UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA TEHNOLOGIA INFORMAȚIEI

Lucrare de licență

DEVOPS ÎN MICROSERVICII

Absolvent
Bahrim Dragoș

Coordonator științific
Conferențiar Doctor Kevorchian Cristian

București, iulie 2023

Rezumat

Microserviciile sunt o arhitectură de sisteme distribuite ce a luat amploare în urma structurării organizațiilor către piață. Aceasta a fost facilitată de schimbări în modul în care lansăm produsele prin dezvoltarea platformelor cloud și a ce a dus la îmbunătățiri la timpul de livrare dar și a consistenței. Obiectivul este prezentarea acestor concepte și implementărea acestora prezentând modul de gândire ce influențează ciclul produsului. În acest scop, mă documentez legat de domeniu și încerc să îmi fac o idee de ansamblu asupra aspectelor ce trebuie luate în construirea unui sistem iar apoi o să aplic aceste informații. Aceasta lucrare ar trebui să servească ca un prim contact cu microserviciile dar și a practicilor DevOps.

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce vitae eros sit amet sem ornare varius. Duis eget felis eget risus posuere luctus. Integer odio metus, eleifend at nunc vitae, rutrum fermentum leo. Quisque rutrum vitae risus nec porta. Nunc eu orci euismod, ornare risus at, accumsan augue. Ut tincidunt pharetra convallis. Maecenas ut pretium ex. Morbi tellus dui, viverra quis augue at, tincidunt hendrerit orci. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam quis sollicitudin nunc. Sed sollicitudin purus dapibus mi fringilla, nec tincidunt nunc eleifend. Nam ut molestie erat. Integer eros dolor, viverra quis massa at, auctor.

Cuprins

Capitolul 1

Introducere

Domeniul informaticii, calculatoarelor și al tehnologiei informației este unul care va fi mereu în continuă evoluție. Dorința companiilor de a-și servi clienții cât mai rapid a dus la un avans tehnologic în care mereu apare ceva nou ce are ca scop îmbunătățirea proceselor actuale. Această arie și-a început dezvoltarea aproape acum o sută de ani iar avansul poate fi apreciat numai gândidu-ne la modul în care scriam cod și livram produse acum, acum zece ani și acum două zeci de ani. Se observă o diferență și la accesibilitatea unităților de calcul, actual majoritatea persoanelor au acces la un dispozitiv ce se poate conecta la Internet, ceea ce le permite să devină consumatori la diferite servicii.

Piața pentru unități de calcul a evoluat. În 1943, Thomas Watson, director la IBM menționează că „I think there is a world market for maybe five computers" (Cred că există o piață globală pentru poate cinci computere). În 1999, Michael Barr afirmă în „Programming Embedded Systems in C and C++" „One of the more surprising developments of the last few decades has been the ascendance of computers to a position of prevalence in human affairs. Today there are more computers in our homes and offices than there are people who live and work in them. Yet many of these computers are not recognized as such by their users." (Una dintre cele mai surprinzătoare dezvoltări a ultimelor decenii a fost ascensiunea computerelor într-o poziție predominantă în afacerile oamenilor. Astăzi sunt mai multe computere în casele noastre și în birouri decât persoane care trăiesc și lucrează în ele. Însă multe dintre acestea nu sunt recunoscute de către utilizatorii lor). Cele două fraze evidențiază modul cum s-a schimbat prezența computerelor în viața noastră. La început acestea erau foarte costisitoare, greu de administrat și de operat însă acestea au devenit din ce în ce mai mici iar recent aproape orice are nevoie de semiconductori datorită integrării ce le fac „smart".

Aceste afirmații se aplică și pentru produsele software, ce au ca menire să ofere asistență altor aplicații sau să fie consumate direct. Indiferent de locul în care mă duc, probabil persoana cu care aș interactiona folosește un computer, fie că îmi cumpăr ceva de la un magazin făcând o plată cu cardul sau că mi se livrează un colet iar curierul marchează pe AWB că a fost livrat. Christopher Little afirmă că „Every company is a technology com-

pany, regardless of what business they think they are in. A bank is just an IT company with a banking license." (Orice companie este o companie ce se axează pe tehnologie, indiferent de mediul de afaceri în care se află. O bancă este doar o companie IT cu licență de a funcționa ca o bancă.)

Desigur nu toate companiile sunt la fel și nu toate concurează cu toate companiile, însă chiar dacă nu ești o companie ce activează în tehnologie, probabil folosești componente tehnologice pentru a-ți îmbunătății randamentul sau să oferi clienților un avantaj față de competiție, care probabil gândește în același mod.

Însă pentru fiecare companie obiectivul este identic, să livreze clienților produse de calitate și cât mai rapid. Pentru aceasta trebuie să ne asigurăm că avem inginerii necesari pentru implementarea cererilor noi, însă momentul în care aceasta este terminată este doar începutul procesului de integrare și ulterior lansare, și uneori acesta este cel care reprezintă un blocaj din mai multe puncte de vedere ce limiteaza numarul de câte ori putem aduce o îmbunătățire produsului nostru.

În încercarea de a livra mai rapid sau pentru a ușura modul în care manevrăm un influx sau o lipsă de trafic, s-a încercat folosirea unei arhitecturi paralele asupra aplicațiilor creând arhitecturi orientate pe servicii, iar în momentul în care aceste servicii sunt concentrare pe un număr redus de funcționalități și pot fi lansate independent și să își păstreze funcționalitate putem vorba de o arhitectură bazata pe microservicii.

Însă schimbând arhitectura sau practiciile din modul de desfășurare al livrarii produsului nu este suficient pentru a asigura performanța, întrucât ambele dintre ele aduc dezavantaje ce trebuie tratate iar uneori acestea sunt mai mari decât avantajele pe care schimbarea le-ar aduce, însă implementarea corectă în locurile ce ar beneficia de o astfel de abordare poate să aducă performanțe ce nu ar fi posibile cu procesele vechi.

Afinitatea mea pentru microservicii provine de la lipsa de cunoștiințelor pentru a putea alege dintre mai multe limbaje de programare, framework-uri sau platforme. De exemplu, ce avantaje mi-ar face să aleg ca pentru server-ul meu să folosesc o aplicație în Node.JS sau una in Go? Chiar dacă putem să cunoaștem niște lucruri informative la început, cei mai buni indicatori sunt monitorizările proprii în producție. De asemenea pot fi cazuri în care îmbunătățirile aduse unei platforme pe parcurs o fac mai bună față de ce era inițial (dezvoltarea TypeScript poate să influențeze alegerea inițială a unei platforme SpringBoot ce este type-safe). Astfel, pe o arhitectură bazată pe microservicii putem să folosim limbajul potrivit pentru serviciul căruia îi aduce cele mai multe beneficii. Acest lucru se extinde și pentru baze de date. Poate în timpul evoluției serviciului apar tehnologii noi, însă având un singur lucru central ar face migrarea mult mai grea, de asemenea uneori ar trebui să facem compromisuri pe care alegerea unui alt tip de baze de date ar face să o dispară.

Dezvoltarea tematiciilor DevOps pornește de la ușurința cu care ne putem crea o bază de date folosind containere, întrucât instalarea unei baze de date pe calculatorul

personal în timp ce lucram la proiectele de facultate a făcut să am experiențe în care la finalul fiecărui an îmi reinstalam sistemul de operare doar pentru că o dată ce instalam o bază de date crea suficiente servicii, foldere care dupa o dezinstalare încă rămâneau. Folosind Docker, pot crea o bază de date cu o singură linie de cod, care în același timp să fie preconfigurată și să o pot împărtășii cu colegii de proiect ca să nu pierdem timp în configurarea mediului de lucru.

Lucrarea mea va urma o structură în care prezint noțiuniile teoretice la început iar la final încerc să aplic aspectele prezentate în crearea unei aplicații. Inițial pornesc cu prezentarea microserviciilor continuând cu diferitele moduri în care acestea pot fi lansate, imediat după cu dezvoltarea noțiuniilor de DevOps ce au ca scop accelerarea dezvoltării, iar la final o prezentare generală a modului în care îmi construiesc aplicația.

Capitolul 2

Preliminarii

Cel puțin până la momentul la care am decis ce temă să aleg pentru lucrarea mea de licență niciun curs de bază nu a abordat în detaliu tema pe care vreau să o dezvolt, din acest motiv lucrarea mea de licență nu este doar aprofundarea unui concept și crearea unei aplicații, ci încercarea familiarizării cu obiectele de lucru în același timp. Desigur, unele aspecte s-ar putea desprinde și în urma practicii în industrie dar nu a fost suficient ca să pot să consider că ma cunoștințe de bază în acest domeniu și doar încerc să găsesc lucruri cât mai specifice.

De asemenea, microserviciile sunt o arhitectură destul de nouă ce a luat amploare datorită limitărilor din punctul de vedere al dezvoltării și scalării aplicațiilor monolitice, astfel în opinia mea, este destul greu să vizualizezi anumite lucruri fără experiență la prima mână cu limitările acestea. Întrucât serviciul pe care îl dezvolt nu ar necesita neapărat o arhitectură bazată pe microservicii, ar funcția complet normal și aș reduce foarte mult din complexitate dacă ar fi dezvoltată în mod normal.

La fel și cu practiciile DevOps, acestea se axează mai mult companiilor care au nevoie de timp de penetrare a pieții foarte rapid, însă eu doar încerc să mă familiarizez cu setarea și folosirea serviciilor pentru uzurile mele proprii. Un lucru important ar fi ca ce folosesc să nu mă limiteze, de exemplu infrastructura de care am nevoie va provenii din Azure doar pentru că primesc credite gratuite ca și student, însă în absența lor probabil aș încerca lucruri cu cost minimal sau chiar nici sa nu le fac, și probabil aș avea același randamanet.

Capitolul 3

Microservicii

3.1 Concepte generale

Microserviciile sunt o arhitectură de sisteme distribuite ce a devenit populară în ultimele decenii datorită necesității scalării. Ceea ce o separă de o arhitectură orientate pe servicii este faptul că microserviciile sunt mult mai specifice, acoperind o singură parte din afacere și faptul că pot fi lansate în execuție independent, astfel microserviciile ar trebui să depindă cât mai puțin de altele. Comunicarea se face direct către interfețele expuse de fiecare, iar informațiile reținute de fiecare (de exemplu baza de date de date sau diferite fișiere) pot să fie modificate doar prin metodele de comunicare oferite ci nu direct.

Un concept important în microservicii este ascunderea de informații, astfel din exterior fiecare ar trebui să fie tratat ca o cutie neagră ce face anumite lucruri și expune anumite date, nu contează detaliile de implementare și nici tehnologiile folosite în implementare, ci doar capabilitățile fiecăruia. Acest lucru permite să modificăm microserviciul în funcție de cerințe, chiar și refactorizarea metodelor inițiale cât timp se respectă capabilitățile pe care microserviciul ar trebui să le îndeplinească.

Cel mai important aspect al microserviciilor este capacitatea de a lucra independent de celelalte. Acestea trebuie să funcționeze independent de celelalte iar dacă aducem schimbări într-unul atunci nu suntem obligați să facem schimbări în alt microserviciu, în caz contrat ar apărea dificultăți în lansare. Obținerea independenței poate să aducă contribuții majore în timpul de deployment însă implementarea este mult mai complicată datorită necesității comunicării cu alte servicii.

Un alt aspect de bază este acoperirea unui singur segment din afacere prin intermediul unui microserviciu. Alături de caracteristica precedentă, dacă afacerea și-ar dezvolta necesitățile ar fi mult mai dificil să modificăm mai multe microservicii și ulterior să coordonăm lansarea lor.

Independența microserviciilor este importantă, în acest scop, ele ar trebui să fie singu-

rele care ar putea să își modifice starea. Întrucât starea se referă la elementele componente, de exemplu o bază de dată, acestea nu ar trebui să fie împărtășite și nici accesate de alte microservicii. De asemenea ar trebui evitate modificarea interfețelor declarate pentru a evita necesitatea modificării altor microservicii. „ Dimensiunea microserviciile ar trebui să fie redusă prin oferirea unui număr restrâns de funcționalități declarate pentru a a fi ușor de înțeles și întreținut. Însă important este cât de multe microservicii pot fi administrate întrucât un număr mai mare de microservicii, deși mici ca funcționalități, aduc dificultăți în comunicare, scalare și depanare.

Microserviciile sunt agnostice din punct de vedere al tehnologiilor care ar putea fi folosite. Din acest motiv, acestea oferă flexibilitate însă prețul plătit este crearea a mai multor puncte vulnerabile.

Orice prezentare al acestui tip de arhitectură nu ar fi completă fără prezentarea conceptului de „monolith" (monolit), întrucât este considerată o arhitectură veche reprezentând modul de funcționare din trecut. În cel mai simplist mod de funcționare, sistemul monolitic este reprezentat de un singur proces „gigantic" ce acoperă toată funcționalitatea sistemului, în general conectat la o singură bază de date ce și ea este reprezentată de un singur proces. Putem intui problemele cu această organizare, dacă conexiunea către procesul monolitic cade sau baza de date are probleme, întreg sistemul este afectat. Microserviciile incearcă să acopere aceste probleme prin independență, astfel dacă un microserviciu este căzut, celelalte nu sunt afectate. Procesul poate fi însă separat pe module dezvoltate separat însă care la final se cuplează formând un singur proces. Există conceptul de sistem monolitic distribuit în care acesta este la fel separat în module independente însă care nu ar funcționa dacă nu sunt toate active, acest tip de sistem oferă toate dezavantajele sistemelor distribuite dar și a monolitului ceea ce îl face destul de rar întâlnit.

Chiar dacă o arhitectură monolitică prezintă dezavantaje evidente ce sunt ușor de văzut, acestea nu sunt la fel de mari pentru companiile mici. Numărul redus de aplicații ce trebuie lansate și monitorizate, posibilitatea de reutilizare a codului mult mai ușoară, obținerea infrastructurii pentru proces și uneori chiar și scalarea, deși este limitată poate să facă o arhitectură monolitică mult mai atractivă. Nu ar trebui să asociem arhitectura monolitică ca un lucru antic, ci să o considerăm ca o opțiune. Însă pentru o organizație mică, acesta ar trebui să fie doar un punct de plecare, eventual după ce serviciul oferit devine mai popular s-ar putea să ajungem să cunoaștem limitările arhitecturii și să ne orientăm tot către microservicii, însă dacă nu avem succes, eliminăm multe complexități apărute în urma introducerii microserviciilor.

Principalele avantaje ale microserviciilor provin de la baza acestora, o arhitectură distribuită, însă cuplat cu conceptul de ascundere a informației și domain-driven design (design orientat domeniu) aduc multe alte avantaje asupra altor arhitecturi distribuite. În cadrul unui sistem format din microservicii putem folosi orice tip de tehnologie pentru

program și orice tip de bază de date, întrucât ascundem implementarea de exterior. Astfel putem alege tehnologii ce aduc avantaje în dezvoltarea microserviciului. Acest avantaj duce la capabilitatea de a folosi tehnologii noi fară a afecta tot sistemul, însă limitează capacitatea de a împărtășii cod (de exemplu prin librării interne, întrucât acestea ar trebui să fie rescrice pentru fiecare limbaj). Un sistem distribuit rezistă mult mai ușor la căderi, întrucât acestea pot fi tratate ca să se prevină un lanț, însă cu un număr mare de puncte slabe devine greu de aproximat cât de mult este afectat un sistem. Scalarea unui sistem monolitic este ușoară, doar replicăm procesul, însă la microservicii scalarea poate fi mult mai concentrată pe punctele care chiar au nevoie să fie scalate. Lansarea unei schimbări într-un microserviciu este mult mai ușoară întrucât nu necesită crearea de timp mort în aplicație, întrucât nu tot sistemul este înlocuit ci doar o mică parte din acesta, ce poate fi chiar și mai controlată prin diferite aplicații de orchestrare.

Adaptarea microserviciilor vine cu un cost, pe lângă dificultățile de implementare a design-ului apar și alte probleme, de exemplu dezvoltarea locală. Atunci când dezvoltăm un microserviciu acesta, posibil, este nevoit să comunice cu alte microservicii care la rândul lor comunică cu alte microservicii și putem continua, toate acestea microservicii trebuie să funcționeze în același timp pe laptop-ul dezvoltatorului, întrucât nu putem folosi microserviciile din producție, însă nu e posibil să rulăm foarte multe microservicii pe un singur computer depinzând de configurație. Adoptarea microserviciilor, în general nu poate avea succes decât dacă este combinată cu elemente de DevOps, ceea ce înseamnă ca dezvoltatorii și operatorii să învețe tehnologii noi, ceea ce este un împediment în momentul în care vrem să lansăm un produs rapid în piață și nu avem experiența necesară. Microserviciile necesită mult mai multă infrastructură pentru a fi rulată, mai multă tehnologie ce trebuie să fie folosită pentru ca dezvoltarea și livrarea să decurgă eficient, însă pe termen lung acesta poate să fie prevăzută dacă arhitectura este folosită cum trebuie prin livrare mai rapidă și mai eficientă deci profitul ar putea să crească. Monitorizarea sistemelui devine mai complicată, într-un proces monolitic toate fișierele de jurnalizare (logging) sunt în același loc. Într-un sistem distribuit, acestea trebuie să fie colectate, și ulterior ansamblate pentru a avea sens, acest lucru devine mai dificil când intră în discuție fenomenul de replicare al unei instanțe. Securizarea unui sistem distribuit este mai grea, traficul de date se face mult mai mult prin rețele sau prin diferite căi alternative ceea ce necesită un grad ridicat de atenție. Testarea unui sistem format din microservicii este mult mai dificilă, mai ales când trebuie testate mai multe microservicii în același timp, acesta duce la creșterea timpului necesar pentru a primi răspuns. Sistemele distribuite cresc timpul de răspuns întrucât acestea nu se mai face în cadrul unui singur proces, deci datele trebuie codificate și decodificate atunci când trec dintr-un mediu în altul. În cadrul unui sistem distribuit o altă problemă care apare este consistența datelor, dată de faptul că în timpul unei cereri, datele dintr-un microserviciu se pot schimba, astfel tranzacțiile devin dificile mai ales când înainte ne bazam pe atomicitatea și consistența oferită de

baza de date.

