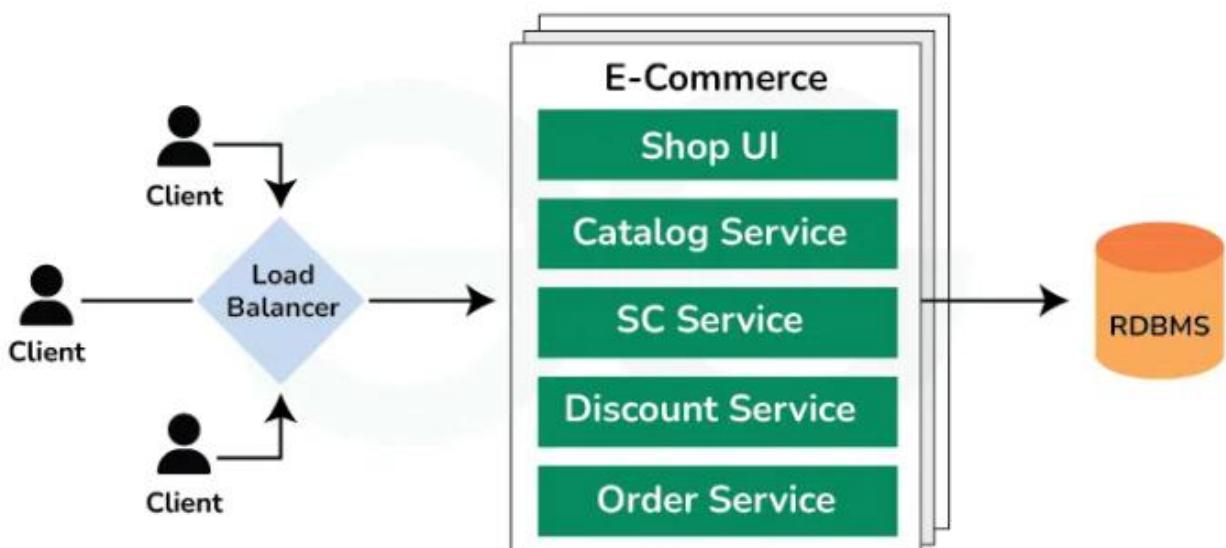


Figura 1.1 – Arhitectura monolică

**Clientul** reprezintă utilizatorul final care accesează aplicația. Poate fi un browser web sau o aplicație mobilă. Trimitе cereri (HTTP requests) către sistem („Deschide pagina produselor”, „Adaugă în cos”, „Finalizează comanda” etc.). Toți clienții comunică cu același punct de intrare în aplicație.

**Load Balancer** este o componentă care distribuie traficul către serverele aplicației monolitice. Scopul lui este să împartă cererile între mai multe instanțe ale aceluiași monolit, dacă există. Este util doar pentru scalare **orizontală**, adică atunci când există mai multe copii ale întregii aplicații.

Aplicația E-Commerce (Monolitul), tot codul aplicației este împachetat într-o singură aplicație care rulează ca un tot. În interiorul monolitului se află toate modulele, precum:

a) Shop UI:

- 1) partea de interfață cu utilizatorul;
- 2) gestiunea paginilor, afisarea produselor, formularelor etc;
- 3) se află în același cod de bază cu celelalte servicii;

b) Catalog Service:

- 1) gestionează lista de produse (denumiri, prețuri, stocuri);
- 2) returnează informațiile necesare pentru afișarea catalogului;

c) SC Service (Shopping Cart Service):

- 1) gestionează coșul de cumpărături (adăugare produse, ștergere, actualizare cantități);

d) Discount Service:

- 1) aplică reduceri, cupoane, coduri promoționale;

e) Order Service:

- 1) se ocupă de plasarea comenziilor, generarea facturilor, statusul livrării.

Cum funcționează comunicarea între componente în monolit? Într-o arhitectură monolică, toate aceste module rulează în același *proces*. Comunicarea dintre ele este directă, prin apeluri de metode interne (funcții, clase). Nu există rețea între servicii, nu există API-uri separate. Totul este un singur program.

Exemplu de flux:

- a) clientul trimitе o cerere către Load Balancer;
- b) Load Balancer trimitе cererea către aplicația monolică;
- c) aplicația parcurge intern modulele:
  - 1) Shop UI → Catalog Service (pentru produse);
  - 2) SC Service → Discount Service (pentru calcule);
  - 3) Order Service → baza de date;
- d) rezultatul se întoarce la client.

**RDBMS** (baza de date relațională), monolitul comunică cu o singură bază de date centralizată. Folosește tabele relaționale, toate modulele (Catalog, Coș, Order etc.) accesează aceeași bază. Există un singur punct de eșec, dacă RDBMS cade, aplicația nu mai funcționează.