Astfel, microserviciile funcționează cel mai bine pentru organizațiile mari ce vor ca dezvoltatorii lor să livreze facilități noi foarte repede care să nu fie întârziate de compatibilitatea cu caracteristici deja existente. Prin natura variată a acestei arhitecturi se pot folosi tehnologii noi dar și folosirea platformelor cloud, astfel se oferă multă flexibilitate cu prețul creșterii complexității. Pentru companiile mici, al căror rată de succes este greu de calculat, overhead-ul produs de adaptarea unei arhitecturi în acest stil nu ar aduce suficient de multe avantaje și ar trebui să opteze pentru o arhitectură tradițională ce este stabilă și oferă colaborare mult mai ușoară în cadrul aceluiași proiect la scale mici.

3.2 Design

Partea de creare a schemei arhitecturale a unui sistem este una dintre cele mai importante faze ale dezvoltării. Acesta este momentul în care putem descoperii că microserviciile nu ar fi chiar o alegere potrivită pentru cerințele noastre. Din acest motiv este important să nu ignorăm potențiale probleme ci să se formuleze toate problemele pe care le putem aștepta în implementare

3.2.1 Interconectare

Deși microserviciile pot fi lansate și pot funcționa independent, acestea trebuie să funcționeze în cadrul unui sistem complex. Astfel trebuie definite modul de comunicare și interfețele expuse. În definirea interconectării întâlnim câteva concepte de bază prezentate anterior ce provin din descompunerea în module.

Ascunderea de informații este procesul de separare al detaliilor de implementare pentru a evita modificarea în alte locuri în cazul în care acestea se schimbă. Acest concept ofer varii beneficii precum îmbunătățirea timpului de dezvoltare datorită posibilități de dezvoltare în paralel, creșterea întelegerii sistemului mult mai ușor întrucât fiecare modul poate să fie înțeles independent și flexibilitate oferit de faptul că putem schimba funcționalitate fără a fi necesar ca alte părți ale sistemului să fie modificate. David Parnas spune că „The connections between modules are the assumptions which the modules make about each other." (Conexiunile create între module sunt ipotezele create de module față de celelalte) [6]. Astfel, cu cât alte microservicii folosesc un anumit microserviciu mai mult se crează mai multe locuri în care devine mai dificil de modificat, astfel cu cât întâmpinăm problema aceasta mai puțin cu atât putem lansa acel microserviciu fără a lua în considerare modificări care să pastreze compatibilitatea cu elementele existente.

Un alt concept important este coeziunea, scopul microserviciilor este de a funcționa independent, astfel vrem ca lucrurile care înteracționează să fie grupate iar atunci când se schimbă ceva să nu trebuiască să modificăm multe locuri. Elementele care nu

interacționează ar trebui să fie separate pentru a nu crea probleme de regresie.

În sisteme distribuite apare cuplarea, atunci când un microserviciu depinde mai mult sau mai puțin de altul pentru a avea funcționalitate completă. Un sistem cu cuplare mică poate să fie lansat fără să trebuiască modificări în locuri multiple.

Cele două concepte definesc lucruri asemănătoare, astfel ele se pot rezuma ușor ca „A structure is stable if cohesion is strong and coupling is low." [2] (O structură este stabilă dacă coeziunea este ridicată și cuplarea este scăzută). Coeziunea se referă la lucrurile din interior iar cuplarea se referă la exterior iar acest lucru este greu de balansat.

Cuplarea poate apărea sub diferite forme, cuplare de domeniu în care un microserviciu interacționează cu alt microserviciu pentru că are nevoie de funcționalitate ce depășește atribuțiile lui. Astfel de cuplare, în general, nu poate fi evitată într-un sistem distribuit. Acesta este cel mai scăzut nivel de cuplare însă trebuie să evităm cazul în care un microserviciu depinde un centru pentru alte microservicii întrucât ar însemna ca acesta are prea multe responsabilități. Există conceptul de cuplare temporară, adică atunci când un microserviciu așteaptă ca alt microserviciu să execute ceva, astfel cuplarea se întâmplă pentru că cele două operațiuni se întâmplă în același timp. Cuplarea Pass-Through apare atunci când un microserviciu este nevoit să trimită date unui alt microserviciu întrucât acesta trebuie să trimite datele unui microserviciu la un nivel mai mic, astfel un microserviciu are rolul doar de a transmite date. Acest lucru este problematic întrucât dacă formatul acestor date ar trebui să se schimbe, ar trebui să modificăm mai multe microservicii. Evitarea acestui tip de cuplare se face prin schimbarea modului prin care comunicăm de exemplu microserviciul ce emite date ar putea să trimită direct datele către microserviciul ce are nevoie de ele sau ca microserviciul interpediar să trimită datele in format pur, fără să îl intereseze o formă anume. Cuplarea comună apare atunci când mai multe microservicii încearcă să acceseze o resursă comună, de exemplu o bază de date sau locația unui fișier. Problema în acest caz este că dacă modul în care această resursă interacționează se schimbă, mai multe microservicii trebuie modificate. Sau dacă unul din microservicii are nevoie de un comportament special, alte microservicii ar trebui să se adapteaze ceea ce crează probleme în dezvoltare. Soluții posibile de evitare pot fi crearea unui alt microserviciu ce funcționează ca o interfață pentru resursa comună ce va funcționa dupa principiile unui microserviciu. Cuplarea de conținut apare atunci când un microserviciu modifică starea internă a altui microserviciu, de exemplu prin modificarea bazei de date a acesteia. Acest lucru este periculos întrucât pot apărea modficări nedorite prin utilizare incorectă, astfel se încalcă procesul de ascundere de informații.

3.2.2 Domain Driven Design

Domain Driven Design este un concept introdus de Eric Evans în cartea sa [3]. Acesta se bazează pe crearea codului în jurul afacerilor pentru ca acesta să fie mai ușor de

înțeles prin creșterea interacțiunii cu analiștii de afaceri, acest lucru este benefic pentru microservicii întrucât acestea se pot orienta către mediul de afaceri pentru a se dezvolta alături de acesta fiind mult mai eficient.

Unul din concepte este limbajul universal ce cuprinde ca codul din interiorul unui microserviciu să folosească termeni folosiți de către utilizatori și de către analiști, acest lucru este benefic întrucât face comunicarea mult mai eficientă, o persoană nou venită poate să înțeleagă afacerea și codul în același timp. Dacă cele două sunt diferite, atunci pot exista probleme în comunicarea problemelor.

În DDD, un agregat este o unitate ce are o anumită stare ce se poate modifică în timp și are ca scop modelarea unui concept din domeniul real. În general un agregat este cuprins în întregime de către un microserviciu chiar dacă acesta poate conține mai multe microservicii. Un agregat poate fi schimbat din exterior însă poate să își controleze singur starea pentru a preveni stările ilegale

Mărginirea contextului se referă la ascunderea mai multor activități ce țin de implementare cu scopul de a defini anumite responsabilități sigure pe care le îndeplinește. Acest lucru se poate îndeplini prin ascunderea modelului, atunci când vrem să expunem o entitate nu trebuie să oferim toate informațiile despre acesta ci doar cele necesare. Distribuirea unui model, adică a unei entități între mai multe microservicii se face prin traducerea acestei entități în domeniu propriu.

DDD în microservicii este un concept fundamental întrucât ambele pun la centru afacerea și dezvoltarea în jurul ei. Astfel conceptele prezentate în DDD pot fi aplicate microserviciilor mărginirea contextului are ca bază ascunderea de informații și duce la crearea de formate standard.

3.2.3 Concepte de programare orientată obiect în microservicii

Multe din problemele cu care ne confruntăm în microservicii pot fi tratate folosind concepte ce au ca scop ușurarea modului în care creăm sisteme prin intermediul claselor în programarea orientată obiect. Deși diferite în natură, ambele au ca rol definirea unor interacțiuni între elemente ce sunt făcute să funcționeze separat sau prin intermediul unuor interfețe.

Începând cu conceptele de bază, abstractizarea se referă la ascunderea implementărilor interne și afișarea doar facilitățile oferite de clasă ce pot fi accesate din exterior. Acest lucru face referire la conceptul de ascundere al informațiilor descris anterior.

Encapsularea se referă la agregarea de date și atribute, adică funcționalități pentru a crea un tot unitar, acest principiu este compatibil cu microserviciile și nevoia acestora de a funcționa independent folosind datele proprii și permite o securizare mai bună a stării interne.

Moștenirea se referă la capacitatea unei clase de a prelua atributele și funcționalitățile

unei clase deja existente, deși mult mai greu de imaginat în microservicii, moștenirea în microservicii ne permite să creăm microservicii ce extind pe cel existent fără a cauza probleme.

Polimorfismul este un concept ce face referire la capacitatea accesarii a unor multiple entități prin intermediul unei singure interfețe, în contextul microserviciilor putem spune că un microserviciu ar putea să înlocuiască alt microserviciu dacă are aceași interfață de comunicare și duce la crearea a unor aceleași rezultate.

Pentru microservicii putem aplica și concepte mai complicate de OOP precum principiile SOLID. Principiile SOLID sunt un set de reguli cu scopul de a oferi dezvoltatorilor o metodă de a scrie cod, astfel încât acesta să respecte cerințele actuale și să permită dezvoltări/modificări ulterioare cu ușurință fără a avea dificultăți cu metodele deja existente. SOLID este un acronim iar fiecare literă face referire la un principiu. Principiul responsabilității unice (Single Responsible Principle) se referă la necesitatea ca fiecare clasă și metodă să aibă o singură funcționalitate, făcând asta reducem complexitatea dar și eventuale redundanțe create prin folosirea aceluiași cod în mai multe locuri făcând ca fiecare porțiune din cod să aibă un singur motiv de a exista. Principiul Deschis – Închis (Open Close Principle) conduce la faptul că entitățile scrise trebuie să fie deschise pentru moștenire dar să nu poată fie modificate, astfel asemănător cu principiul anterior, codul deja scris nu va fi afectat întrucât noile modificări se aduc asupra unei porțiuni noi. Principiul substituției al lui Liskov (Liskov Substitution Principle) se referă la faptul că clasele derivate trebuie să acopere funcționalitățile clasei pe care o moștenesc astfel încât acestea să poată să le înlocuiască fără modificări adiționale. Principiul separării interfețelor (Interface Segregation Principle), clasele ce implementează interfețe nu ar trebui să implementeze metode de care nu au nevoie, astfel dacă este nevoie să adăugăm o metodă nouă unei interfețe aceasta trebuie să fie implementată doar de către obiectele care au nevoie, dacă acest fapt nu este satisfăcut trebuie ca interfețele să fie separate ca să nu forțăm alte clase să implementeze metode noi. Principiul inversării dependințelor (Dependency Inversion Principle) se referă că modulele de nivel înalt nu ar trebui să depindă de cele de nivel scăzut, ele trebuie să fie abstractizate astfel acestea pot fi modificate în orice moment dacă vrem să aducem un alt modul cu funcționalități similare dar cu implementări diferite.[1]

3.2.4 Migrarea

În planificarea urbană există conceptul de dezvoltare pe spațiu verde. Aceasta se referă la dezvoltarea pe un loc în care la momentul începerii lucrărilor este spațiu varde ce nu necesită costuri adiționale pentru curățare. Atunci când vrem să construim un sistem bazat pe microservicii pornind de la zero, apare acest fenomen. Însă de multe ori există un sistem existent ce ajunge la anumite limitări ce fac dorința de trecere către

microservicii. Acest lucru face referire la conceptul de dezvoltare pe spatiu maro, adică pe un spațiu post industrial în care există costuri de demolare sau restaurare și ce necesită o atenție mult mai ridicată datorită posibilității alterării compoziției pământului ce ar putea să fie toxic.

Migrarea unei aplicații monolitice nu ar trebui să fie făcută cu scopul a obținerii unei aplicații formate din microservicii ci pentru că există o funcționalitate ce este împiedicată de a funcționa la capacitățile necesare pentru funcționare optimă, astfel migrarea unui sistem monolitic ar trebui să fie făcută incremental și posibil nu complet dacă nu este nevoie.

Începutul migrării ar trebui să fie făcut cu o analiză a sistemului și identificarea dacă avem posibilitatea de a întreține microserviciile, acest lucru înseamnă să avem prezentă infrastructură și oameni care să știe să configureze uneltele necesare precum cele de CI/CD pentru a livra cu aceași consistență. Ulterior putem analiza sistemul pentru a observa locurile unde se formează coeziune fapt ce duce la posibilitatea creării unui microserviciu cu cod grupat. Pentru acest lucru se pot folosi unelte precum CodeScene ce pot identifica porțiunile de cod ce se schimbă împreună. Acest lucru ajută în crearea unui context mărginit și să identificăm modul în care acesta ar funcționa cu exteriorul.

O dată ce identificăm lucrurile ce vrem să le separăm putem separa funcționalități și elementele conținute, UI-ul, backend-ul și baza de date. Am fi tentați să ignorăm o parte din aceste componente însă microserviciile funcționează cel mai bine atunci când toate acestea sunt separate prin simplul motiv ca chiar daca UI-ul nu este 1:1 cu backend-ul, acestea tot vor fi modificate în același timp. În acest scop, există două modalități în care putem face migrarea, pornind de la cod, însemnâand că separăm funcționalitatea însă în continuare folosim datele din baza de date monolitică. Acest tip de abordare este destul de simplu, întrucât obținem rezultate bune pe termen scurt, însă există limitări întrucât este posibil să nu putem separa datele din baza de date, fapt ce înseamnă că munca depusă pentru pentru separare ar fi inutilă. Separarea datelor mai întâi se referă la crearea unei baze date separate pentru datele pe care vrem să le folosim, acest tip de abordare aduce mai multe probleme la început întrucât ne face să ne gândim la probleme precum pierderea integrității datelor.

Atunci când facem modificări de tipul acesta trebuie să avem grijă cum orchestrăm schimbările. O modalitate ce provine de la un arbust tropical „Strangler Fig” se referă la crearea unui strat de intercepție ce verifică dacă funcționalitatea apelată a fost implementată într-un microserviciu, și daca este atunci va ruta traficul către microserviciu. Acest tip de implementare permite să nu modificăm aplicația monolitică și să putem să ne întoarcem la ea în cazul în care avem probleme. De asemenea, dacă datele sunt comune putem folosi load balancing între cele două la nevoie. În rularea paralelă, microserviciul și aplicația monolitică funcționează în paralel și ne permite să le comparăm ca să ne asigurăm că avem funcționalitatea dorită. Metoda comutatorului ne permite să avem cele două

sisteme aplicația monolitică și microserviciul să funcționeze în același timp și să comutăm între cele două foarte ușor printr-un proxy.

În momentul în care începem separarea datelor, apar probleme ce nu pot fi prezise cauzate de lipsa unei baze de date monolitice. De exemplu, performanța unei baze de date este extrem de bună când este vorba de crearea de cereri ce conțin operații de alăturare de tabele, astfel noi trebuie să mutăm aceste operații extrem de optimizate în interiorul microserviciilor. O dată cu lipsa unei baze de date comună pentru toate tabelele apar problema lipselor de constrângeri, nu ne mai putem baza pe baza de date să forțeze aceste constrângeri și trebuie să verificăm și să tratăm erorile în cazul în care apar erori. Tranzacțiile nu mai sunt garantate. O dată cu separarea datelor se pierde ACID-itatatea, o altă problemă a microserviciilor atunci când avem tranzacții distribuite între mai multe microservicii. În funcție de necesități ar fi necesar să rulăm cereri direct în baza de date sau să oferim acces către persoane din exterior către aceasta, însă prin conceptul de ascundere de informație acest lucru nu ar fi posibil. Ca să rezolvăm această problemă putem crea o copie a bazei de date, read only, pe care microserviciul o va popula o data ce populeaza baza de date inițială.

Astfel, migrarea unui sistem monolitic este extrem de dificilă și vor apărea probleme, din acest motiv este recomandat să folosim o abordare incrementală în care mutăm funcționalități una câte una, acoperindu-ne obiectivul pe care ni-l dorim prin introducerea arhitecturii formate din microservicii.

3.3 Tehnici de implementare

Trecerea de la o aplicație monolitică la una bazată pe microservicii uneori nu este ușoară. Unul din principalele motive este o posibilitate de a regândii modul cum accesăm anumite elemente. Atunci când totul se află în cadrul unui singur proces, acessarea datelor devine mai grea întrucât nu avem acces la ele și uneori nu este rentabil să căutăm după ele. Atunci când apelăm o funcție local, timpul necesar pentru executarea acesteia este neglijabil, atunci când această funcție trebuie să apeleze un alt microserviciu, lucrurile se complică și trebuie decis dacă chiar avem nevoie. Un apel făcut printr-o rețea mereu va avea o întârziere, de asemenea cresc locurile în care acesta poate eșua. În dezvoltarea aplicațiilor mobile întrucât ne așteptam ca aplicația să reacționeze la atingeri, nu putem să facem anumite operații pe thread-ul principal, acestea pot fi apelarea lucrurilor de pe Internet sau accesarea unei baze de date (chiar și dacă este de tipul SQLite3 fiind locală), aceasta pune în perspectivă impactul pe care acest tip de apeluri le pot avea.

Atunci când modificăm modul în care comunică un microserviciu trebuie să avem grijă la modul cum ar afecta alte microservicii, sau atunci când acest lucru este necesar trebuie să orchestrăm aplicarea acestor schimbări. Un alt concept ce într-o arhitectură monolitică se tratează ușor este tratarea erorilor, întrucât toate se întâmplă în același mediu de lucru

putem avea o viziune asupra ce poate să cadă și în ce mod. Într-un sistem distribuit, pot apărea mai multe tipuri de erori care se datorează rețelor, de exemplu crash-uri de microservicii ce necesită resetări, erori de omisie de mesaje, când încercăm să apelăm un microserviciu însă acesta nu trimite un răspund, în același timp acest răspund poate să fie mult prea rapid și să nu putem să îl recepționăm sau prea târziu și să considerăm că nu am primit, dar și erori arbitrare precum unele în care ne așteptăm la un răspund și nu primim ce avem nevoie.