Avantajele arhitecturii monolitice includ [2]:

- simplitatea în dezvoltare inițială, o singură bază de cod, ușor de gestionat de o echipă mică (2–5 dezvoltatori);
- lansarea rapidă a unei versiuni minime funcționale (MVP);
- costuri inițiale reduse și menenanță facilă în fazele incipiente;
- testare unitară și implementare directă fără integrare multiplă.

Totuși, pe măsură ce aplicația crește, arhitectura monolică devine greu de întreținut, odată cu extinderea, ca exemplu de la 1500 de utilizatori, dezavantajele ies la iveală, ca scalabilitate limitată, deoarece întregul sistem trebuie scalat uniform, chiar dacă doar o parte are nevoie. Menenanță dificilă, unde o modificare minoră poate cere redeploy total și riscuri de downtime și un potențial "spaghetti code" care complică colaborarea în echipe mari, ducând uneori la decizii radicale de refactorizare completă.

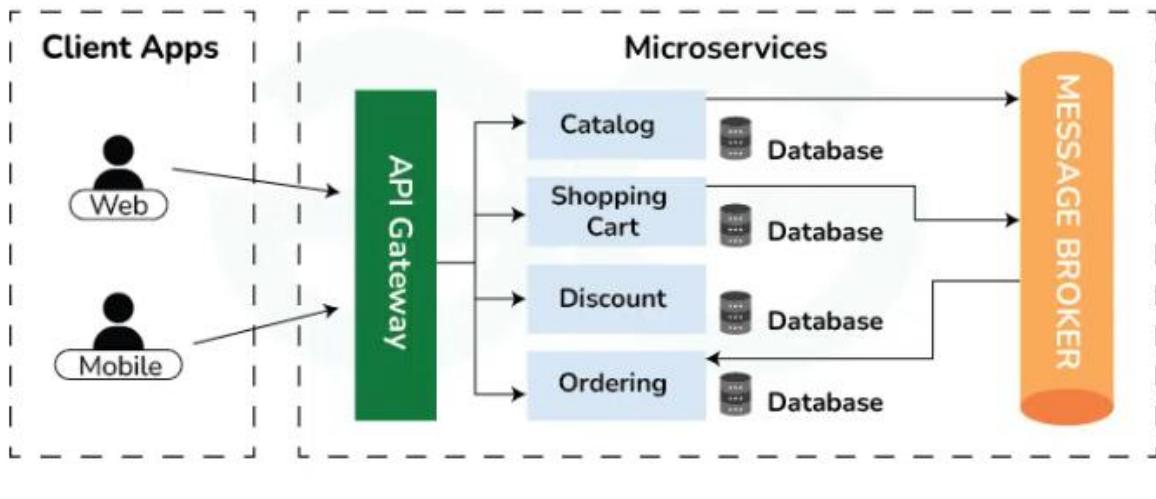
Printre dezavantaje, se remarcă:

- scalabilitate limitată, aplicația poate fi extinsă doar în ansamblu, nu pe module individuale;
- dificultate în identificarea și izolarea erorilor;
- orice modificare impune recompilarea și redeploy a întregului sistem;
- risc ridicat de instabilitate, o eroare într-o componentă poate afecta întregul sistem;
- dificultate în munca paralelă a mai multor echipe pe același cod sursă.

**Arhitectura bazată pe microservicii** reprezintă o abordare modernă și flexibilă de dezvoltare a aplicațiilor complexe. În acest model, sistemul este împărțit în componente independente, numite **microservicii**, fiecare responsabilă pentru o funcționalitate clar definită (de exemplu autentificare, gestionarea sesizărilor, raportare, administrarea utilizatorilor etc.). Fiecare microserviciu rulează autonom, are propria bază de date și comunică cu celelalte componente prin **interfețe API** sau mesaje standardizate [1].

Principiul fundamental al acestei arhitecturi este **independența componentelor**, fiecare microserviciu poate fi dezvoltat, testat, scalat și implementat separat, fără a afecta funcționarea celorlalte. Această abordare oferă o agilitate ridicată în dezvoltare și întreținere, fiind adoptată de companii mari precum Google, Netflix, Amazon și Uber.

Figura 1.2, remarcă cum funcționează o aplicație de tip **E-Commerce** construită folosind **microservicii**, o arhitectură în care funcționalitățile sunt separate în servicii independente care comunică între ele prin API-uri și mesaje.