Toate aceste aspecte trebuie tratate atunci când lucrăm cu microservicii, însă fiind vorba de o arhitectură paralelă ce oferă independență avem flexibilitatea de a alege ce folosim și ce nu. De aceea este important să alegem tehnologii și stiluri de comunicare care oferă performanțele dorite încercând să minizăm timpul de răspuns și tipurile de avarii ce pot apărea.

3.3.1 Tipuri de comunicare

La fel ca în comunicarea între procese sau în cadrul unui proces, există comunicare sincronă ce introduce blocare la un anumit nivel și comunicare asincronă ce permite procesele să trateze cererile în momentul în care pot, aducând îmbunătățiri ale perfomanței dar crescând dificultatea întelegeri și a modurilor în care aplicația poate cădea.

Cel mai simplu de înteles mod de comunicare este comunicarea sincronă bazată pe cerere și răspuns. În cadrul acestuia un microserviciu face o cerere(indiferent de formă) către un alt microserviciu și așteaptă un răspuns. Acest scenariu este extrem de familiar, semănând cu cererile către o bază de date sau către un API extern, ceea ce le face ușor de implementat însă nu este o tactică eficientă în cadrul unui sistem distribuit întrucât aduc dezavantaje precum apariția acestui blocaj, creșterea latenței dar și apariția acestui cuplaj temporar întrucât se poate întâmpla ca unul dintre microservicii să nu fie același ca cel care a răspuns întrucât a fost necesară o resetare. Acest tip de comunicare poate fi folosită pentru microservicii simple dar trebuie evitat crearea unui șir întreg de cereri întrucât crește timpul total de răspuns dar și apariția unui eșec în cascadă.

Însă, o arhitectură paralelă are posibilitatea creării a unui tip de comunicare asincronă ce nu blochează funcționarea microserviciilor, adică ele nu așteaptă un răspuns direct. Avantajul unui tip astfel tip de comunicare este posibilitatea tratării operațiunilor ce ar dura mai mult întrucât se elimină conceptul de cuplare temporară. Dezavantajul este creșterea nivelului de complexitate și creșterea dificultății de monitorizare și de a manevra erorile, însă ar trebui să fie utilizată atunci când avem procese de lungă durată sau pentru a comunica cu mai multe microservicii în același timp.

Cel mai simplu mod de a implementa comunicarea asincronă este crearea de resurse partajate. Un microserviciu adaugă date, fie într-o locație de stocare, adăugăm un fișier sau la o bază de date adăugăm alte date, iar celelalte microservicii ce vor să acționeze în

urma primirii acestei informații doar verifică apariția de date noi, iar atunci când apare o prelucrează. Dezavantaje includ necesitatea tratării cazurilor în care microserviciile ce scanează sunt căzute, iar în momentul în care vor reveni sunt nevoie să trateze date noi dar și faptul că se realizează o formă de cuplare asupra domeniului partajat. Această formă de comunicare poate fi folosită atunci când suntem restricționati de tehnologiile folosite, întrucât orice tip de sistem poate să citească date dintr-un fișier dar și atunci când vrem să trimitem un volum de date ridicat.

Un alt tip de comunicare este cel sub formă cerere-răspuns, acesta poate fi sincron sau asincron. În mod blocant, microserviciul ce face cererea va astepta un răspuns iar în funcție de acesta își va continua activitatea. În mod asincron, se va proceda în mod asemănător însă de data aceasta nu se va mai aștepta un răspuns, sau cel puțin nu unul cu informația cerută. Acest lucru este mai dificil, întrucât pentru a răspunde cererii, microserviciul trebuie să trimită unui microserviciu specific răspunsul creat și există posibilitatea ca microserviciul ce inițiază cererea să nu mai existe astfel am putea încerca ca orice microserviciu să poată să trateze orice cerere. Putem folosi acest mod de implementare atunci când avem nevoie să avem o confirmare la cererile făcute și de asemenea este destul de ușor de implementat, cel puțin varianta blocantă.

Una dintre comunicările reprezentative pentru microservicii este cea făcută prin evenimente, este asincronă în natură și se bazează că atunci când în cadrul unui microserviciu se întâmplă un lucru pe care consideră că ar trebui să îl comunice în exterior, acesta emite un eveniment cu diferite informații astfel celelalte microservicii pot asculta iar în momentul primirii acestea vor reacționa independent. Este un opus al modelului anterior întrucât în acest tip de comunicare, în general, nu avem cunoștințe dacă evenimentul a ajuns către celelalte părți sau dacă cineva ascultă pentru acest tip de evenimente ci eventual dacă a fost procesat de către Event Bus. Comunicarea prin evenimente oglindește modul de organizare al unor echipe autonome din cadrul companiilor, acestea anunță atunci când produc un rezultat nou și nu primesc feedback direct dacă s-a acționat după mesajul lor în mod implicit, întrucât comunicarea poate fi un mail general la o listă de abonați, aceștia ar primi doar confirmarea că serverul ce procesează mail-urile a validat cererea. Implementarea constă în crearea unei metode de a trimite evenimente și de a recepționa. Un prim mod este folosirea de programe specializate precum Apache Kaftka sau RabbitMQ ce livrează platforme dar și implementări pentru ambele probleme ce vin cu avantajul că toate problemele care le-am avea precum garantarea livrării și permit scalare însă sunt un sistem în plus de administrat. Un alt mod de implementare include folosirea specificatiilor declarate in Atom pentru a crea un flux web de resurse ce poate fi folosit pentru a trimite și consuma evenimente însă nu oferă latență mică. În general, ar fi bine să nu implementăm comunicarea prin evenimente de la zero întrucât este dificil de administrat și sunt multe probleme greu de tratat, de aceea putem folosii produse existente. Avantajul unui astfel tip de comunicare este posibilitatea ca un eveniment sa

fie tratat de mai multe microservicii in acelasi timp, astfel evenimentul trebuie să conțină informații suplimentare pentru ambele microservicii, deci apare riscul de a distibui mai multe informații ceea ce induc diferite probleme de securitate, astfel putem folosii acest tip de arhitectură acolo unde putem profita de acest avantaj.

Deși comunicarea asincronă aduce îmbunătățiri în decuplarea sistemelor și în timpul de răspuns în cazul în care vrem să comunicăm cu mai multe evenimente acesta aduce complexități ce devin greu de depanat, astfel avem nevoie de monitorizare și moduri de identificare a modului evoluției cererii implementate corespunzător.

3.3.2 Tehnologii de comunicare

Am menționat anterior că microserviciile ne cumpără flexibilitate, însă din acest motiv avem și mai multe probleme întrucât mereu când alegem un tip de tehnologie trebuie să limităm dezavantajele și în același timp să maximizăm avantajele pe care le poate aduce. Astfel putem să alegem tehnologii mai vechi care au fost testate dar limitate sau tehnologii noi care promit că aduc îmbunătățiri dar e posibil să nu fie lipsite de erori ce s-ar rezolva în timp.

Când alegem o tehnologie, trebuie să avem grijă că aceasta trebuie să fie compatibilită cu serviciile pe care le avem, de asemenea trebuie să nu ne limiteze în alegerea tehnologiilor viitoare. În acest scop modul de comunicare al microserviciilor ar trebui să fie clar definit întrucât nu putem fi siguri ce elemente sunt folosite și care nu. De asemenea tehnologia aleasă nu ar trebui să inhibe comunicarea cu alte microservicii.

Remote procedural call (RPC) (Apel procedural la distanță) este un mod de apelare de funcții local însă care se execută într-un serviciu la distanță. Pentru a face asta posibil sunt necesare definirea unor scheme explicite precum SOAP sau gRPC definite ca limbaje de definit interfețe (interface definition language). Definirea unor scheme explicite face ca folosirea tehnologiilor diferite să fie mai ușoară însă dacă nu vrem să folosim astfel de scheme putem folosi Java RMI care necesită folosirea aceleași tehnologii însă fără schemă definită. Folosind o astfel de tehnologie ne permite generare de cod pe care clientul le poate folosi pentru a comunica cu serverul făcând implementarea pe partea clientului mai ușoară întrucât nu trebuie să facă ceva în mod special. Limitările acestei tehnologii reprezintă restricționarea tehnologiilor folosite (mai ales în cazul Java RMI) însă celelalte menționate funcționează cu diferite tehnologii. Ascunderea faptului că prin aceste metode se face un apel peste Internet poate fi înșelător și ascunde complexitatea. De asemenea atunci când producem modificări asupra codului serverului trebuie să generăm codul pentru client, însă dacă adăugăm sau eliminăm câmpuri din interiorul funcțiilor, clientul trebuie să facă mai multe modificări deși el poate să nu folosească acele părți din cod. Acest tip de abordare este potrivit pentru comunicarea blocantă între doua sisteme asupra căruia știm când se modifică lucrurile și în ce fel, astfel cunoaștem cât de multe modificări vor trebui

să fie făcute.

Representational State Transfer (REST) prin HTTP, în acest tip de arhitectură avem definite niște resurse ce pot fi accesate prin link-uri sub diferite forme ce permite identificarea și modificarea acestora în funcție de modul de accesare. Acest tip de abordare funcționează foarte bine cu verbele HTTP predefinite (GET, POST, UPDATE) întrucât știm la ce comportament ne putem aștepta indiferent de resursa pe care o accesăm. Faptul că folosim HTTP ne permite să folosim toate elementele dezvoltate pentru a îmbunătății sistemul precum caching, echilibrări de trafic, unelte de monitorizare existente dar și elemente de securitate precum autentificare și certificate. O altra trăsătură adusă în REST este Hypermedia as the engine of application state (HATEOS) ce permite să nu cunoaștem întreg URI-ul către resursă ci doar cel de început. O dată intrat pe prima intrare, aceasta va afișa alte intrari pe care putem intra și ce am putea accesa dacă intrăm pe ele. Acest lucru este util pentru că nu trebuie să definim niște căi explicite ci acestea pot fi căutate însă pentru a face asta cresc numărul de cereri pe care trebuie să le faci doar pentru a executa o singură comandă. Atunci când folosim această tehnologie nu avem o modalitate simplă de a crea librării pe care clienții serverului le pot folosi, din acest motiv s-a creat specificația OpenAPI din Swagger ce permite ca din anumite informații să generezi automat o documentație dar și cod pentru o varietate de limbaje însă nu a fost adoptat la fel de mult întrucât ar fi dificil de implementat pentru proiectele existente. Prin acest format putem trimite o varietate de formate de date însă din cauza cerințelor adăugate de HTTP atunci când creăm o cerere ar putea să împiedice performanțe ridicate atunci când nu avem date multe de trimis. Putem folosi acest mod de comunicare cam în toate situațiile în care avem nevoie de comunicare unu la unu între servicii, fiind un stil comun de comunicare este suportat de majoritatea tehnologiilor deci ne garantează ca am avea compatibilitate.

Ambele tipuri de apeluri au aceași problemă, în unele cazuri ar trebui să facem mai multe apeluri pentru aceași resursă întrucât nu avem suficiente date, sau în cazul în care un câmp dintr-o cerere se schimbă acesta ne-ar forța să modificăm în mai multe locuri. Astfel de probleme sunt rezolvate de GraphQL ce permite să specificăm exact ce câmpuri avem nevoie fără a fi nevoie să facem mai multe cereri. Deși ar ușura cererile la nivelul clientului, dezavantajele apar la server, în primul rând acesta este un Query Language, deci dacă cererea este complicată atunci server-ul va trebui să o proceseze, dacă se adună multe cereri de tipul acesta ar scădea performanța. De asemenea, întrucât de fiecare dată se face un query, caching-ul este mai dificil de realizat și nu poate fi folosit la fel de eficient pentru a scri informații.

În arhitectura bazată pe evenimente putem folosi agenți de mesaje (Message Brokers), aceștia funcționează ca un intermediar pentru comunicațiile între microservicii, astfel în loc să comunice direct acestea comunică prin intermediar trimițând orice tip de mesaje. Acestea oferă diferite facilități în modul în care trimit mesaje, printr-o coadă în care se

pun mesajele iar consumatorii doar preiau direct din coada aceasta sau prin topică ce permite consumatorilor să se aboneze la un anumit tip de mesaj iar atunci când un mesaj vine ce îndeplinește criteriul, acesta va fi trimis la toți abonații. Principalul motiv pentru care am folosi o astfel de tehnologie este pentru că oferă diferite caracteristici care ar fi greu de implementat precum garanția livrării, astfel dacă nu există nici-un microserviciu care să accepte un mesaj, atunci el va fi livrat la primul care va fi disponibil să prelucreze acel mesaj. Alte caracteristici includ garantarea ordinei în care mesajele sunt primite sau să îți permită să trimiți același mesaj către mai multe topice.

Asupra acestor diferite moduri de comunicare apar diferite moduri de a trimite un mesaj, referindu-ne la compoziția acestora. Astfel, acestea pot fi sub formă de text sau sub formă binară. În general, putem trimite sub formă text adică JSON sau XML însă dacă o să ne intereseze să eficientizăm viteza atunci va fi necesar să encodăm mesajele trimise și să le trimitem sub formă binară însă acesta nu este un proces ce ar aduce suficient de multe îmbunătățiri.

Atunci când am vorbit de comunicarea între microservicii am menționat importanța creării unei interfețe sau a unei scheme pentru a afișa modul cum comunică microsericiul cu exteriorul. Acest lucru permite consumatorilor să știe la ce să se aștepte și să implementeze mult mai ușor crearea legăturii iar pentru cei ce administrează microserviciul ar cunoaște ce câmpuri sunt expuse și ce trebuie să facă. Deși unele tehnologii necesită prezența unei scheme sub o formă sau alta, schemele ne ajută și în crearea unei documentații consistente ce ajută la implementare. Cu ajutorul unei scheme putem găsi probleme mult mai ușor, asemănător cu modul în care funcționează limbajele de programare ce necesită specficarea unui tip atunci când declarăm variabile și limbajele interpretate ce pot avea orice valoare, avem un nivel în plus de siguranță, cel la nivelul la compilare, astfel într-un limbaj în care acest tip de verificare ar lipsii, unele probleme apar doar în producție. Atunci când trebuie să modificăm o schemă aceasta se poate face în doua feluri, structural când câmpurile se schimbă, astfel clienții trebuie sunt nevoiți să administreze noul câmp, sau semantice atunci când câmpul nu își schimbă numele însă modul de calculare este diferit. Acest ultim tip este destul de periculos întrucât este ușor de ignorat dacă nu apar probleme evidente. În general este recomandat să creăm scheme de tipul acesta, mai ales dacă microserviciile sunt deținute de echipe diferite, în cazul acesta nivelul de comunicare dintre cele două este îmbunătățit iar problemele sunt rezolvate mult mai ușor.

3.3.3 Schimbări în funcționalitate

Atunci când este vorba de microservicii, este inevitabil ca funcționalitatea acestora să se schimbe la un moment dat, iar în cazul cel mai fericit în care cuplarea și coeziunea sunt favorabile, atunci cel mai probabil nu trebuie să modificăm alte microservicii, însă de multe ori acesta nu este cazul.

O primă parte sunt metodele de evitare ale acestor probleme, cel mai simplu prin asigurarea că funcționalitatea microserviciului nu este schimbată ci doar se adaugă funcționalitate. De asemenea, putem implementa în consumatori ca aceștia să fie mai toleranți și să încerce să își găsească datele de care au nevoie chiar dacă se schimbă schema, acest lucru se poate face și prin prelucrarea strictă a câmpurilor de care avem nevoie și nu de întreg fișierul. Putem folosi tehnologii care să permită adaptare ușoară, de exemplu atunci când serializăm un obiect in Java acesta are o anumită versiune, însă dacă modificăm obiectul adăugând un câmp această versiune s-ar schimba deși câmpurile de care am avea nevoie ar rămâne la fel, acest lucru creând o incompatibilitate, însă dacă am folosi ceva asemănător cu HATEOS ar fi mult mai ușor de adaptat fără nicio modificare. Este important ca orice modificare adusă datorită unei schimbări în funcționalitatea unui microserviciu să fie prinsă cât mai rapid, de asta sunt importante testele iar cu ajutorul schemelor definite explicit putem folosi diferite unelte ce pot valida diferențe între modificări la nivelul protocoalelor, schemelor sau a specificațiilor OpenAPI.

Însă uneori este imposibil să mitigăm toate problemele și este nevoie să adăugăm schimbări pe care și alte microservicii sunt obligate să le facă, astfel apar dificultăți în orchestrarea ordinii acestor schimbări. Una dintre soluții este anunțarea schimbărilor pe care vrem să le facem și să lăsăm toți consumatorii să lanseze versiuni noi ce tolerează schimbările și după aceea să lansăm varianta schimbată. Acest lucru nu este în general suficient întrucât datorită modului de planificare aceste apariția acestor schimbări în consumatori poate să fie destul de lentă. O altă metodă de administrare a schimbărilor este de a lansa două servicii în paralel, una ce conține modificările și una care nu le conține pentru microserviciile care încă nu s-au adaptat. Acest lucru este de evitat din același motiv ca precedentul dar și faptul că crează multe dificultăți să ai mai multe servere de același timp lansate. De asemenea se poate păstra versiunea veche și doar crearea unei noi versiuni a endpoint-ului. Versiunile mai vechi pot fi schimbate să fie compatibile cu versiunea nouă și eventual retrase când suntem siguri că toți clienții au primit modificările necesare. Acest lucru se poate face prin monitorizarea traficului pe interfață. Acesta este tipul preferat de adus schimbări întrucât este cel mai convenabil pentru consumatori dar și pentru server.

3.3.4 Partajare între microservicii

Unul dintre cele mai simple concepte de programare ce aduce îmbunătțiri în scrierea codului este DRY, „Să nu te repeți” (Don’t Repeat Yourself) acesta la prima vedere are ca sens să nu duplici codul întrucât dacă ar trebui să schimbi o funcționalitate e posibil să fie nevoie să modifici în mai multe locuri din cadrul sistemului ceea ce face ca modificările să fie mai greu de făcut.