## Microservices Architecture



**Figura 1.2 – Arhitectura bazată pe microservicii**

În această imagine avem trei zone importante, **Client Apps** (Aplicații client, Web și Mobile), **API Gateway** și **Microserviciile** (Catalog, Shopping Cart, Discount, Ordering) [3].

Client Apps sau utilizatorii accesesează aplicația fie de pe un browser web, fie dintr-o aplicație mobilă. Indiferent de dispozitivul folosit, toți intră în sistem prin același punct, API Gateway. Aceasta joacă rolul unei recepții centrale, unde fiecare cerere este preluată și trimisă exact acolo unde trebuie. Astfel, clienții nu „văd” în interiorul sistemului și nu comunică direct cu serviciile, totul trece prin acest singur filtru care ordonează traficul și asigură securitatea.

Odată ce API Gateway a primit cererea, el o direcționează către microserviciul potrivit. Spre deosebire de arhitectura monolică, unde toate funcționalitățile se află într-un singur bloc, aici fiecare parte a aplicației este construită separat și funcționează independent. Există un microserviciu pentru *catalog*, unul pentru coșul de *cumpărături*, unul pentru *reducere* și altul care gestionează comenzi. Fiecare dintre acestea este ca o „*mini-aplicație*” cu propriile reguli și cu propria bază de date. Această separare permite fiecarui serviciu să fie actualizat, reparat sau scalat fără a afecta restul sistemului.

De exemplu, atunci când un utilizator cere lista produselor, API Gateway trimite solicitarea către microserviciul Catalog. Acesta cauță informațiile în baza lui de date și le trimită înapoi. Dacă utilizatorul adaugă un produs în coș, cererea este direcționată către serviciul Shopping Cart, care își actualizează propria bază de date independent de celelalte servicii. La fel se întâmplă și cu microserviciul Discount, care calculează orice reducere, și cu cel pentru Ordering, care finalizează o comandă și îi salvează detaliile.

Pe lângă comunicarea directă prin API Gateway, microserviciile pot comunica și în mod indirect, printr-un Message Broker. Acesta funcționează ca un sistem de mesagerie intern, unde serviciile pot publica evenimente pe care altele le pot prelua. De exemplu, atunci când o comandă este finalizată, microserviciul Ordering poate trimite un mesaj de tip „Order Placed”, iar alte servicii, cum ar fi procesarea plășilor sau

notificările, pot reacționa automat la acesta. Această comunicare asincronă face sistemul mai flexibil și reduce încărcarea directă dintre servicii.

Întregul ecosistem funcționează ca un ansamblu de componente autonome, fiecare având clar stabilite responsabilitățile. Fiecare microserviciu are propria sa bază de date, ceea ce înseamnă că nu există interferențe la nivel de date și nici dependențe rigide între module. Dacă un serviciu necesită actualizări sau trebuie scalat pentru a suporta mai mulți utilizatori, acesta poate fi modificat fără a afecta funcționarea celorlalte părți ale sistemului.

În final, această arhitectură oferă un sistem flexibil, ușor de extins și rezistent la erori. Fiecare componentă poate evoluă independent, iar comunicarea dintre ele, fie prin API Gateway, fie prin mesaje interne, asigură o colaborare fluidă și eficientă. Astfel, microserviciile rezolvă multe dintre limitările arhitecturilor clasice și permit construirea unor aplicații complexe, dar bine organizate și ușor de întreținut.

Avantajele arhitecturii cu microservicii includ:

- scalabilitate excelentă, fiecare componentă poate fi extinsă separat în funcție de nevoile de performanță;
- mențenanță facilă, erorile pot fi izolate și corectate fără impact asupra întregului sistem;
- flexibilitate tehnologică, diferite microservicii pot fi dezvoltate în limbaje și medii diferite;
- organizare eficientă a echipelor, fiecare echipă gestionează un modul clar definit;
- continuitate operațională, actualizările pot fi făcute fără a opri întregul sistem.

Pe altă parte, dezavantajele acestui model sunt asociate cu complexitatea sa crescută:

- necesitatea unei infrastructuri mai avansate (orchestrare, DevOps, CI/CD);
- dificultăți în sincronizarea versiunilor între microservicii;
- costuri mai mari de implementare inițială;
- nevoie unei comunicări excelente între echipele tehnice.

Pentru sistemul de monitorizare și suport destinat asistenților sociali, o arhitectură bazată pe microservicii ar permite o gestionare mai eficientă a volumului mare de sesizări, integrarea cu alte sisteme (eSocial, bazele de date teritoriale) și adaptarea ușoară la noi cerințe funcționale. De exemplu, un microserviciu ar putea fi dedicat recepționării solicitărilor, altul ar gestiona comunicarea cu operatorii, iar un al treilea ar analiza indicatorii de performanță în timp real.

Implementarea unei arhitecturi pe microservicii nu este doar o alegere tehnică, ci și o strategie organizațională care influențează modul în care lucrează echipele și cum evoluează un produs software. De-a lungul timpului, s-au conturat câteva practici esențiale pentru ca microserviciile să funcționeze eficient într-un mediu real.

Una dintre cele mai importante abordări este **Domain-Driven Design (DDD)**. Prin împărțirea sistemului în domenii bine definite, fiecare cu propriile limite de responsabilitate, echipele pot crea microservicii independente, clare și ușor de întreținut. Această delimitare reduce interdependențele inutile

și permite echipelor să lucreze simultan, fiecare pe partea sa de business, fără să blocheze dezvoltarea celorlalți.

O altă componentă critică o reprezintă **service discovery și managementul API-urilor**. Pentru ca microserviciile să comunice între ele într-un mod dinamic și sigur, sistemele moderne folosesc gateway-uri API și mecanisme automate de descoperire a serviciilor. Aceste instrumente gestionează accesul, versiunile API-urilor și securitatea canalelor de comunicare, oferind un control centralizat asupra întregii interacțiuni dintre servicii.

Într-o arhitectură distribuită, apar inevitabil probleme locale. De aceea, este esențială aplicarea unor **pattern-uri de reziliență**. Tehnici precum „circuit breaker”, „retry”, „fallback” sau „bulkhead” ajută la izolarea erorilor și previn propagarea lor în restul sistemului. Chiar dacă un microserviciu întâmpină un incident, aplicația poate continua să funcționeze, menținând o experiență stabilă pentru utilizatori.

Pentru a gestiona corect acest ecosistem complex, este necesară o observabilitate avansată. Monitorizarea nu se limitează doar la loguri și grafice de performanță. Microserviciile necesită vizibilitate completă, obținută prin instrumente precum „distributed tracing”, centralizarea logurilor, dashboard-uri de metriți și sisteme automate de alertare. Aceste mecanisme permit echipelor să identifice rapid problemele și să mențină stabilitatea operațională.

În final, toate aceste strategii devin cu adevărat eficiente doar atunci când sunt susținute de **procese solide de automatizare și DevOps**. Fără pipeline-uri automate pentru testare, integrare și deploy, complexitatea microserviciilor poate ajunge să depășească beneficiile lor. Automatizarea este cea care menține un ritm constant de livrare și reduce risurile umane într-un sistem cu multe componente distribuite [2]. Un pipeline obișnuit poate avea:

- **build**, codul e compilat / împachetat;
- **test**, se execută automat teste;
- **code quality check**, se analizează stilul, vulnerabilitățile (linting, SAST);
- **deploy pe staging**, se trimite automat aplicația pe un server de test;
- **deploy pe producție**, dacă totul e OK, aplicația se publică pentru utilizatori.

## 1.2 Analiza comparativă generală

Alegerea arhitecturii software potrivite reprezintă una dintre deciziile esențiale în planificarea unui sistem informatic. Întrucât obiectivele acestui proiect presupun fiabilitate, scalabilitate și un flux continuu de asistență tehnică pentru un număr mare de utilizatori, este necesară o analiză comparativă detaliată între cele două paradigmă arhitecturale „*monolică* și *bazată pe microservicii*” prin prisma **criteriilor tehnice**.

**Scalabilitate**, arhitectura monolică oferă o scalabilitate limitată, întrucât aplicația trebuie extinsă în ansamblu. Orice creștere a volumului de date sau a numărului de utilizatori implică resurse suplimentare la nivelul întregii aplicații. În schimb, arhitectura cu microservicii permite scalarea individuală a

componentelor, de exemplu, modulul de raportare sau cel de gestionare a sesizărilor poate fi extins fără a afecta restul sistemului. De exemplu, Amazon a făcut această trecere (de la monolit, la microservicii), deoarece baza sa de cod monolică, aflată în rapidă expansiune, nu putea fi scalată sau actualizată eficient. Structura monolică le-a împiedicat capacitatea de a inova și de a ține pasul cu cerințele în creștere [4].

**Mantenabilitate**, într-un sistem monolic, menenanța devine tot mai dificilă odată cu creșterea volumului de cod. Modificarea unei funcționalități poate avea efecte nedorite asupra altor părți ale aplicației. În microservicii, fiecare modul este izolat, ceea ce permite intervenții și actualizări locale, fără riscul de a compromite sistemul global.