În universul microserviciilor, acestea sunt separate din natură, astfel repetarea codului

în cadrul mai multor microservicii este inevitabilă, de asta am fi tentați să creăm librării comune (de exemplu pachete de NPM sau dependințe în Maven) însă chiar dacă inițial ar fi mai ușor de scris acest lucru generează o cuplare ridicată între aceste sisteme deoarece atunci când o librărie comună este actualizată atunci sistemele trebuie să implementeze versiunea nouă, însă nu de multe ori este posibil acest lucru întrucât nu toate microserviciile ar putea fi actualizate în același timp, în acest scop trebuie să înțelegem că librăria creată de noi va fi consumată sub forma unor versiuni multiple de clienți diferiți.

Un mod avansat de a folosi diferite microservicii este crearea unei librării pentru acel microserviciu pe care mai târziu o oferim altor consumatori. Când este vorba de echipe mici, este dificil de implementat întrucât logica de server ar trebui să fie diferită de logica ce se găsește în librăria de client, când libraria ar trebui să funcționeze ca o abstractizare al API-ului pe care îl oferă microserviciul, astfel e indicat ca cele două să fie separate. La fel ca librăriile trecute trebuie să ne obișnuim cu ideea că librăria va fi consumată în mai multe versiuni.

Când lansăm microservicii trebuie să ne asigurăm că acestea pot să comunice între ele. Acest lucru, în general, este dificil de implementat întrucât microserviciile pot rula pe servere care dispar și apar din nou constant, mai ales când e vorba de infrastructură oferită ca serviciu. Sisteme de orchestrare precum Kubernetes ne oferă o metodă simplă de configurare a microserviciilor ce se află într-un singur cluster însă atunci când introducem mai multe, devine mai complicat. Însă chiar și așa nu ne putem baza pe serviciul de orchestrare să ne facă legătura întrucât dacă avem un număr mic de microservicii nu ar fi suficient de rentabil să rulăm o astfel de tehnologie.

O metodă simplă folosită și pentru Internet este Domain Name System (Sistem de nume de domeniu) prescurtat DNS. Acesta ne permite să asociem unui IP un nume, astfel nu ar trebui să cunoaștem exact IP-ul sistemului cu care vrem să comunicăm ci doar numele acestuia. Avantajele acestei metode este familiaritatea tuturor persoanelor cu acest lucru, din acest motiv ansamblarea unui sistem nu ar trebui să fie grea, însă în cadrul microserviciilor apar mai multe probleme datorită faptului că acestea pot să fie înlocuite mereu de instanțe noi. Fiecare intrare de DNS are o perioadă în care nu este actualizată (Time to live - TTL), și poate fi cached sub diferite forme. Din acest motiv putem opta pentru varianta în care păstrăm în DNS doar calea către un load-balancer sub care se află microserviciile de care avem nevoie, însă această solutie aduce problema în care nu știm exact cu cine comunicăm iar dacă unul din ele are probleme acesta nu va fi evitat.

Limitările DNS-ului au adus la apariția tehnologiilor sub nume de Regiștrii dinamice de servicii Dynamic Service Registries ce presupune ca microserviciul să fie cel care se înregistrează la un serviciu central iar pentru ceilalți ce vor să îl acceseze vor lua intrarea din cadrul acestui centru. Astfel de sisteme includ ZooKeeper ce necesită rularea mai multor noduri într-un cluster ce poate funcția ca un sistem de configurare și permite

crearea unei ierarhii în care se pot insera, modifica și interogra noduri sub diferite forme. Consul este asemănător cu tehnologia precedentă însă are mai multe facilități printre care include o interfață RESTful prin HTTP ce face integrarea cu diferite tipuri de tehnologii mult mai ușoară sau prin faptul că se pot genera fișiere de configurare ce vor fi actualizate în timp real, ceea ce permite ca programul să nu interacționeze cu Consul. O altă metodă am menționat-o anterior, adica prin Kubernetes, modul în care lucrează acesta este că prin faptul că microserviciile sunt lansate prin aceasta platformă, aceasta poate să înregistreze modul în care a evoluat și să își păstreze un serviciu ce poate identifica locația pod-urilor lansate prin el. Kubernetes oferă multe servicii fără nicio configurare ce permite să evităm platforme dedicate ca cele două menționate anterior.

Serviciile Cloud în general se împart în regiuni ce oferă servicii. În cadrul acelor regiuni putem avea mai multe resurse ce ar fi grupate sub același centru. Putem definii ca trafic Nord-Sud, traficul ce se află în afara centrului de date (de exemplu clienți) și Vest-Est traficul ce se produce în interior.

Pentru a facilita comunicarea Nord-Sud există conceptul de API Gateway (Pasarelă API) ce funcționează ca un intermediar între persoanele din exterior ce vor să acceseze microserviciile din interior. Un segment de piață care s-a dezvoltat recent este acela de a monetiza API-uri create prin diverse metode precum abonamente cu un număr limitat de cereri. Când vrem să oferim un astfel de serviciu apariția unui intermediar ce poată să îndeplinească diferite facilități precum monitorizare, limitare de trafic sau funcționarea ca un reverse proxy și poate să ofere un fel de panou de cantrol, însă în general este folosit de către companii pentru a-și accesa microserviciile din aplicațiile dezvoltate, acest lucru se întâmplă pentru că în general un API Gateway este destul de complex și uneori nu avem nevoie de toată funcționalitatea pe care ne-o oferă, de asemenea pot să fi și alte soluții care să fie mai eficiente. De exemplu, dacă vrem să externalizăm microservicii ce rulează în Kubernetes putem folosi soluții precum Ambassador. Dacă vrem un API Gateway întrucât ne așteptăm la mult trafic, atunci probabil nu avem nevoie de ceva central ci de mai multe. Un API Gateway poate funcționa însă trebuie să știm dacă se pliază cu funcționalitatea de care avem nevoie și să folosim alte tehnologii mai specializate dacă nu găsim.

Comunicarea Vest-Est se poate îmbunătății folosind Service Meshes, aceasta îmbunătățește comunicarea între microservicii prin adăugarea unui proxy, astfel microserviciile nu ar comunica direct ci fiecare ar avea propriul proxy cu care comunică iar acestea vor comunica între ele. Ele sunt administrare de servicii de control și îmbunătește traficul întrucât proxy-urile pot fi în aceași rețea locală. Acest lucru simplifică dezvoltarea întrucât microserviciile doar fac cereri normale însă defapt sunt rutate de către proxy. Deși utile, acestea induc multă complexitate în infrastructură ceea ce e bine să fie evitată.

Îmbunătățirea metodelor de comunicare între microservicii nu este suficientă pentru a crea un sistem complet pentru care este ușor să dezvoltăm. Am menționat importanța

schemelor explicite, acestea au ca rol îmbunătățirea comunicării capabilităților unui unct de acces prin prezentarea schemei însă nu este suficient, de asta e necesară documentație explicită. Standardul OpenAPI este eficient în crearea eficientă a documentației combinând cu Ambassador pentru Kubernetes poate să descopere atuomat puncte de accest OpenAPI și să afișeze documnetația automat. Documentația pentru comunicarea bazată pe evenimente nu este la fel de dezvoltată însă avem opțiuni precum AsyncAPI sau CloudEvents.

Atunci când avem multe sisteme devine din ce in ce mai greu să cunoaștem exact ce face fiecare sistem, de asta e important să știm starea fiecăruia. Folosind ID-uri de corelare pentru tranzacții putem să vedem modul cum interacționează. Folosind servicii de descoperire precum Consul putem să vedem unde rulează microserviciile, OpenAPI ne arată ce capabilități are fiecare sistem și sistemele de monitorizare ne arată starea sistemelor. Toate informațiile acestea sunt disponibile prin intermediul diferitelor API-urilor de care dispunem atunci când folosim aceste tehnologii. De asta este destul de important să avem pagini dedicate care se actualizează în timp real ce arată nu doar documentația microserviciiliro ci și starea acestora. O astfel de unealtă a fost construită la Financial Times, Biz Ops, ce oferă starea întregului sistem și calculează pentru fiecare o metrică ce ar indica dacă acestea au nevoie de schimbări. Astfel de unelte există disponibile și liber, precum Backstage de la Spotify.

O arhitectură distribuită ne permite creșterea eficienței prin distribuirea sarcinilor către mai multe sisteme de calcul. Într-o aplicație monolitică putem să obținem performanțe mai bune prin eficientizarea algoritmilor, prin folosirea thread-urilor în interiorului sau prin crearea unor procese copil, însă toate acestea rulează sub un singur procesor, ceea ce înseamnă că indifierent că se întâmplă aproximativ concurent, nu este chiar în totalitate. Însă cu ajutorul microserviciilor, acest lucru este posibil.

Însă atunci când apare un lanț lung de instrucțiuni, înseamnă că sunt multe puncte în care acesta se poate întrerupe și să aibă nevoie să se facă o analiză a lucrurilor care s-au întâmplat și să se întoarcă la o versiune sigură. Acest concept apare în contextul prezenței unei baze de date, în general relaționale însă există și versiuni care nu sunt relaționale care promit același lucru. O tranzacție reprezintă un grup de instrucțiuni ce ar trebui să se execute împreună, dacă una din ele eșuează din diverse motive, atunci toate acestea sunt anulate și ne întoarcem la un punct de recuperare de la începutul tranzacției. Acest lucru se referă la „aciditatea” bazei de date ce fac parte din acronimul „ACID” ce reprezintă un set de proprietăți ce se referă la consistența datelor dintr-o bază de date. Consistența se referă că în cadrul unei operațiuni cu baza de date, la finalul acesteia este lăsată într-o stare validă pe care se poate continua să se execute operațiuni. Izolarea se referă ca multiple tranzacții se pot întâmpla în același timp fără ca cealaltă sa fie afectată, acest lucru este asigurat de separare stărilor intermediare către o altă tranzacție. Durabilitatea se referă că la finalul unei tranzacții, datele nu sunt pierdute dacă sistemul cade.

În cadrul unui sistem format din microservicii, putem avea tranazacții de tipul acesta, în general atunci când microserviciul interacționează cu datele din baza de date proprie, iar microserviciul are control complet asupra acestora. Problemele apar atunci când modificările trebuie să fie făcute în cadrul mai multor microservicii în același timp iar asupra acestora să rămână proprietățile ce oferă consistența datelor. Acesta este un lucru mai dificil de implementat și e nevoie de mai mulți pași pentru a avea un comportament asemănător ca în cadrul unui sistem monolitic.

Cea mai simplă implementare a unei tranzacții distribuite este numită „Two-Phase Commit", (Înregistrare în două faze) aceasta presupune o parte de inițializare și execuția propriu-zisă. În prima parte, un microserviciu ce vrea să inițieze o tranazcție va trimite către microserviciile implicate o înștințare cu începerea tranzacției. Acestea verifică dacă pot să execute cerințele cerute și întorc un răspuns. Dacă toate răspunsurile sunt pozitive atunci se încep schimbările și se face commit-ul propriu zis. Din cauză că răspunsul acesta nu este livrat în același timp la toți cei implicați, înseamnă că putem să avem stări intermediare, deci se pierde izolarea datelor în timpul tranzacției.

O tranzacție implementată în acest stil are foarte multe probleme. În primul rând, garantarea faptului că un microserviciu poate să își execute partea sa din tranzacție se face de cele mai multe ori prin blocarea coloanei sau tabelului ce trebuie să fie modificat. Acesta introduce probleme atunci când avem un sistem destul de mare, întrucât trebuie să tratăm cazurile în care alte tranzacții vor să se execute dar nu pot. Dacă tranzacția aceasta este de lungă durată, atunci această blocare poate să afecteze pe termen lung. După apar probleme de tipul, ce se întâmplă dacă un microserviciu spune că poate să execute commit-ul, însă după nu primim răspuns dacă s-a executat sau nu, asta ar aduce multă complexitate pentru a anula tranzacția și e greu să garantăm că nu ar exista pierderi de date. Din cauza complexității și a multor probleme de acest tip, e bine să fi evitate. În general putem să facem asta grupând elementele care ar avean evoie de o tranzacție distribuită și să le punem într-un singur microserviciu.

Chiar și în aplicațiile monolitice apar tranzacții care se desfășoară pe o perioadă mai lungă de timp, numite și long-lived transactions, acestea cauzează probleme întrucât pot bloca tabele pe o perioadă lungă de timp. În „Sagas"[4] se descrie un mod de prevenire a acestor tipuri de tranzacții prin separarea unei singure tranzacții în mai multe astfel încât acestea să poată să fie realizate separat pentru a evita blocările. Deși inițial a fost formulat pentru baze de date normale, se aplică foarte bine pentru microservici, însă se pierde atomicitatea generală a tranzacției, chiar dacă fiecare componentă este în continuare „acidă".

Atunci când facem o tranzacție prin acest mod, se fac mai multe etape în funcție de câte microservicii sunt implicate, fiecare sub tranzacție fiind o etapă. Atunci când o operație eșuează într-o tranzacție unică, se anulează tranzacție. În acests mod de operare fiecare etapă care a rulat trebuie să fie anulată, acest lucru se face lansând operații de

anulare pentru toate etapele care au fost efectuată.

Acest mod de operație ne permită să tratăm erorile care apar din cauza erorilor care apar din motive condiționate de date și nu tehnic. Există două tipuri de anulare, cele care încearcă să anuleze tranzacția și cele care încearcă să compenseze tranzacția. De exemplu atunci când încercăm să anulăm tranzacția lansăm operații de anulare însă acestea nu pot fi identice ca cele care le-au creat pentru că aceste operații pot conține pași care nu pot fi anulați (de exemplu trimiterea unui mail), în cazul acesta operația de anulare ar trebui să fie o înștiințare că mail-ul a fost o greșeală, astfel acest tip de anulare este doar semantic. Compensarea unei tranzacții apare atunci când apare o eroare însă în loc să anulăm tranzacția, încercăm să continuăm cu ea însă în mod compensator pentru eroare care apare.

În general, anularea unei tranzacții de lungă durată este complex întrucât pot să existe cazuri în care nu se poate ajunge înapoi la starea inițiala, deci pot exista tentative de anulare, de exemplu mutarea pașilor care sunt cei mai posibili să eșuează sau cei care ar fi mult mai greu de recuperat la început, astfel operațiile compensatoare sunt mai ușor de executat.

Implementarea unor astfel de tranzacții se poate face în mai multe moduri. Saga-urile orchestrare apar prin crearea unui microserviciu central care va trimite comenzile ce implementează etapele din cadrul tranzacției asupra căruia are control total. Acest mod de implementare duce la crearea unui cuplaj intre aceste microservicii și are la bază comunicarea bazată pe cereri-răspunsuri ce în general face ca tranzacțiile să fie aproape secvențial. Saga-urile coregrafiate au la centru paralelismul arhitecturii, astfel nu mai există comenzi trimise de un microserviciu central ci acestea emit și reacționează la evenimentele pe care fiecare etapă le crează. Această abordare ne permite ca doua microservicii să consume același eveniment însă problemele apar la partea de monitorizare, datorită naturii distribuite nu mai avem implicit o metodă să vedem ce se întâmplă, astfel putem crea un microserviciu ce ascultă toate evenimentele și asociem un ID de corelare folosit pentru a grupa acțiunile din cadrul unei tranzacții.

Ambele abordări ale saga-ului dar și tranzacțiile distribuite în două faze aduc complexitate sistemului nostru, cel mai bine e să fie evitate dar dacă este nevoie atunci ar trebui să o alegem pe cea care se pliază cel mai bine cu modul în care operația ar avea sens, iar dacă ar trebui să alegem modul de derulare atunci ar trebui să încercâm să fie cât mai asemănător cu stilul pe care vrem să îl implementăm și în cod.

3.4 Build

Partea de construire a executabilului este o parte importantă din dezvoltare. Pentru limbajele compilate este un moment în care putem să vedem dacă avem erori ce în general sunt destul de minore însă sunt prinse încă de la începutul rulării ceea ce ne permite să

le rezolvăm rapid, având feedback aproape instant de la compilator.

În aplicațiile monolitice, crearea executabilului este destul de ușoară întrucât există un singur produs ce se află într-un singur loc ce trebuie să fie lansat, astfel avem un număr de executabil minim. În contextul microserviciilor, fiecare microserviciu are „executabilul" propriu, însă lucrurile se complică atunci când luăm in considerare că fiecare microserviciu are și o bază de date locală pentru el, astfel aceasta trebuie să vină la pachet și să fie configurată și executată.

Un concept devenit standard este integrarea continua („Continious Integration") - CI, aceasta presupune ca atunci când lucrăm într-o echipă, codul cu care lucrăm să fie mereu merge-uit într-un loc central pe care se rulează teste automate pentru a verifica că nu există probleme la integrare, iar dacă există atunci acestea ar trebui să fie principala prioritate. Acest concept își are baza în Agile și partea în care vrem ca mereu produsul să fie într-o stare în care poate fi livrat. CI-ul presupune să avem artefacte create din codul ce este merge-uit, astfel beneficiem de un istoric al build-urilor ce ne permite să testăm pe versiuni anterioare în cazul în care apar probleme.

Necesarul pentru un sistem de CI eficient este să integrăm codul frecvent, ideal zilnic, pentru a putea valida că codul rulează bine împreună, acest lucru făcându-se prin teste automate pentru a asigura că modificările aduse nu aduc schimbări de comportament, fiind validat prin teste dar și corectarea codului ce face ca testele să nu funcționează, acest lucru este necesar întrucât cu cât mai multe modificări aducem unui produs într-o stare stricată, va fi mai dificil de depanat.

Un mod de dezvoltare a caracteristicilor noi este de a crea branch-uri separate pentru fiecare. Dacă cerința pe care încercăm să o dezvoltăm necesită mult timp, asta înseamnă că nu o să integrăm codul decât la finalizare, ceea ce nu este în corcodanță cu principiul integrării frecvente. Astfel, e mai indicat să fie un singur branch în care cei care dezvoltă adaugă schimbări frecvent ceea ce evită problemele de integrare, frecvent numit și „trunk-based development", însă nu este perfect întrucât modificările făcute de alții sunt mai puțin vizibile, întrucât pull-request-urile sunt făcute ca o atenționare asupra modificărilor care vor apărea în cod, din acest motiv problema cea mai mare este să așteptăm primirea acceptului la pull-request-urile lansate.