**Securitate**, arhitectura monolică oferă o securitate mai simplă de gestionat datorită controlului centralizat, însă orice vulnerabilitate afectează întregul sistem. În microservicii, securitatea este granulară, fiecare serviciu poate avea propriile politici de autentificare și criptare, reducând riscul de propagare a breșelor, dar necesitând o coordonare mai complexă.

**Performanță**, în monolit, comunicarea internă este mai rapidă, deoarece toate funcțiile se află în același spațiu de memorie. Totuși, odată cu creșterea dimensiunii aplicației, performanța scade. În microservicii, comunicarea prin API-uri poate adăuga un mic cost de latență, însă avantajul major este stabilitatea și performanța predictibilă pe termen lung, datorită scalării selective [4].

Analiza comparativă arată că, deși arhitectura monolică are avantaje în fazele inițiale, chiar costuri reduse, echipă mică, implementare rapidă, aceasta devine dificil de întreținut și extins odată cu creșterea complexității sistemului. În schimb, arhitectura bazată pe microservicii oferă o scalabilitate superioară, reziliență operațională și adaptabilitate la schimbări frecvente, elemente esențiale pentru un sistem național de suport tehnic.

Din perspectiva unui Project Manager, microserviciile presupun un efort mai mare de coordonare și planificare, dar oferă beneficii clare pe termen lung, reducerea riscurilor de blocaj, menenanță distribuită, actualizări independente și posibilitatea de integrare cu alte platforme. Prin urmare, pentru un sistem aflat la început, un prototip monolic poate fi o alegere pragmatică. Totuși, pentru implementarea finală și extinderea la scară națională, arhitectura bazată pe microservicii reprezintă soluția recomandată, oferind echilibrul optim între performanță, flexibilitate și sustenabilitate.

## **2 CONCEPTUL ȘI DETALIEREA ARHITECTURII CU MICROSERVICII**

Arhitectura cu microservicii a apărut ca o evoluție firească în lumea dezvoltării software, în momentul în care aplicațiile au devenit tot mai mari, iar modul tradițional de construire a lor, *arhitectura monolică*, începea să creeze multe dificultăți. Pe măsură ce companiile creșteau și nevoile utilizatorilor se diversificau, sistemele monolitice devineau greu de modificat și și mai greu de scalat. Practic, orice schimbare mică într-o aplicație mare putea afecta întregul sistem. De aici a apărut nevoia unei arhitecturi mai flexibile, care să permită dezvoltarea mai ușoară, actualizări rapide și separarea responsabilităților.

Microserviciile vin tocmai să rezolve această problemă. În loc să existe o singură aplicație uriașă care face tot, aplicația este împărțită în bucăți mai mici, numite microservicii. Fiecare microserviciu este responsabil pentru o singură funcționalitate bine definită, de exemplu, gestionarea catalogului de produse, a coșului de cumpărături, a reducerilor sau a comenziilor. Aceste servicii mici funcționează independent, pot fi dezvoltate separat și chiar au propriile baze de date, ceea ce le oferă autonomie totală.

Arhitectura cu microservicii este foarte utilă atunci când aplicația este complexă, are mulți utilizatori sau trebuie să se adapteze rapid la schimbări. De exemplu, într-o perioadă de trafic intens, cum ar fi Black Friday, doar microserviciul responsabil de catalog sau de coș poate fi scalat, fără a aloca resurse suplimentare pentru toată aplicația. Această flexibilitate nu era posibilă în arhitectura monolică.

Modul în care funcționează microserviciile este destul de logic. Utilizatorul trimite o cerere printr-o aplicație web sau mobilă, iar aceasta ajunge la un API Gateway. Acesta este un punct unic de intrare care decide unde trebuie trimisă cererea. Fiecare microserviciu procesează partea lui de logică și, dacă este nevoie, comunică cu alte servicii fie direct prin API-uri, fie prin intermediul unui sistem de mesaje precum un message broker. Comunicarea prin mesaje face posibilă procesarea în fundal și permite sistemului să funcționeze fluid chiar și atunci când un serviciu este mai încărcat decât altul.

De ce a apărut arhitectura cu microservicii? Arhitectura cu microservicii nu a apărut întâmplător, ci ca o consecință directă a problemelor tot mai evidente ale aplicațiilor monolitice. Odată cu digitalizarea accelerată, companiile au ajuns să construiască sisteme uriașe, folosite de milioane de utilizatori. Într-o astfel de realitate, modelul monolic devinea dificil de gestionat, deoarece aplicațiile au devenit prea mari și greu de gestionat, în monolit modificările mici afectau întregul sistem, scalarea era dificilă și costisitoare, echipele mari aveau probleme de coordonare pe un cod unic, nevoia de flexibilitate și actualizări rapide.