Integrarea continuă se face prin intermediul pipeline-urilor care sunt configurate prin intermediul anumitor unelte care sunt făcute special ca să îndeplinească aceste cerințe, în cadrul lor se rulează mai multe etape în care se face o parte din procesul de construire al artefactului. Acestea sunt definite cu ajutorul aplicației de CI. Cel mai important pas este cel în care se rulează testele. În general este bine ca testele care rulează repede să fie executate la început întrucât cu cât feedback-ul pipeline-ului este mai rapid, cu atât fixarea lui va fi mai rapidă. În cadrul pipeline-ului se crează un artefact asupra căruia se vor executa restul pașilor.

Un alt aspect de considerat este modul de organizare al codului sursă, când avem o

aplicație monolitică aceasta nu are motiv să fie despărțită în mai multe repo-uri, însă în cadrul microserviciilor avem opțiuni.

Cea mai simplă alegere este să nu facem nimic special și doar să punem toate microserviciile într-un singur director, iar la final avem un singur build. Din punctul de vedere al unui operator, acesta trebuie să lanseze cod și să configureze un singur loc, iar un dezvoltator indiferent de cât de multe microservicii ar schimba, acesta face un singur commit. Însă o astfel de arhitectură face lucrurile simple la început dar complicate la final. Având un singur build, timpul pentru feedback crește întrucât se pot rula teste și pentru microservicii care nu sunt schimbate. De asemenea, nu știm ce microservicii ar trebui să fie lansate și care nu întrucât la final ce ne face să lansăm totul în același timp. Astfel, putem să începem cu o astfel de organizare însă dacă vrem să avem un sistem de succes, eventual va trebui să avem o altă abordare.

Într-o arhitectură distribuită formată din mai multe programe, probabil primul gând ar fi să avem o sursă pentru fiecare microserviciu, numit și multirepo este un mod de organizare în care fiecare dintre acesta are spațiul și pipeline-ul propriu, astfel se produce o separare foarte ușoară între organizare și se cunosc echipele ce lucrează la fiecare microserviciu. Însă simplitatea acestei separări aduce probleme în alte părți. Reutilizarea codului devine mai complicată, întrucât fiecare microserviciu va depinde de librării specializate, dacă microserviciul de la care pornește această librărie face o extensie atunci pentru a lansa caracteristicile noi trebuie să așteptăm ca microserviciile ce implementează librărie să aducă o versiune actualizată. Arhitecturile distribuite contribuie la un tot unitar, deci uneori o modificare trebuie să fie făcută în cadrul mai multor microservicii, în acest caz trebuie să avem acces la ambele repo-uri, să facem commit-urile în ambele și după să actualizăm și UI-ul ce consumă aceste microservicii. Dificultatea modificărilor crește mai ales dacă e nevoie să fie implicați mai mulți oameni, însă dacă avem astfel de probleme e posibil ca modelul de microservicii să fie prea cuplat. Folosirea unei astfel de organizări merge suficient de bine indiferent de nivelul la care ne aflăm însă dacă facem modificări des în mai multe locuri probabil ar trebui să facem câteva modificări la modul de desfășurare.

Un alt mod de organizare este de avea tot codul sursă într-un singur loc. Acesta sună asemănător cu prima variantă prezentată însă aceasta este configurată și se cunoaște faptul că în cadrul ei sunt mai multe proiecte. În primul rând, serverele de integrare continuă ar începe să diferențeze între proiecte în funcție de poziția lor în director, astfel nu ar mai exista probleme cu build-urile de lungă durată întrucât s-ar face per proiect modificat. Un avantaj al acestui mod este faptul că reutilizarea codului este mult mai ușoară întrucât fișierele s-ar afla într-un loc foarte apropiat, singura dificultate ar fi să includem aceste fișiere în serverele ce crează build-urile. Totuși separarea între echipe nu este la fel de evidentă, pe GitHub există un tip special de fișier numit CODEOWNERS în care putem să specificăm pentru fiecare folder cine deține elementele, acest lucru ajută întrucât cei

menționați vor fi adăugați automat la recenzie atunci când se face pull request. Acest tip de pattern are dezavantajul că ar consuma extrem de multă memorie, în același sens ar fi destul de greu de descărcat mai ales când nu avem nevoie. În acest scop companiile mari își creează unelte proprii pentru a rezolva aceste probleme, precum Google ce folosește unelte de versionare a codului propriu, Piper, astfel acest tip arhitectural funcționează foarte bine pentru companiile mari ce pot investi în îmbunătățirea acestuia dar și pentru echipe foarte mici în care separarea între proiecte este ușor de făcut.

3.5 Testare

Testarea este o parte importantă în dezvoltarea produsului, avantajele creării de produse sustenabile ce nu conțin defecte sunt destul de evidente întrucât clienții pot aprecia faptul că aplicația nu se strică și pot construi o reputație în acest sens. Pe de altă parte, apariția unei probleme majore în produs poate fi extrem de costisitoare în funcție de momentul în care aceasta este găsită, astfel este important ca în procesul creării defectele să fie prinse și rezolvate din timp.

Testarea poate să aibă o reputație scăzută fiind asociată ca ceva repetitiv ce trebuie făcut manual însă aceasta a avansat și se face în diferite moduri inclusiv automat astfel poate să fie la același nivel cu dezvoltarea însă necesitând un alt set de abilități și un mod de gândire diferit.

Însă acest tip de testare se referă la oamenii dedicați pentru asigurarea calității produsului, deci echipa de QA. Însă ei nu sunt singurii cu responsabilitatea de a monitoriza calitatea sistemului, acesta pornește direct de la dezvoltator. În general, un tester poate să nu aibă acces sau să nu se uite la cod, numit si black-box testing, în timp ce primind acces se numește white-box testing iar atunci când avem acces parțial, gray-box testing.

Testele pot urmării lucruri diferite, astfel conform cărții „Agile Testing" de Lisa Crispin și Janet Gregory putem împărții testele în funcție de ce încearcă să descopere. Testele de acceptare testează dacă funcționalitatea este conform specificațiilor. Testarea exploratorie se face în general manual și încearcă să verifice dacă există probleme în implementare. Testele unitare sunt automate și verifică dacă nu s-a stricat funcționalitatea funcțiilor în urma unor modificări noi. Testele de proprietate verifică performanța sistemului și capacitatea acestuia, se face prin folosirea de unelte specializate și urmărește timpul de răspus, scalabilitatea sau reziliența.

Putem grupa aceste categorii și să obținem mai multe clasficări, testele de afaceri ajută alte persoane să înțeleagă calitatea produsului (TU, TE) și testează o parte mai largă din produs, testele ce vizează tehnologia sunt mai rapide și au componente care sunt automatizate (TU, TP). Pentru a observa calitatea produselor se pot folosi teste care au ca scop asistența în dezvoltarea pentru a nu apărea regresii (TU, TA) dar și unele care critică produsul în funcție de performanță sau adaptibilitate (TE, TP). În general

testele se fac în timpul dezvoltării, însă există teste care se fac pe produsul care este în piață.

Testele de serviciu au ca scop testarea unui microserviciu, acestea evită testarea UI-ului ci doar a componentelor folosite de UI ca să aducă funcționalitate utilizatorului. Limitând testele la un singur microserviciu, testăm capacitatea acestuia iar orice comunicare cu exteriorul este mimată, astfel cauzele de eșec pot să fie strict legate de microserviciul testat Aceste tipuri de teste, întrucât nu testează componenta grafică pot sa fie la fel de rapide ca testele unitare însă întrucât pot intra în componență apeluri la baza de date sau la rețea. Astfel în implementarea acestora există necesitatea înlocuirii comunicării cu exteriorul și se face în general fie prin mimicarea datelor sau prin înlocuirea acestora. Mimicarea se referă că la orice tip de apel al unei funcții, indiferent de datele de intrare, timpul la care a fost făcut sau numărul de ori de câte este apelată acesta va returna un singur set de date. Înlocuirea se face prin replicarea funcționalității astfel putem apela funcții care procesează datele de intrare și returnează date în funcție de acestea. Putem să dezvoltăm această idee și să încercăm să creăm servere de test care furnizează această înlocuire existând unelte pentru aceasta, de exemplu mountebank ce poate să mimice date trimite prin HTTP/S, TCP sau SMTP însă nu poate să mimice comunicarea prin evenimente.

Însă chiar dacă facem aceste teste, nu putem să garantăm că putem livra microserviciul. Testele end-to-end se fac prin mimicarea utilizatorului și a diferitelor comportamente pe care acesta le-ar face, astfel verificând funcționalitatea sistemului prin interfața cu care el comunică. Acestea pot să implice multiple microservicii ce trebuie să nu aibă probleme, însă având o arie mai mare de acoperirea acestea sunt mai greu de testat iar atunci când pică nu se poate observa din prima posibila cauză. Una dintre problemele care arpare în aceste teste este nevoia de a rula mai multe microservicii in același timp, astfel pentru fiecare din aceste microserviciu s-ar rula aceleași teste. Acesta ar duce la o repetare de teste care nu s-au schimbat, astfel o metodă de a eficientiza procesul este de a combina testele end-to-end pentru mai multe microservicii in același timp.

Aceste tipuri de teste acoperă o arie extinsă de cod iar fiecare dintre acestea poate să aibă probleme, astfel pot exista în suita de teste unele teste care uneori pică deși uneori trec, cauza unor astfel de situații poate să fie probleme temporare de rețea ce cauzează un timeout. În astfel de cazuri ar trebui ca acest test să fie fixat, fie eliminat întrucât existența unui test neconcluziv poate duce la ignorarea acelui test, astfel este ca și cum acesta nu ar rula de la bun început.

Întrucât testele end-to-end acoperă mai multe microservicii apare problema asocierii responsabilității scrierii acestora. Întrucât în același timp mai multe echipe sunt implicate, acestea toate ar avea o parte de responsabilitate însă scrierea unui astfel de test include cunoașterea sistemului întreg și observarea locurilor ce pot fi mai sensibile, astfel companiile mari pot investii în crearea de echipe dedicate ce au ca rol scrierea, rularea

și rezolvarea (când este cazul) a testelor end-to-end. Însă crearea unor teste end-to-end complexe poate duce la timpuri mari de executare ale acestora, astfel feedback-ul compatibilității întârzie iar în acest timp se pot aduna mai multe caracteristici adăugate de dezvoltatori, astfel se încalcă regula menținerii o prioritate a rezolvarilor problemelor ce sunt asociate pipeline-urilor, în acest caz putem analiza suita de teste și să păstrăm doar esențialele, mai ales când un build poate să pornească suite de teste end-to-end diferite întrucât acesta este prezent în mai multe procese care se desfășoară în cadrul mai multor microservicii.

Problemele cu testele end-to-end duc la crearea unor probleme asociate cu arhitectura paralelă, întrucât acestea ar trebui să ruleze separat, atunci ar putea să fie și testabile independent, astfel putem rula teste end-to-end mai rar dacă avem diferite caracteristici care să ajute la interoperabilitatea precum interfețele explicite sau prin folosirea unor unelte precum Pact putem să creăm o schemă a comunicațiilor exterioare ce include așteptările pe care le avem de la aceștia. Acestea pot fi rulate după de dezvoltatorii microserviciului și să cunoască dacă apar probleme de compatibilitate. Acestea sunt folositoare întrucât nu trebuie să mai rulăm mai multe microservicii în același timp ci rulează ca testele unitare sau cele de serviciu în cadrul unui singur microserviciu, astfel devin mult mai rapide.

Majoritatea testelor rulează în timpul perioadei de dezvoltare, însă putem rula teste și pe produsul final. Înainte de lansare se poate lansa o suită de teste numită Smoke, ce are ca scop verificarea funcțiilor de bază, de multe ori aceasta ar trebui să ruleze înainte de testarea manuală întrucât căderea acestuia ar însemna lipsa funcționalităților de care am avea nevoie în mod normal, însă ca o ultimă măsură putem să îl rulăm alături de un test de Sanity ce conține o gamă mai largă de de teste înainte unei posibile lansări. De asemenea, alte teste în producție pot fi cele de A/B sau feature toggles ce includ lansarea unei funcționalități în două variante pentru a fi mai ușor de decis care dintre variante putem să o alegem (un caz recent a fost unul făcut de eMAG în care au testat performanța vânzărilor încercând să ascundă recenzille asupra acestora) sau Canary releases în care utilizatorii se pot întregistra pentru a primi varianta mai nouă a produsului, aceștia sunt limitați iar în cazul în care apar erori, doar o parte mică din utilizatori a fost afectată. O rulare în paralel se referă la lansarea unei funcționalități și păstrarea versiunii anterioare, orice cerere este trecută prin alte servicii și sunt evaluate anumite metrici. Ingineria haosului se referă la adăugarea de greșeli intenționat în produs și testarea acestuia dacă poate să își revină. Acestea pot fi considerate și metode de lansări de produs. Alte tipuri de teste pot fi cele care simulează un utilizator normal în timp ce produsul este în piață pentru a verifica dacă funcționalitățile rămân la fel și nu există probleme de configurări ale mediilor de lucru.

În cazul apariției a unei probleme în producție, unele companii rulează simulări și rapoarte a modului de lucru în cazul apariției unei probleme. În cazul acestora se pot definii

concepte precum timpul mediu de reparare(MTTR) sau timpul mediu între erori(MTBF). Prima dintre ele se poate rezolva ușor prin crearea unei restaurări a versiunii anterioare iar a doua prin îmbunătățirea capacității sistemului de a rezista la erori sau de a le trata. Aceste două metrici sunt la fel de importante ca și testarea întrucât ambele au același scop, asigurare calității unui produs.

Majoritatea testelor rulate vor avea ca scop testarea caracteristicilor și validarea că acestea funcționează în parametrii normali, însă datorită volatilității traficului pot apărea scenarii complexe în care pot apărea un influx de trafic, astfel există dorința de a testa după funcționalitate, adica aspecte precum performanța sistemului. În cadrul unei arhitecturi paralele aceasta poate cauza probleme întrucât într-o aplicație monolitică comunicarea se face într-un singur loc, însă datorită necesității a comunicării colective pot apărea momente în care sunt necesare crearea mai multor cereri, acestea pot cazuri în care pentru microservicii suprasolicitate să creze creșterea latenței sau chiar acestea să nu mai poată fi accesibile. Astfel pot exista mai multe teste de performanță, cele de enduranță ce testează sistemul pe o perioadă lungă de trafic dar în parametrii normali, de stres în care testăm sistemul ăn afara parametrilor normali, ale spike-urilor ce sunt creșteri mari in trafic pe perioade scurte de timp și de scalabilitate ce observă capacitatea sistemului de a servi atunci când numărul de utilizatori crește. Toate acestea ar trebui să se facă cunoscând funcționalitatea normală a sistemului pe trafic real, în absența acestei condiții putem să ne concentrăm pe direcții pe care utilizatorul nu le ia. Într-o arhitectură paralelă, la fel ca în securitate, capacitatea acestuia de a rula eficient este dată de capacitatea celei mai încete piese astfel putem crea scenarii de test în care testăm lipsa unor componente ce au ca scop creșterea rezilienței în cazul erorilor.

3.6 Deployment

Ciclul de viață al unei aplicații monolitice continuă să fie simplă și în contextul lansării acesteia. În urma trecerii prin toate fazele pipe-ului de build și posibila validare manuală acesta poate fi trimis către client sau urcat din diverse platforme de hostare, posibil să existe o perioadă în care platforma nu ar putea să fie accesibilă datorită necesității de a reporni, dacă nu folosim un proces mai amplu de lansare care ar acoperi aceste goluri. Dacă este implicată și o bază de date atunci aceasta poate să fie lansate înaintea aplicației și totul ar funcționa, dacă este invers atunci probabil aplicația tratează cazurile în care aceasta nu este disponibilă.

În cazul microserviciilor, procesul este mai complex întrucât caracteristica specficiă acestora este capacitatea de a fi lansate și a funcționa independent de celelalte, chiar dacă pot fi uneori limitate de absența celorlalte, astfel procesul menționat anterior se repetă pentru fiecare microserviciu. Totuși, fiecare microserviciu poate să fie de o tehnologie diferită, poate ca baza de date a fiecăruia să aibă un anumit tip și anumite cerințe de

lansare, astfel flexibilitatea oferită de acestea se poate resimți atunci când este timpul lansării.

Până acum, toate conexiunile între microservicii erau făcute doar logic, nu cunoșteam detalii despre modul cum acestea vor fi aranjate sau unde vor fi lansate sau sub ce formă. O caracteristică a microserviciilor este capacitatea de a scala un anumit serviciu ce simte o cerere mai mare, în acest context putem avea mai multe instanțe al aceluiași microserviciu, la fel cum am face și cu o aplicație monolitică în cazul în care am vrea să servim mai mulți clienți, mai lansăm o copie a aplicației. În acest caz, comunicarea nu se va mai putea face direct către microservicii întrucât scalarea se face automat, ci este nevoie să comunicăm cu un intermediar ce ar face rutarea către acestea care are și scop să echilibreze traficul între cele două, astfel folosim un load balancer și vom comunica direct cu acesta și nu cu microserviciul propriu-zis. În cazul comunicării cu evenimente, trebuie doar să ne asigurăm că evenimentul este consumat de un singur microserviciu iar în urma se va lansa evenimentul răspuns.

Un factor important însă care a fost ignorat în lansarea microserviciilor, este baza de date. Întrucât fiecare microserviciu trebuie să aibă propria bază de date pentru a-și ascunde starea internă și să poată să aibă control asupra tuturor operațiilor ce ar avea loc pe domeniul său, microserviciul trebuie să fie lansat împreună cu aceasta independent. Însă în momentul în care avem mai multe microservicii de același tip, adică mai multe replici, baza de date va fi defapt comună pentru toate microserviciile, adică toate replicile vor folosi ca sursă de date același loc. În acest context apar probleme cu scalarea, întrucât e posibil ca serviciul nostru să scaleze infinit, baza de date nu se va scala, astfel poate să devină un impediment în performanța sistemului, dacă nu cumva își cauzează sieși atacuri de tipul Denial of Service prin existența prea multor cereri. Însă scalarea bazelor de date este complicată, mai ales în cazul în care acestea sunt relaționale, întrucât necesită fragmentarea datelor ceea ce este greu de calculat, din acest motiv putem construi organizări precum existența unei singure baze de date pentru scriere și multiple baze de date pentru citire, în acest context putem separa traficul însă datele pe care le servim pot să nu fie actualizate. Modul de lansare al bazei de date poate să difere de cel al microserviciilor, mai ales pentru companiile existente ce au deja date, migrarea devine mult mai dificilă, din acest motiv aceștia pot preferă să folosească infrastructura existentă în loc să migreze către cloud unde ar putea avea o bază de date per microserficiu, astfel microserviciile pot avea baze de date logic diferite însă care rulează de pe aceleași server.