Astfel, era nevoie de o arhitectură flexibilă, modulară, ușor de extins. Microserviciile au apărut ca o soluție naturală pentru aceste provocări, susținute și de evoluția tehnologiilor precum cloud-ul, containerele (Docker), orchestrarea (Kubernetes) și DevOps. În final, microserviciile lucrează împreună ca un grup de componente independente, fiecare cu rolul său bine stabilit. Ele permit sistemelor moderne să fie flexibile, scalabile și ușor de extins, reprezentând o soluție eficientă pentru problemele pe care aplicațiile mari le întâmpină în arhitectura monolică.

## 2.1 Problemele rezolvate de microservicii

Principiul modularității și independenței componentelor. Arhitectura bazată pe microservicii reprezintă un model modern de proiectare a aplicațiilor, în care sistemul este împărțit în componente mici, independente și specializate, fiecare responsabilă pentru o funcționalitate bine delimitată. Principiul fundamental al acestei arhitecturi este modularitatea, care permite ca fiecare microserviciu să fie dezvoltat, testat, scalat și implementat separat, fără a afecta funcționarea altor părți ale sistemului. Fiecare serviciu are propriul său ciclu de viață, o logică de business proprie și, în multe cazuri, chiar propria bază de date. Această independență structurală asigură o flexibilitate ridicată, o scalabilitate granulară și o mențenabilitate superioară în comparație cu o arhitectură monolică [5].

Arhitectura cu microservicii rezolvă problema complexității și rigidității (**elimină înțepenirea și lipsa de flexibilitate**) aplicațiilor mari. Prin împărțirea sistemului în componente independente.

**Dezvoltare mai rapidă**, fiecare echipă se poate ocupa de un microserviciu diferit, fără a afecta restul aplicației.

**Scalare eficientă**, doar microserviciile suprasolicitata pot fi scalate (de exemplu, doar serviciul de catalog dacă utilizatorii caută intens produse).

**Izolarea erorilor**, dacă un microserviciu eşuează, restul aplicației poate continua să funcționeze.

**Implementare independentă**, un microserviciu poate fi actualizat fără a opri întreaga aplicație.

**Libertate tehnologică**, fiecare microserviciu poate fi scris în limbajul sau tehnologia cea mai potrivită pentru rolul său. Rigiditatea înseamnă că aplicația este:

- greu de modificat;
- greu de extins;
- greu de actualizat;
- greu de scalat;
- greu de întreținut.

Pentru ce a fost dezvoltată această arhitectură:

- aplicații complexe cu multe funcționalități;
- sisteme cu mulți utilizatori simultani;
- contexte unde este nevoie de viteză în dezvoltare;
- echipe mari sau distribuite geografic;
- aplicații care necesită disponibilitate ridicată.

## 2.2 Funcționarea arhitecturii cu microservicii

Arhitectura cu microservicii poate fi înțeleasă cel mai bine după explicarea la o aplicație mare, precum un magazin online. În loc ca aplicația să fie construită ca un singur bloc mare și greu de schimbă, ea este împărțită în componente mici, numite microservicii. Fiecare microserviciu este responsabil pentru o

funcționalitate specifică, unul gestionează produsele, altul administrarea coșului de cumpărături, altul reducerile, iar altul comenzi. Aceste servicii funcționează independent, își pot face treaba fără să depindă direct de restul sistemului și pot fi modificate sau actualizate separat, atunci când este nevoie.

Când un utilizator face o acțiune, de exemplu, cauță un produs sau adaugă ceva în coș, cererea lui ajunge mai întâi la un punct central numit API Gateway. Acesta funcționează ca o recepție, primește cererea și o trimită exact spre microserviciul care trebuie să o rezolve. Dacă utilizatorul vrea să vadă produsele, cererea merge către microserviciul Catalog. Dacă apasă pe „Adaugă în coș”, cererea merge către microserviciul specializat în coșuri. Totul se întâmplă rapid, fără ca utilizatorul să vadă ce se petrece în interior.

Fiecare microserviciu are propria logică și chiar propria bază de date, ceea ce înseamnă că poate funcționa fără să interfereze cu celelalte. Dacă, de exemplu, microserviciul Catalog se actualizează sau este oprit pentru întreținere, celelalte microservicii, precum cel pentru reduceri sau cel pentru coș, pot continua să funcționeze normal. Acest lucru face aplicația mult mai flexibilă și mai rezistentă la erori.

În plus, microserviciile comunică între ele în două moduri. Uneori comunică direct, ca într-o conversație telefonică, folosind API-uri. Alteori comunică prin mesaje trimise printr-un sistem de tip „broker”, asemănător cu o poștă internă. De exemplu, atunci când o comandă este finalizată, microserviciul de comenzi trimite un mesaj general: „Comandă creată”. Alte microservicii, cum ar fi cel pentru plăți sau cel pentru notificări, pot prelua acest mesaj și își pot face treaba automat, fără ca cineva să le spună în mod direct ce să facă.

Întregul proces se desfășoară rapid și eficient, deoarece fiecare microserviciu este mic, organizat și bine specializat. La final, după ce toate serviciile și-au făcut partea lor, răspunsul ajunge înapoi la utilizator prin API Gateway, iar acesta vede rezultatul imediat pe ecran.

Astfel, arhitectura cu microservicii transformă aplicațiile mari într-un ansamblu de componente mici, independente și ușor de întreținut. Această organizare face aplicațiile moderne mult mai flexibile, scalabile și capabile să se adapteze rapid la nevoile utilizatorilor.

Funcționarea microserviciilor:

- utilizatorul trimite cereri din aplicația web sau mobilă;
- cererea ajunge la API Gateway, punct unic de intrare;
- API Gateway o rutează către microserviciul potrivit;
- fiecare microserviciu are propria logică și propria bază de date;
- serviciile comunică între ele prin API-uri sau prin mesaje, **synchronous (direct)** prin HTTP sau REST API;
- un Message Broker poate gestiona evenimentele asincrone, **asynchronous (indirect)** prin (Kafka, RabbitM);
- sistemul răspunde utilizatorului după procesarea internă.

### 3 DOMENIUL APLICATIV

Arhitectura cu microservicii nu este doar un concept teoretic, ci stă la baza unora dintre cele mai mari și cunoscute aplicații din lume. De fapt, multe dintre platformele pe care le folosim zilnic au trecut de la arhitecturi monolitice la microservicii tocmai pentru a putea face față creșterii rapide, traficului imens și nevoii de actualizări continue.

Un exemplu cunoscut este **Netflix**, care inițial funcționa ca un monolit tradițional. Pe măsură ce numărul de utilizatori creștea rapid, au început să apară probleme: aplicația devinea greu de actualizat, erorile mici puteau afecta întregul sistem, iar traficul ridicat din anumite zone încărca întreaga platformă. Netflix a decis să treacă la microservicii pentru a putea scala doar acele părți ale aplicației care aveau nevoie de mai multe resurse, cum ar fi serviciile de streaming sau recomandările personalizate [6]. Astfel, fiecare microserviciu a devenit responsabil de o funcție exactă, iar platforma a devenit mai stabilă și mai flexibilă.

Un alt exemplu este **Amazon**. La început, Amazon funcționa cu o arhitectură monolică masivă, greu de gestionat de echipele mari care lucrau simultan la platformă. Orice modificare, oricât de mică, necesita reconstruirea unei părți semnificative din sistem, ceea ce îngreuna inovația. Trecerea la microservicii a permis echipelor să lucreze independent, un grup se ocupa de plăți, altul de recomandări, altul de stocuri. Fiecare microserviciu putea fi actualizat fără să opreasă întregul site, iar Amazon a căpătat o viteză de dezvoltare impresionantă [7].

Și platforma **Spotify** este construită pe microservicii. Pentru o aplicație care livrează muzică în timp real către milioane de utilizatori, este esențial ca diferite funcționalități, playlisturile, streamingul, recomandările, gestionarea conturilor, să lucreze independent. Prin folosirea microserviciilor, Spotify poate să actualizeze doar componenta de recomandări fără să afecteze streamingul muzical sau să repare o eroare din secțiunea de conturi fără să afecteze restul aplicației.

Chiar și în domeniul social media, microserviciile sunt foarte populare. **Facebook** și **Twitter** folosesc componente independente pentru feed-ul principal, pentru afișarea notificărilor, pentru încărcarea imaginilor sau procesarea mesajelor private. Acest lucru le permite să adauge funcționalități noi rapid, fără a afecta părțile deja existente.

Toate aceste aplicații au ales arhitectura cu microservicii din aceleași motive, nevoie de scalare, flexibilitate, viteză de dezvoltare și posibilitatea de a menține sistemul funcțional chiar dacă o parte a lui întâmpină probleme. Microserviciile le permit să evolueze constant, să adauge funcționalități noi fără a întrerupe serviciul și să ofere utilizatorilor o experiență stabilă, indiferent cât de mare devine platforma.