Atunci când se rulează un microserviu se crează un mediu lucru pentru acesta. Când acesta este în dezvoltare, programatorul are drepturi depline la locul în care acesta rulează și cu ce alte microservicii comunică, însă atunci când ajunge în producție microserviciul are alt mediu ceea ce face creșterea importanței creării unui mediu de testare cât mai asemănător cu cel pe care va rula. Un exemplu simplu sunt datele de test, întrucât acestea pot avea o anumită distribuție atunci când sunt în producție și alta atunci când generăm

aleator date, din acest motiv timpul de răspuns al bazei de date și alegerea parametrilor de indexare nu poate fi ușor testată fără să interacționăm direct cu mediul de producție, iar existența anumitor legi nu ne permite să copiem datele utilizatorilor în mediul de test cu ușurință. Astfel lansarea microserviciului trebuie să fie ușor de adaptat pentru fiecare mediu, fie el prin anumite variabile ce ar lua valorile mediului sau comportamente diferite în funcție de numărul de instanțe ale microserviciului în fiecare loc.

Lansarea și mentenanța microserviciilor este o sarcină ce necesită coordonarea mai multor echipe, în general cei care se ocupă cu dezvoltarea nu o să se implice în configurarea infrastructurii, din acest motiv există câteva principii pe care le putem urmării atunci când lucrăm cu microservicii pentru a maximiza caracteristicile pe care acestea le oferă.

Întrucât microserviciile sunt implementate pentru a rula independent, ar fi o idee bună ca acestea să ruleze independent. În general, dificultățile în crearea infrastructurii apar atunci când sistemele de calcul trebuie să fie configurate, astfel o metodă ar fi lansarea microserviciilor pe aceași mașină, în acest caz monitorizarea sistemului scade întrucât metricile asupra impactului peste componente nu ar putea fi atribuit unui anumit microserviciu, în același stil acestea nu ar putea să fie la fel de scalabile și ar fi constrânse la fel ca o aplicație monolitică. Recent, acesta a devenit mai ușor prin lansarea în container sau prin platforme precum Heroku ce izolează sistemele by default. Există mai multe tipuri de izolare în funcție de modul în care lansăm aplicația din acest motiv, cu cât izolarea este mai puternică cu atât cresc costurile asociate acelui microserviciu și dificultățile în administrare.

Toate activitățile încep manual însă în timp ce le facem putem să găsim metode de a le eficientiza, în general prin automatizare. Atunci când încercăm o arhitectură bazată pe microservicii, crește complexitatea lansării, astfel unele configurări care într-o aplicație monolitică era ușor de considerat că nu merită să se încerce să se automatizeze, în microservicii ar trebui ca tot ce se poate automatiza să fie automatizat. În general, problemele care apar pot să fie din greșeli umane care nu pot să fie evitate. Putem porni procesul de încercare a automatizării prin adaptarea de tehnologii și procese ce ne pot ajuta în acest scop.

Dezvoltarea cu ajutorul Git-ului ne permite să observăm schimbările din cod foarte ușor, putem să avem un istoric și persoanele implicate iar atunci când se crează o schimbare prin intermediul Pull Request-urilor putem fi anunțați că se dorește modificarea unei porțiuni de cod. În domeniul configurării infrastructuri aceasta se face manual, prin aducerea infrastructurii prin instalarea de computere, crearea unor mașini virtuale sau rularea unui container. Toate acestea implică munca unui operator ce ar trebui să seteze mașinile și ulterior să le configureze prin intrarea manuală prin SSH, comenzile făcute în interior, deși înregistrare nu sunt la fel de ușor vizibile ca și ceva pe Git, astfel a apărut conceptul de infrastructură ca și cod în care infrastructura este configurată prin intermediul unor limbaje ce pot fi interpretate de unelte ce vor configura pentru noi

computerele precum Puppet sau Ansible. Acestea ajută să scriem starea mașinii într-un mod declarativ iar acestea vor rula script-urile de care avem nevoie în interior. Terraform este o unealtă ce ajută în crearea resurselor ce provin din platforme cloud precum Azure sau AWS. Având infrastrctură ca și cod aceasta poate fi urmarită de toți participanți și în același timp poate fi reaplicată pentru toate resursele ce trebuie să fie configurate în același stil.

Funcționarea independentă ne permite să lansăm produse fără ca alte microservicii să necesite o actualizare, în acest context putem să facem lansări mult mai dese accelerând dezvoltarea, acestea aduc avantajul că putem face o lansare pentru un număr restrâns de caracteristici, astfel dacă apar probleme în producție putem să fixăm mult mai ușor întrucât sunt puține funcționalități schimbate. În acest context trebuie să măsurăm contextul în care sunt afectate celelalte microservicii atunci când facem o lansare, în mod tradițional sistemul ar fi oprit și ar cauza o perturbare pentru consumatori și ulterior al utilizatorilor. Această probleme poate fi diminuată cu tehnici de lansare precum lansarea treptată în care serviciile existente se actualizează una câte una în timp ce consumatorii existenți sunt păstrați iar cei noi redirecționați către cele noi. În cazul acesta apar probleme atunci când între microserviciu și client se crează conexiuni de lungă durată precum un socket întrucât acestea nu pot fi întrerupte fără a cauza dificultăți.

O altă specificație importantă este capacitatea de a putea configura o stare dorită a întreg sistemului și să folosim uneltele potrivite pentru ca acesta să se întâmple. În acest scenariu, platforma folosită va monitoriza sistemul și va aplica comenzile necesare pentru a păstra forma dorită. Printre astfel de unelte se numără Kubernetes, iar unii distribuitori de infrastructură în cloud oferă aceleși servicii. În acest context, întrucât platforma este cea care administrează starea este important să automatizăm modul de lansare, astfel platforma primește doar imaginea și o aplică. O continuare al acestui mecanism este GitOps, la fel ca infrsastructura ca și cod care prevedea configurarea sistemelor ca și cod, acest sistem prevede menținerea stării ca și cod și folosirea unor operatori automați care vor aplica comenzile din acel repository. Acest procedeu are aceleși beneficii ca și pentru infrastructură, adică capacitatea de a avea o privire mai bună per ansamblu, un istoric, anunțarea schimbărilor și interogatoriu. În acest mod de lucru configurarea manuală a infrastructurii pe care rulează sistemul nu mai este permisă, iar fiecare schimbare se va face în repo.

Locurile în care putem să lansăm microserviciile vor fi mai detaliate în chapter 4 însă la fel cum avem flexibilitate în implementare, avem flexibilitate și în lansare. Acestea pot include o lansare directă pe mașini direct sau într-o abstractizare precum un container sau ca Function as a Service (FaaS), astfel ar trebui să ne documentăm despre opțiunile pe care le avem, avantajele și dezavantajele fiecareia și ce suntem confortabili să facem pentru a administra sistemul. Ar trebui să folosim metodele existente de deployment însă dacă acestea au nevoie de modernizare atunci putem face schimbări. Cu ajutorul platfor-

mele existente precum FaaS sau containerizarea aplicației putem să scădem dificultățile în configurarea mediului și să ne concentrăm doar pe rulare. În același mod se pierde controlul asupra modului de configurare întrucât acesta va fi administrat de platformă, astfel dacă folosim servicii precum Heroku, acesta ne va oferi anumiți parametrii însă configurarea efectivă a sistemului de calcul va fi abstractizată ceea ce ne permite să ne concentrăm pe alte aspecte ale sistemului.

În cadrul implementării unei lansări eficiente s-au adunat doi termeni, livrare continuă (continuous delivery) și lansare continuă (continuos deployment). Diferența între cele două se crează la starea produsului. Cea din urmă poate fi considerată chiar și o continuare. Livrarea continuă se referă la menținerea produsului într-o stare în care poate fi lansat. Lansarea continuă se referă la automatizarea procesului din care produsul livrabil ajunge la utilizatori, astfel conceptul de intrare a unei caracteristici „în producție" poate fi considerat schimbat, astfel această stare poate fi considerată atunci când aceasta intră în produsul livrabil, deși ea nu a ajuns la consumatori. Lansarea continuă poate fi limitată de platformele pe care lucrăm dar și de modul de organizare, astfel acesta necesită crearea unei automatizări puternice atunci când dezvoltăm produsul.

3.7 Monitorizare

Necesitatea monitorizării nu apare atunci când dezvoltăm microserviciile ci atunci când le lansăm. Atunci când apare o problemă nereproductibilă, singurul lucru ce ne poate descrie modul în care am ajuns în această situație sunt urmele pe care le creăm intenționat, adică fișierele de jurnalizare sau tranzacțiile care s-au desfășurat. Într-o aplicație monilitică acesta are un număr mic de locuri în care poate să cedeze, astfel investigațiile pornesc mereu din același loc. Monitorizarea acestuia este mai simpla întrucât numărul de resurse pe care acesta le apelează este mai mic. În cazul unei arhitecturi distribuite atunci când apare o problemă, aceleași investigații pe care le-am face pe o platformă monolitică o să le facem asupra mai multor componente. Din acest motiv trebuie să găsim metode de monitorizare eficiente ce sunt ușor de accesat și folosit pentru cei implicați

Implementarea unui astfel de sistem în interiorul unui microserviciu depinde de modul în care acesta comunică. În cazurile simple în care avem o singură instanță al unui microserviciu atunci monitorizarea acestuia este simplă întrucât trebuie să colectăm date dintr-un singur loc. Atunci când acest microserviciu este scalat și există mai multe instanțe al acestuia care pot să apară sau să dispară în funcție de load, atunci trebuie implementat un sistem care colectează datele de la acestea și eventual le asociază datele asupra cererilor. Atunci când aceste microservicii comunică și cu exteriorul devine și mai complicat întrucât nu vom ști exact modul în care sistemul va acționa, în acest caz agregarea informațiilor nu este suficientă și avem nevoie de metode de prelucrare eficientă al acestora.

Observabilitatea este capacitatea de a întelege starea unui sistem în funcție de datele pe

care acesta le comunică, asemănător cu bordul unei mașini ce ar lumina anumite beculețe în funcție de problemele pe care mașina ar putea să le întâmpine în interior. Aceste date furnizate pot ajută în prevenirea unui eventuale probleme, fie prin scalarea automată a sistemului în cazul în care datele furnizate de acesta indică că este suprasolicitat fie prin identificiarea problemelor înainte să apară prin crearea de algoritmi bazați pe inteligența artificială, iar toate acestea ar trebui să se vadă dintr-un loc central pentru a putea examina starea sistemului în același mod pentru toate componentele acestuia.

În acest scop, atunci când creăm un sistem bazat pe microservicii ar trebui să pornim de la construirea unei baze pentru observabilitate, astfel putem avea în vedere anumite caracteristici care trebuie să apară. Una dintre cele mai importante și simple de implementat acțiuni este agregarea log-urilor. Dacă putem crea asta atunci putem să construim un istoric al acțiuniilor ce s-au declanșat în cadrul sistemului. Deși există unelte dedicate, toate ar lucra în același stil, microserviciile crează fișiere de jurnalizare local pe care alt componentă le va colecta și le va trimite către componenta de agregare. Este important ca acesta să nu există procesări mari ale acestor linii întrucât cantiatea de informație care apare poate să creeze creșteri mari în consumul de resurse. Fiecare intrare în aceste fișiere ar trebui să fie standardizată cu informațiile de care avem nevoie, de exemplu data, microserviciul, mașina, nivelul de importanță al log-ului și atle date de care am avea nevoie în depanare. Toate aceste date sunt folosite de agregator pentru a crea centralizări informațiilor sau pentru a cunoaște momentele în care ceva grav se întâmplă și să fie nevoie să alertăm personalul. Întrucât microserviciile funcționează pentru rezolvarea unei singuri cereri, este important să cunoaștem cererea pentru care se lucrează, în general se asociază un ID de corelare pe care fiecare microserviciu îl va comunica pentru ca tranzacțiile din cadrul cererii să fie ușor de interogat atunci când acestea vor fi agregate. Deși fiecare înregistrare are o dată, aceasta nu poate să fie sigură iar încrederea în acestea poate fi o capcană. În programare, cel puțin pentru mine, lucrul cu timpul este cel mai dificil, în interfață deoarce există fusuri orare ce trebuie tratate (desigur e ușor să salvăm datele în UTC și doar să le afișăm în mod diferit, însă se adaugă și schimbările create de economiserea de economiserea de lumină), iar în cadrul fișierelor de log ale microserviciilor timpul nu poate fi precis întrucât fiecare are timpul său intern, deși poate să fie sincronizat cu un server de sincronizare Network Timing Protocol (NTP) nu poate să fie exact. Putem folosi unelte precum Fluentd ce trimite date în Elasticsearch cu interfață în Kibana însă acest spațiu este populat cu soluții. Atunci când avem un sistem de acest tip, pot apărea probleme, pe lângă ce a fost menționat, apar costuri cu privire la stocarea datelor dar și necesitatea unei puteri computaționale în creștere pentru a menține index-urile coloanelore din bazele de date.

Un alt aspect ce comunică starea sistemului sunt metricile și telemetria. Datele colectate de acestea pot include procentul de resurse consumate, arhitectura sistemului, modul cum răspund la cereri sau chiar performanța acestora. Aceste metrici sunt în genera

simple întrucât urmăresc stocarea unei singure informații sau acțiuni, însă acompaniate de acestea apar informații suplimentare iar acestea sunt costisitor de depozitat întrucât va trebui să fie interogate. În acest scop putem folosi unelte precum Prometheus sau Graphite.

Colectarea datelor despre un sistem se face într-un singur scop, aprecierea dacă totul funcționează în parametrii nominali. În cadrul unei aplicații monolitice, această apreciere era simplă întrucât lucrurile pot să meargă sau să nu meargă. Într-o arhitectură paralelă ce se bazează pe independență această apreciere devine mai dificilă. Dacă un microserviciu rulează, este un pas, însă nu este suficient. Acesta trebuie să fie accesibil pentru toți consumatorii acestuia. Însă dacă funcționarea acestui microserviciu depinde de alt microserviciu care momentan are probleme, putem să spunem că acest microserviciu este într-o stare bună? În aceste cazuri se creează anumiți termeni, primul concept fiind Termeni la nivel de serviciu, „Service level agreement" ce se crează între cei care construiesc un sistem și cei care folosesc acest serviciu, astfel aceștia din urmă pot să își creeze un anumit grad de încredere sau nivelul la care se așteaptă. De exemplu, providerii de infrastructură în cloud pot crea astfel de termeni pentru a garanta funcționalitatea serviciilor oferite de ei, pe perioadele în care acestea nu pot fi accesibile din cauza lor, probabil nu vor exista credite consumate însă probleme ale sistemului nostru tot pot apărea ce pot afecta clienții proprii. Obiectivele la nivel de serviciu („Service-Level Ojective) sunt create pentru a crea o imagine sau un posibil progres în atingerea SLA-ului. Accestea pot include lucruri precum timp stabil sau latență. Pentru a aprecia atingerea SLO-ului putem crea Indiccatori la nivel de serviciu („Service-level Indicators") ce depind de sistemul oferit. În construirea unui sistem putem să alocăm spațiu pentru erori, acesta este scopul SLA-urilor, întrucât nu putem să fim perfecți ar trebui să păstrăm loc și pentru apariția unor erori. Acestea ajută în crearea deciziilor care includ riscuri precum înceracare de tehnologii noi.

Apariția erorilor este inevitabilă, din acest motiv trebuie să înțelegem modul în care acestea pot să fie raportate. Există erori al căror apariție poate fi ignorată pe moment, în general la aceste tipuri de erori ne putem aștepta și au o severitate redusă. În interiorul unui sistem trebuie să știm tipurile de erori care impactează funcționalitatea sistemului și trebuie remediate în cel mai scurt timp, în general pentru cazurile acestea un tehnician ar fi alertat prin diferite forme pentru a remedia problema. Întrucât astfel de situații crează un grad mare de disconfort, trebuie să alegem erorile care generează acest tip alerte cu grijă, întrucât dacă sunt prea multe e posibil ca acestea să fie ignorate. Studii create pentru a studia motivele pentru care unele accidente apar pot să concluzioneze că din cauza numărului mare de alerte raportate care nu au gradul corespunzător pot genera pentru cei care le urmăresc o oboseală, din acest motiv aceștia pot să nu urmeze procedurile corespunzătoare. Steven Shorrock menționează în articolul „Alarm Design: From Nuclear Power to WebOps" (Humanistic Systems (blog), October 16, 2015) că motivul alertelor este de a direcționa atenția utilizatorlui către aspecte ale operației ce

au nevoie de atenție într-un timp scurt. („The purpose of [alerts] is to direct the user's attention towards significant aspects of the operation or equipment that require timely attention."). Din acest motiv ne putem inspira din reguli vizează comportamente ce necesită utilaje complexe. Asociația utilizatorilor de echipament și materiale (Engineering Equipment and Materials Users Association - EEMUA) a creat o descriere pentru crearea alertelor într-un mod corect. Acestea trebuie să fie relevante, să ilustreze o caracteristică importantă a sistemului, unice, pentru a evita repetiția, să ajungă într-un timp util în care acțiuniiile utilizatorilor mai pot crea un impact, sa conțină suficiente informații pentru ca acesta să le poată prioritiza, informația ce provine din alerte să fie clară și ușor de înteles, să conțină informațiile necesare pentru a începe o acțiune cu scopul de fixare, dacă este posibil să ofere sfaturi pentru rezolvare, și să fie concentrate pe problema pe care o raportează.

Monitorizarea unui sistem este dificilă, însă aceasta poate să fie introdusă treptat însă putem începe cu agregare fișierelor de jurnalizare dar și notarea parametrilor în care rulează mașinile de lucru și raportarea acestuia într-un loc central.

3.8 Securizare

Indiferent de platforma sau tipul dezvoltării pe care îl facem, front-end, back-end sau IoT, securitatea va fi una din problemele care vor fi ridicate atunci când dezvoltăm produsul. Însă aprofundarea acestei părți nu este ușoară întrucât necesită o mentalitate specială în care luăm în considerare un adversar. În general, în timpul dezvoltării construim elemente și le construim într-un mod în care să funcționeze eficient, însă există persoane care vor să abuzeze de vulnebaribilățile ce pot apărea la orice nivel al aplicației.

Din acest motiv, în cadrul unei companii există persoane speciale care se ocupă de securitate. Aceștia cercetează și studiază posibilele vulnerabilități care apar și crează anumite investigații asupra produsele companiilor. În același sens, se pot folosi și unelte speciale în interiorul pipe-ului care au ca scop scanarea fișierelor și raportarea posibilelor vulnerabilități, chiar dacă nu este amănunțit sau o scanare în detaliu, aceasta ar oferi feedback rapid ce poate fi soluționat în același mod.

Prin stilul lor, microserviciile ne oferă posibilitatea de a îmbunătății securitatea sistemului întrucât fragmentarea duce la ascunderea informației către serviciile din exterior, astfel un atacator ar primi de fiecare dată un minim de informație, însă pe cealaltă parte, microserviciile sunt foarte complexe și necesită mult mai multe setări, pe lângă infrastructură și datele pe care acestea le păstrează, cresc numărul de rețele ce trebuie securizate, numărul de dependințe ce pot proveni de la terți (Eventbus, load-balancers, log tools) cresc și trebuie scanate regulat pentru vulnerabilități, astfel numărul de sisteme ce trebuie să fie luate în considerare atunci când construim metodele de protejare ale sistemului.

La baza securității pot sta anumite gânduri ce au ca scop restricționarea și controlarea

accesului. Anumite elemente le-am menționat până să ajungem aici, principiul ascunderii al informației se referea la limitarea numărului de date pe care le expunem, în cadrul securității ne putem gândii la limitarea privilegilor, astfel, ar trebui să oferim drepturi fiecărui microserviciu cu care comunicăm numai cât are nevoie pentru a funcționa normal, dacă el folosește doar câteva funcționalități atunci acesta ar trebui să aibă vizibilitate numai asupra lor, motivul fiind, dacă acesta este compromis atunci atacatorul nu poate să acceseze mai mult decât microserviciul avea nevoie. Atunci când construim o soluție de securitate pentru sistemul nostru, este indicat să conțină mai multe nivele și moduri ce oferă protecție. Există atacuri ce s-au produs iar în timpul lor nu a existat vreo metodă de alertare. Astfel, putem să facem distincția între modurile preventive de securizare, acestea au scopul de a preveni apariția unui atac, iar dacă se întâmplă să se limiteze daunele ce pot fi produse, printre acestea includ acțiuni precum securizarea variabilelor secrete, crearea unui sistem de autentificare și criptarea datelor. Putem spori detecția prin setarea aplicațiilor de firewall sau alerte atunci când se accesează anumite date. Pentru a mitiga atacul putem folosi diferite procedee însă cel mai important este comunicarea eficientă între echipe iar ulterior metode de a recupera sistemul. Implementarea automatizării este o metodă de a crește securitatea întrucât forțează gândirea metodelor de acces dar și eliminarea greșelilor umane iar adăugarea unor metode de analizare a securității în pipeline ce pot investiga erorile simple la nivel de cod.

Tot de la bazele securității provine crearea unui model de amenințare. Procesul creării acestuia include mai multe etape ce au ca scop cunoașterea sistemului și identificarea atacatorilor. Prima etapă se referă la identificarea elementelor de valoare din componența sistemului, modul în care acestea ar putea fi abuzate și identificarea posibililor atacatori ce ar putea apărea pentru a cauza probleme în interiorul sistemului. În urma acestuia, putem să ne gândim la modul în care o să protejăm datele și accesul la acestea. Întrucât nu putem preveni inevitabilul de fiecare dată următorul pas este creșterea metodelor de detecție, în general prin folosirea de unelte dedicate la nivel de rețea și infrastructură. Capacitatea de răspuns în fața unui atac nu se referă doar la metodele de oprire ale atacului dar și alte elemente precum comunicarea problemelor către clienți și parteneri. Metodele de recuperare se referă la revenirea la normal și aplicarea elementeleor descoperite în cadrul atacului pentru a preveni astfel de incidente.

Întrucât dezvoltarea pornește de la aplicație, de aici putem începe și cu crearea de practici bune pentru a crește securitatea sistemului. În general, o aplicație va avea cel puțin câteva date sensibile ce au de a face cu autentificarea, fie a microserviciilui în sine către baze de date sau alte microservicii sau alte utilizatorilor. Întrucât numărul de conexiuni poate fi ridicat, am putea încerca crearea unor date generale și distribuirea acestora însă ar încălca unul din principiile menționate anterior. Microserviciile sunt create pentru a oferi un serviciu către utilizatori, în general aceștia pot să nu fie experimentați și să ignore sfaturi precum folosirea parolelor unice, lungi și administrare de programe speciale,

iar în cazul de față se poate ajunge ca dauna provocată să nu se producă doar în cadrul seriviciului nostru ci prin refolosirea datelor chiar și către alte servicii pe care nu le administrăm. Pe lângă sfaturile simple precum securizarea parolele prin folosirea salt-ului și a algoritmilor siguri într-un timp ridicat (pentru a preveni atacurile de tip brute force) și folosirea de 2FA, putem extrage din articolul [5] al lui Troy Hunt în care a prelucrat recomandări de la NIST. Poate exista o limită la dimensiunea parolelor însă restricția ar trebui să fie mai mare de 64 de caractere suportând și caracterele UNICODE nerestricționate de reguli precum o anumită compoziție permițând folosirea de unelte de memorare a unor parole ce nu expiră ce sunt verificate pentru a preveni prezența acestora într-o bază de date de parolă existente și oferirea utilizatorului unei pagini în care își poate observa activitățile și dispozitivele. Pe lângă acestea, pot exista secrete, adică date senzitive care au ca scop funcționarea microserviciului într-un anumit mediu, acestea pot fi perechi de chei asimetrice publice/private dar și de API. Diferența dintre acestea și parolele sunt faptul că noi administrăm modul lor de viață, controlând crearea, distribuirea, stocarea, monitorizarea și schimbările. În acest context, în funcție de modul deployment-ului putem folosi caracteisticile acestora, Kubernetes prezintă o formă simplificată de secret management, acestea sunt memorate în memorie RAM și definite prin perechi chei-valoare iar în interiorul pod-ului sunt accesate sub forma de fișiere montate local, limitarea lor este că orice schimbare a secretului nu va avea efect decât atunci când pod-ul este restartat sau putem folosi platforme specializate precum Hashicorp Vault ce oferă o soluție complexă în acest domeniu. Implementând o metodă de rotație ne permite să generăm chei atunci când avem nevoie pentru o perioadă scurtă de timp, astfel dacă acesta este compromisă atunci putem limita pagubele ce pot fi produse, în același timp revocarea ne ajută în cazul conștientizării unei expuneri iar acestea ar trebui să ofere acces doar la resursele de care avem nevoie.

Securitatea aplicației nu se referă strict la ea ci și la modul cum interacționăm cu ea, adică prin infrastructură. Trebuie să fim la zi cu actualizările sistemului pe care aplicația rulează, dacă folosim infrastructură din cloud probabil acestea sunt făcute deja. Alte servicii pe care operatorii cloud le oferă sunt opțiunile de backup și encriptare a datelor la idle. Backup-urile ne permit să ne întoarcem la versiuni anterioare ale aplicației, astfel nu avem nevoie de backup-ul unei mașini propriu zise ci doar a modului de a ajunge în aceași stare. Codul și imaginile proceselor sunt deja track-uite prin VCS, folosind infrastructură ca și cod sau GitOps atunci avem și configurarea sistemului, rămând doar datele utilizatorilor ce în general se fac prin baze de date ce probabil vin cu versiunile proprii de backup în funcție de tehnologia aleasă. Pentru ca back-up-urile să fie eficiente, trebuie să avem încredere că acestea funcționează astfel putem să creăm un proces în care frecvent facem restaurare a backup-ului pentru a fi testat.

Microserviciile funcționează prin comunicarea la distanță prin diferite rețele, astfel se disting două cazuri, întrucât microserviciile sunt lansate într-un loc în care avem control

total și putem identifica orice neregularitate, putem avea încredere deplină că atunci când vine o cerere din interior, aceasta vine de la un serviciu de încredere. În partea opusă apare conceptul de zero încredere, în care toate serviciile care încearcă să ne acceseze sunt deja compromise, în acest caz trebuie să luăm precauții. Acesta este un mod de gândire specific securității, astfel cererile trebuie inspectate iar datele ce sunt comunicate trebuie să fie criptate. Atunci când construim un sistem, ar fi dicil ca acesta sa fie descris ca cel din ultimul caz, iar siguranța ar putea fi afectată dacă ar fi ca în cel din primul, astfel putem să le folosim pe amândouă în funcție de datele pe care le folosim.

Microserviciile vor face mereu schimb de date, fie ele către un singur alt microserviciu, fie către mai multe. Din acest motiv am vrea ca aceste date să nu fie interceptate de către un terț. Dacă comunicarea este făcută prin HTTP putem folosi TLS pentru a securiza traficul, iar dacă comunicarea se face printr-o aplicație de comunicare de evenimente atunci putem consulta documentația platformei și să găsim modalități de securizare. Atunci când dorim să securizăm o comunicare, sunt câteva puncte pe care le urmăm. Una din ele este stabilirea autenticității server-ului, astfel serviciul care face cererea vrea să știe dacă cel către care face cererea este chiar cel căruia vrea să îi facă cererea. Astfel de cazuri au făcut probleme pe Internet, astfel s-a dezvoltat HTTPS pe care îl putem folosi sub diferite forme. În același sens, serverul ar vrea să cunoască dacă clientul este chiar cel care încearcă să facă cererea. Astfel de cazuri nu prea există pe Internet-ul public întrucât serverele sunt accesibile, iar autentificarea clientului nu este o prioritare, mai ales când utilizatorii folosesc lucruri precum combinații de autentificare ce cuprind parole. În cazul microserviciilor putem folosi protocoale precum Mutual TLS pentru a stabili autenticitatea clientului. Implementarea acestuia ar fi fost dificilă în trecut fiind o problemă distribuirea certificatelor însă datorită uneltelor precum Hashicorp Vault acestea devin mai ușoare. După ce participanții la comunicare sunt autentificați aceștia își doresc ca mesajele lor să fie securizate. În acest sens ne interesează dacă mesajul poate să fie văzut de un terț, însă folosind HTTPS acesta va cripta datele folosind cheile publice iar cel care le decriptează va folosi cheia sa privată, în același mod ne interesează dacă datele au fost modificate, pentru asta putem folosi HMAC-ul prin calcularea un hash asupra datelor trimise pe care recipientul le poate calcula pentru și să vadă dacă acestea au fost modificate.

După ce securizăm comunicarea, a doua prioritate este securizarea stocării, pentru asta putem folosi o varietate de unelte, în general majoritate distribuitorilor de infrastructură cloud oferă modalități de encriptare iar majoritatea bazelor de date oferă același lucru. Însă această criptare poate deveni costisitoare, și din privința procesorului ce ar trebui constant să lucreze dar și a spațiului de stocare necesare, din acest motiv ar trebui să ne limităm datele salvate și să encriptăm orice dată primim și să decriptăm doar atunci când fără aceasta nu am putea să continuăm operațiunile.

O altă problemă a microserviciilor este modul de autentificare și autorizare. Într-o

aplicație monolitică, aceasta nu era o problemă la fel de mare întrucât o dată autentificat toate cererile erau rezolvate de aceași entitate, însă în cadrul unei arhitecturi decuplate acest lucru devine mai delicat. Prima necesitate de autentificare este cea a microserviciilor, pentru a valida componentele ce participă la comunicare, în acest sens putem folosi cheile pentru mutual TLS sau chei de API pentru a cripta cererea și server-ul ar valida că acel client chiar există. Însă dificultatea mai mare este autentificarea utilizatorilor. Întrucât aceștia folosesc mai multe microservicii, ar trebui să se autentifice către fiecare, iar acestea ar primi informația despre utilizator. Însă acest lucru este un inconvenient pe care vrem să îl evităm. Soluția ar fi crearea unui loc central în care utilizatorul se autentifică iar după cererea este redirecționată către cererea inițială căreia îi lipsea autorizația. Acest lucru este întâlnit la companiile ce oferă mai multe servicii precum Microsoft la care avem Outlook, OneDrive, Teams, acestea au o pagină comună de conectare, iar conectarea la una îți oferă posibilitatea de a te conecta la toate celelalte cu ușurință. Acest sistem ar funcția prin intermediul unui intermediar care ar căuta dacă utilizatorul este autentificat și dacă cererea pe care încearcă să o facă necesită autentificare și este fie lăsat să își termine cererea fie către pagina de autentificare. O astfel de abordare ar avea probleme întrucât doar prima cererea ar primi informații despre utilizator, iar următoarele ar trebui să aibă încredere că microserviciul cu care comunică nu este compromis. Pentru a rezolva această problemă am putea încerca să facem autentificarea în cadrul fiecărui microserviciu, însă ar trebui să reutilizăm cod și deși am putea folosi librării nu ar fi sustenabil. O metodă de autentificare poate fi folosira de JSON Web Tokens, aceștia pot fi semnați de către sistemul de autentificare central, pasați către utilizator care de fiecare dată când va face un request își va trimite token-ul, iar autentificarea de la un microserviciu la alt microserviciu s-ar putea face cu ușurință întrucât ar consta doar în trimiterea token-ului. În componența token-ului avem un header (ce conține informații despre algoritmul de semnare), datele și semnătura propriu zisă. Deși JWT-urile sunt ușor de folosit, acestea nu sunt perfecte. O primă problemă este setarea unui timp bun de expirare. Fiecare token are o dată de expirare setată de noi, după trecerea acestui timp, Token-ul ar trebui să fie considerat invalid, astfel pot apărea probleme în tranzacțiile de lungă durată, însă dacă timpul de expirare este mai ridicat atunci dacă este compromis se pot provoca mai multe daune întrucât nu există o metodă ușoară de a invalida un JWT. De asemenea, folosirea JWT-ului include administrarea informațiilor necesare pentru criptare, adică cheile și timpul de expirare ce poate fi greu de sincronizat între toate microserviciile care ar avea nevoie, însă unelte precum Vault ne poate ușura munca în acest context.

3.9 Evolutie

3.9.1 Stabilitate

Într-un proces clasic de dezvoltare a software-ului în stilul Waterfall, după ce aplicația este construită, testată și trece de audit-ul echipei de securitate aceasta va fi lansată și se va ajunge în starea de mentenanță. Însă modern, dezvoltarea aplicației nu se termină aici ci se va dori extinderea, consumul aplicației de către diferite interfețe, scăderea numărului de erori și tratarea acestora dar și eficientizarea sistemului.

Fiind implicată conexiunea prin Internet dar și globalizarea, utilizatorii se așteaptă ca serviciile pe care le oferim să fie disponibile 24/7. O arhitectură distribuită ne permite să creștem această disponibilitate întrucât poate să se întâmplă ca un serviciu pe care îl creăm să fie căzut, celelalte fie nu ar fi afectatate sau ar rula într-o variantă mai limitată.

Pentru a crește reziliența sistemului putem definii anumite arii în care aceasta acționează. Când aplicația poate să trateze erori pe care le anticipăm, precum tratarea erorilor sau reîncercarea cererilor căzute. Din această cauză putem să construim soluții ce devin mai complicate, de exemplu adoptarea Kubernetes pentru a beneficia de facilitățile acestuia precum menținerea constantă a stării, introducem complexitate și un efort ce poate să creeze mai multe probleme. Capacitatea de a absorbi atunci când se produce o anomalitate sporește stabilitatea sistemului, concentrarea pe îmbunătățirea metodelor de prevenire al cazurilor de acest tip ne poate lăsă vulenaribili, astfel putem crea metode de recuperare sau metode alternative de funcționare atunci când apar probleme în sistem. Creșterea rezilienței ar trebui să fie făcută constant și să ne adaptăm la evoluția sistemului.

Deși putem încerca să reducem șansele de apariție ale unei probleme, acestea nu pot fi evitate, iar acestea pot apărea la orice nivel al aplicației inclusiv în afara ei la partea de infrastructură. Însă tratarea tuturor evenimentelor care ar apărea ar fi obositor și nu ar da randament, întrucât pot fi microservicii în sistemul nostru la care ne putem aștepta să cadă și să nu creeze perturbări la fel de mari. Astfel putem să testăm monitorizăm sistemul și să calculăm timpi de răspuns, dacă aceștia nu sunt în parametrii pe care îi dorim putem folosi sisteme de scalare, sau a timpului în care sistemul este căzut și stabilirea dacă aceasta este acceptabilă la fel și cât de multe date putem să pierdem.