## CONCLUZII

Arhitectura cu microservicii s-a impus în ultimii ani ca una dintre cele mai flexibile și moderne abordări în dezvoltarea aplicațiilor software. Privind în ansamblu, putem spune că microserviciile nu sunt doar o metodă tehnică, ci o schimbare de perspectivă asupra modului în care construim sistemele, un răspuns direct la provocările lumii digitale de astăzi. Ele au apărut din nevoie reală de a depăși limitele aplicațiilor monolitice, care, pe măsură ce creșteau, devineau din ce în ce mai greu de întreținut, de actualizat și de scalat. În acest sens, microserviciile reprezintă o soluție modernă la o problemă veche, complexitatea tot mai mare a aplicațiilor și ritmul rapid al inovației.

Pe măsură ce aplicațiile au devenit mai mari, arhitectura tradițională de tip monolit s-a dovedit rigidă. O schimbare mică putea afecta întregul sistem, iar lansarea unei actualizări necesită timp, coordonare și risc. Prin împărțirea aplicației în servicii mici, independente și bine delimitate, microserviciile au reușit să ofere o flexibilitate greu de obținut în alte arhitecturi. Fiecare microserviciu poate evoluă în ritmul lui, poate fi reparat fără a opri tot sistemul, iar scalarea se poate face selectiv, doar acolo unde este cu adevărat nevoie. Astfel, arhitectura devine mai ușoară, mai adaptabilă și mai pregătită pentru schimbări.

Un alt aspect important care rezultă din această arhitectură este autonomia echipelor. Într-un monolit, echipele depind una de alta și lucrează pe aceeași componentă, ceea ce încetinește întregul proces. În schimb, microserviciile permit echipele să se organizeze ca niște grupuri independente, fiecare responsabil pentru un serviciu bine definit. Această autonomie sporește viteza de dezvoltare, îmbunătățește claritatea rolurilor și permite rafinarea continuă a funcționalităților, fără a interacționa cu restul sistemului.

Deși microserviciile aduc libertate, ele nu sunt lipsite de provocări. Organizarea unei arhitecturi distribuite implică un nivel mai mare de complexitate operațională, instrumente de monitorizare, infrastructură pentru mesaje, gestionarea comunicației între servicii și asigurarea consistenței datelor. Uneori, chiar această fragmentare poate complica procesul dacă nu este bine planificată. Totuși, pentru aplicațiile mari și în continuă evoluție, beneficiile depășesc aceste dificultăți.

Se poate observa și faptul că microserviciile au devenit punctul forte a unor platforme gigant precum Netflix, Amazon, Uber sau Spotify. Aceste exemple nu sunt întâmplătoare. Ele demonstrează că, atunci când o aplicație crește dincolo de anumite limite, microserviciile devin aproape inevitabile.

În final, microserviciile pot fi privite ca o arhitectură orientată spre viitor. Ele nu garantează simplitate din start, dar oferă un cadru în care aplicațiile pot crește natural, fără să se blocheze în propria complexitate. Într-o lume digitală în continuă mișcare, unde viteza, flexibilitatea și disponibilitatea sunt esențiale, arhitectura cu microservicii se dovedește a fi o soluție matură și durabilă. Ea transformă aplicațiile în sisteme elastice, robuste și pregătite pentru schimbările rapide ale industriei moderne. Astfel, putem concluziona că microserviciile nu sunt doar o tendință tehnologică, ci o arhitectură care modelează viitorul dezvoltării software.

## BIBLIOGRAFIE

- [1] "Microservices vs. monolithic architecture | Atlassian." Accessed: Nov. 12, 2025. [Online]. Available: <https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith>
- [2] "Monolithic vs Microservices Architecture: Pros and Cons for 2025 - Scalo." Accessed: Nov. 12, 2025. [Online]. Available: <https://www.scalosoft.com/blog/monolithic-vs-microservices-architecture-pros-and-cons-for-2025/>
- [3] "Monolithic vs. Microservices Architecture - GeeksforGeeks." Accessed: Nov. 19, 2025. [Online]. Available: <https://www.geeksforgeeks.org/software-engineering/monolithic-vs-microservices-architecture/>
- [4] "Advantages and Disadvantages of Microservices: A Guide for 2025." Accessed: Nov. 16, 2025. [Online]. Available: <https://apiko.com/blog/advantages-of-microservices/>
- [5] "Microservices Architecture Style - Azure Architecture Center | Microsoft Learn." Accessed: Nov. 18, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices>
- [6] "Netflix Microservices Architecture Guide for Tech Professionals." Accessed: Nov. 18, 2025. [Online]. Available: <https://www.yochana.com/netflixs-evolution-from-monolith-to-microservices-a-deep-dive-into-streaming-architecture/>
- [7] "Decomposing monoliths into microservices - AWS Prescriptive Guidance." Accessed: Nov. 18, 2025. [Online]. Available: <https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/welcome.html>