Pentru a continua funcționalitatea sistemului în procent cât mai mare, uneori e nevoie să facem compromisuri, astfel dacă cunoaștem că un microserviciu de care suntem dependenți este căzut, putem să îl ascundem, astfel utilizatorul are o experiență mai proastă, însă sistemul încă este disponibil. Metode de menținere a stabilității sistemului pentru astfel de situații pot fi, introducerea de time-out-uri în cereri, astfel clienții nu așteaptă mai mult decât trebuie. Durata după care se introduce un timeout poate fi calculată după rularea testelor și adjustată conform telemetriei. Însă unele erori pot să fie doar temporare, iar o reîncercare probabil ar fixa problema, în acest caz putem introduce un sistem

de reîncercare în funcție de eroare pe care o primim. Aceste două metode de tratare a eșecului totuși sunt costisitoare din punctul de vedere al timpului. Întrucât conexiunile încete și un număr ridicat de reîncercări ar face ca clientul să trebuiască să aștepte foarte mult, chiar și pentru o eroare. În acest caz putem crea un fel de siguranță ce are scopul ca atunci când avem funcționalitatea normală aceasta să continue, iar atunci când avem probleme în sistem, clienții care ar încerca să acceseze resursele afectate ar primi un răspuns cât mai rapid, astfel nu ar cauza frustrări pentru aceștia. Erorile se pot propaga, astfel crearea unei izolări între microservicii ar cauza ca acestea își afectează starea reciproc. În acest sens putem crea separări logice în care comunicare între microservicii se face printr-un intermediar care în absența unuia dintre participanți va salva lucrurile ce trebuie să le trimită și le va trimite atunci când va fi disponibil, sau izolare fizică, astfel să ne asigurăm că bazele de date ale microserviciilor dar și aplicațiile în sine sunt separate între ele, astfel dacă un server cade, nu ar afecta mai multe servicii în același timp. Deși acest lucru este mai costisitor putem să folosim asta și să coordonăm metodele de eșec, astfel dacă știm că funcționalitate unui serviciu este puternic afectată de un altul, atunci acestea pot fi grupate și dacă inevitabilul se întâmplă cele două vor cădea împreună. În același stil putem să creștem numărul de apariții ale unui microserviciu, astfel dacă unul din ele este căzut, avem alte clone ce îi pot lua locul. Platforme precum Kubernetes ne permite să facem acest lucru cu ușurință. În același stil, putem să trimitem mai multe cereri de mai multe ori, acest lucru funcționează dacă cererile create și modul lor de prelucrare este idempotent, adică de oricât de multe ori este aplicat, acesta va da mereu același rezultat oricât de multe rulări am face.

În construirea unui sistem bazat pe microservicii stabil am discutat despre metode de eficientizare și de îmbunătățire a rezilienței în care folosim mai multe baze de date sau multiple instanțe ale unui microserviciu. Există o teoremă ce a fost demonstrată și matematic ce spune că putem avea consistență, disponibilitate (avalability) și tolerare a separărilor (partition tolerance) însă dintre acestea putem să alegem în sistemul nostru doar două dintre ele, a treia neputând fi atinsă. Consistența reprezintă ca oricare din instanțele ale unui microserviciu, toate ar oferi același răspuns pentru aceași cerere. Disponibilitatea reprezintă capacitatea unui microserviciu de a primi un răspuns iar toleranța separării reprezintă capacitatea sistemului de a manevra incapacitatea serviciilor de a comunica între ele.

Renunțând la consistență ar însemna ca atunci când avem mai multe microservicii de același tip ce comunică cu o bază de dată, dacă nimerim la noduri diferite am primi răspuns diferit. Acest lucru se poate întâmpla atunci când baza de date este formată din mai multe baze de date, unele destinate doar citirii iar una doar pentru scriere. În acest context, sincronizarea dintre cele două tipuri de baze de date este pe moment întreruptă. În general putem sacrifica un pic din consistență în funcție de importanța datelor cu care lucrăm, însă ne așteptăm ca sincornizarea să se facă la un moment dat, cu cât

aceasta întârzie cu atât va fi mai dificil de executat, iar dacă refuzăm scrieri atunci când sincronizarea este lentă o să menținem consistența ridicată dar disponibilitatea scăzută.

Consistența este greu de atins întrucât necesită mulți pași pentru menținerea ei, iar interogarea datelor se face inițiind tranzacții pentru a ne asigura că datele pe care vrem să le accesăm sunt cu adevărat la fel, iar dacă aceste noduri nu sunt disponibile, atunci pentru a nu sacrifica consistența vom refuza cererea, astfel scăzând disponibilitatea. Crearea unui mod de partajare a datelor între mai multe noduri este dificilă, astfel ar fi bine să folosim soluții deja implementate, Consul oferă astfel de servicii pentru disitribuirea elementelor de configurare.

Negarea tolerării separări este imposibilă în cadrul unui sistem distribuit, întrucât aceasta ar însemna să renunțăm la microservicii și să revenim la o arhitectură monoliticăm în care nu există separare. Astfel singurele tipuri de sisteme pot fi CP sau AP, adică cele care renunță la disponibilitate, respectiv la consistență, însă în lumea reală acesta nu este neapărat aplicabil. Putem în cadrul unui singur sistem să avem mai multe tipuri de alegeri, în funcție de datele și tehnologia cu care lucrăm.

Testarea stabilității unui sistem distribuit este destul de complicat și s-ar face la mai multe nivele. Un termen popular, original provenit de la Netflix, este „chaos engineering" (ingineria haousului) acesta are ca rol testarea rezilienței sistemului. În general, poate fi considerat ca doar rularea unor unor unelte și interpretarea acestora, ceea ce nu e greșit, însă aceasta are ca scop îmbunătățirea încrederii atunci când apar probleme ce nu pot fi prevăzute, ceea ce se poate întâmpla în orice moment. Netflix se bazează într-o proporție foarte mare de infrastructura ce provine de la AWS, însă la un moment dat aceștia au fost forțați să efectueze lucrări de mentenanță asupra unei întregi zone de serviciu, astfel se putea crea o gaură mare de down-time pentru clienții Netflix (dar și pentru ceilalți ce se bazează pe AWS). Ei au creat un serviciu numit Chaos Monkey ce are ca scop testarea sistemului la apariția unor probleme neprevăzute în cadrul infrastructurii. În cadrul serviciului putem întâlnii mai multe stagii precum, Chaos Gorilla ce are ca scop căderea unei zone de disponibilitate, adică a unei regiuni mai mici (Europe Central), Chaos Kong, simulează căderea unei regiuni întregi, de exemplu Europa, Latency Monkey are ca scop creșterea latenței în cererile executate de clienți pentru a simula încetiniri în trafic, Conformity Monkey are ca scop închiderea instanțelor ce nu sunt în conformitate cu practicile cerute de sistem, Doctor Monkey monitorizează starea fiecărei instanțe și oprește sistemele ce nu sunt tratate în timp util, Janitor Monkey observă infrastructura ce nu este utilizată și o oprește după un anumit timp, Security Monkey are ca scop închiderea serverelor ce nu au toate practicile de securitate necesare.

3.9.2 Scalare

În timpul prezentării microserviciilor, un avantaj al acestora a fost mereu flexibilitatea, în același stil am menționat posibilitatea îmbunătățirii sistemului prin replicarea unor anumite funcții ale acestuia, fie pentru administrarea a mai mult trafic, fie pentru îmbunătățirea disponibilității și eliminarea timpilor în care serviciul este inaccesibil. Scalarea poate fi făcută în mai multe moduri, în funcție de necesitățile și de problemele pe care le-am rezolva.

Scalarea verticală se referă la îmbunătățirea sistemului de calcul prin aducerea mai multor resurse de calcul, fie prin îmbunătățirea sistemelor de I/O, adică folosirea unor SSD-uri rapide față de HDD-uri clasice, fie de calcul prin îmbunătățirea procesorului și a memoriei RAM. În cazurile clasice, când noi suntem cei care hostăm aplicația, aceasta poate fi mai de lungă durată întrucât sunt mai multe echipe implicate, însă având în vedere sistemele cloud, scalarea verticală se face foarte rapid și nu necesită configurări adiționale. Însă acest tip de scalare nu este o soluție ce va funcționa la nesfârșit, se observă că legea lui Moore nu se mai aplică, astfel îmbunătățirile tehnologice nu sunt la fel de drastice în fiecare an. Nu ar trebui să ne bazăm că putem schimba mașina de calcul, mai ales că folosirea tehnologiilor foarte puternice ar deveni costisitor. În acest scop, putem folosi algoritmi mai eficienți sau alte tipuri de scalare.

Scalarea orizontală se referă la multiplicarea instanțelor ale unui serviciu, în acest stil putem crește numărul de procesări ce se pot face în același timp. Implementarea cea mai simplă se poate face folosind un balansator de încărcătură (load balancer) ce va dirija traficul asigurând că traficul este egal pe fiecare dintre microservicii. Acest lucru este posibil atunci când putem să distribuim o acțiune către mai multe locuri, însă putem avea situația în care distribuim sarcinile din interiorul unei cereri, astfel o altă implementare ar fi separarea unor cerințe din interiorul unei cereri către mai multe microservicii ce au unic, de procesare, iar scalarea s-ar face asupra acestora. Am mai menționat și scalarea bazei de date prin folosirea de replici de citire ce sunt sincornizate cu baza de date destinată scrierilor. Limitările acestui tip de scalare se rezumă la dificultățile de implementare, astfel putem întâmpina mai multe probleme de rețelistică sau de sincronizare a datelor sau chiar recrearea unui nou mod de desfășurare destinat acestui stil distribuit de lucru.

Partiționarea datelor se referă la distribuirea cererilor către un anumit set de microservicii în funcție de natura cererii. Astfel, putem avea microservicii de același fel, cu funcționalitate identică însă care se ocupă de date diferite. Aceasta poate să fie o îmbunătățire întrucât am avea sisteme de procesare dedicate unor anumite cereri, astfel dacă am întâlnii mai mult trafic într-un anumit loc, am putea planifica o partiționare din nou, iar dacă nu avem atunci să regrupăm anumite date. Implementarea se poate face în funcție de tehnologie, astfel, dacă baza de date pe care noi o folosim oferă asistență în acest loc (de exemplu Cassandra) atunci putem să o facem la nivelul acesteia, sau să

implementăm servicii de proxy ce redirecționează traficul către microserviciile ce ne oferă funcționalitatea de care avem nevoie. Acest tip de scalare este eficient când avem suntem constrânși din punct de vedere al scrierilor, întrucât acestea nu pot fi eficient distribuite în alt mod. Limitările pot fi faptul că nu se aduce o îmbunătățire la fel de mare și în funcție de modul în care ne facem grupările ne poate duce să avem un grup de persoane afectat, astfel putem grupa acest tip de scalare cu cele precedente pentru efectivitate mai bună. În același stil, stabilirea modului de grupare și dimensiunile partițiilor pot fi dificil de stabilit.

Majoritatea formelor de scalare încercau să nu modifice structura aplicației prea mult, însă după ce epuizăm aceste forme, putem încerca extinderea aplicației prin decuplarea funcționalității în alte microsisteme. Beneficiul acestui tip ar fi posibila eficientizare a procesului întrucât s-ar putea parta încărcătura însă ar crește complexitatea sistemului și poate cauza degradări ale perfomanței întrucât s-ar introduce o cerere pe rețea nouă, astfel trebuie să ne asigurăm că implementarea acestui tip de funcționalitate ar avea sens.

Sunt multe cazuri în care soluțiile pe care le creăm sunt mult mai complicate decât ceea ce ne cere sistemul. Toate implementările de scalare sunt incluse în această frază, astfel nu ar trebui să adăugam complixate fără să fie dovedit că am avea nevoie. În general, există multe caracteristici introduse de dezvoltatori însă care nu sunt folosite mai niciodată de către utilizator, însă atunci când aceste funcționalități au șansa să strice aplicația ar trebui să fie evitate și aplicate doar la nevoie. Iar în acest context, întrucât microserviciile ne oferă o flexibilitate ridicată putem să aplicăm la un anumit nivel toate tipurile de scalare în loc să alegem doar una din ele.

Există și conceptul de auto scalare, în general ne putem gândii ca sistemul să fie mai puternic accesat într-un anumit interval, astfel putem să creștem instasnțele de microservicii în această perioadă. Ofertanții de infrastructură cloud ne oferă diferite metode de autoscalare deja în platformă și sub diferite forme, de exemplu atunci când ajungem la un anumită încărcătură pe un anumit nod pe o anumită perioadă putem să mai creăm o instanță automat. Astfel autoscalarea poate fi reactivă sau predictivă, ambele aducând îmbunătățiri însă trebuie să analizăm datele, ce poate fi făcut cu algoritmi complecși precum cei de Inteligența Artificială / ML.

3.9.3 Caching

Caching este o metodă de a salva informația cerută la un moment dat de către un utilizator și prevenirea recalculării modului de a ajunge la date și doar servirea aceluiași răspuns. Atunci când putem să oferim același răspund întrucât l-am salvat putem spune că avem un „cache hit” și când nu, un „cache miss”

Există multe motive pentru care am adopta această tactică, întrucât sistemul de cachare are ca rost prevenirea recalculărilor inutile, putem îmbunătății performanța sistemului

întrucât unele dintre aceste calcule pot include cereri pe Internet ce adaugă latență. În același stil putem eficientiza traficul dacă avem cachate anumite răspunsuri pe care doar le livrăm de mai multe ori, astfel întrucât nu aduc complexitate pot fi scalate cu ușurință. Deși ar scădea consistența, existența unui sistem de cache ne-ar putea permite să rulăm o anumită perioadă de timp fără ca sursa obținerii datelor să fie disponibilă, în acest stil putem ascunde problemele de puțină durată întrucât avem deja o sursă de date introdusă.

Această tehnică poate fi aplicată la mai multe nivele, la nivelul clientului, putem salva în interfață datele, de exemplu pe telefoane mobile putem face o dată o cerere către baza de date, salvăm rezultatul și acesta va fi disponibil chiar și atunci când nu avem Internet, acest lucru este vizibil în rețelele de socializare care atunci când deschidem aplicația mereu au anumite postări preîncărcate, această interacțiune ne permite să salvăm baterie și să nu creăm prea multe cereri, în același stil conținutul pe care îl afișăm utilizatorului este mai greu de controlat și poate diferi de la unul la altul. Cacharea la nivel de server ne permite să evităm anumite operațiunii de lungă durată pe care ne așteptăm să nu se întâmple foarte des, astfel putem să trimitem un răspuns salvat. Această operațiune ne permite să avem un control mai mare asupra datelor pe care le salvăm, însă clientul tot ar trebui să consume resurse făcând cererea. În acest stil putem să salvăm informații ale unei cereri specifice, iar dacă aceasta este repetată atunci se va trimite răspunsul vechi.

Atunci când vorbim despre salvarea unei informații prin caching, trebuie să luam în vedere modul în care ștergem această salvare. Dacă nu am luat asta în considerare atunci clientul ar vedea informații destul de vechi ce nu ar fi actualizate. Cel mai simplu mod de implementare este specificarea unui timp la care informația din cache expiră. Intervalul poate fi dificil de ales iar în același timp pot exista cazuri în care datele se actualizează fix după ce clientul le înregistrează ca find cele mai recente iar în cazul în care TTL (Time To Live) este setat mare ar putea cauza probleme. În același fel putem eticheta fiecare cerere cu un ID, iar dacă ID-ul nu se schimbă atunci putem trimite un răspuns mult mai mic. Dacă avem o resursă ce este disponibilă la un anumit URI, dacă am face cerere pentru aceași resursă putem trimite un identificator al datelor pe care le avem iar server-ul ne poate trimite fie date noi, fie faptul că datele nu s-au schimbat. O altă metodă de invalidare a cache-ului poate fi folosirea de notificări, astfel server-ul va trimite fiecărui client care a cache-uit un anumit mesaj, fie cu date noi fie doar faptul că au fost invalidate. Aceasta implementare ar aduce avantajul că nu ar exista date vechi pentru clienți însă implementarea unui astfel de mecanism este complexă și poate consumă destul de multe resurse. De asemenea, cache-ul poate fi modificat fix după ce resursa se schimbă, o astfel de abordare ar avea sens dacă am face cache-ing pe un server, sau să îl folosim ca un buffer în care scriem mai întâi în cache și după acesta se va sincroniza cu serverul.

Caching-ul este o metodă de optimizare a traficului, astfel ar trebui să fie folosit doar acolo unde am considera că ar crea o îmbunătățire a sistemului, la fel ca la scalare acesta poate fi introdus treptat, deși poate face parte încă din arhitectura originală însă putem

să investim resurse în ceva în care nu ar fi necesar și am aduce complexitate într-un loc în care poate nu e nevoie.

3.10 Organizare

3.10.1 Consum

Atunci când construim un sistem, ne așteptăm să îl dăm în folosință pentru cineva, fie clienți sau ca asistent al unui sistem existent. Însă în general acești clienți pot să nu consume chiar sistemul direct întrucât acesta poate fi complex, ci putem să îi oferim printr-o interfață construită tot de noi (în cazul aplicațiilor, putem considera front-end-ul o astfel de interfață), astfel putem avea ca responsabilitate și crearea unui astfel de lucru.

În modul tradițional de lucru, am avea dezvoltatori grupați în funcție de rolul și platforma pe care lucrează, astfel toți cei responsabili de o anumită componentă mai mare (de exemplu domeniu sau prezentare) ar fi în același loc. Acest tip de organizare ne permite să avem o experiență identică la același nivel, însă comunicările între nivele pot fi afectate, astfel s-ar putea introduce probleme de sincornizare.

O metodă mai eficientă de organizare este gruparea dezvoltatorilor în funcție de locul în care aceștia lucrează, astfel toți cei responsabili la toate nivelele de o anumită componentă a sistemului nostru vor lucra împreună. Acest mod ne permite un control mult mai bun, având metode de schimbări la orice nivel, astfel atunci când avem o inițiativă, aceasta e mai ușor de implementat și nu depindem de echipe exterioare pentru care trebuie să așteptăm o implementare.

Construirea unei interfețe grafice poate devenii dificilă. În cadrul unei companii ce oferă mai multe produse putem fi tentați să creăm o echipă dedicată pentru aplicațiile pe care utilizatorii le-ar folosi, astfel aceasta ar fi consistentă tuturor. Această abordare poate crea probleme atunci când vrem să lansăm caracteristici noi, de aceea o idee mai bună ar fi crearea unei echipe ce va lucra în cadrul mai multor echipe, având posibilitatea să schimbe proiectele.

Organizarea construirii interfețelor grafice poate fi făcută în mai multe moduri. Cea mai simplă variantă presupune existența unei aplicații monolitice, în cadrul acesteia consumăm API-urile propriu zise fără a face vreo optimizare modului cum sunt făcute. Dezavantajul unei astfel de abordări este decuplarea între backend și frontend dezvoltarea caracteristiciilor acestora nu poate fi la fel de bine coordonată. În același stil, crearea multor cereri poate să cauzeze probleme legate de memorie, astfel acestea nu sunt efectuate eficient.

O versiune mai avansată bazată pe framework-urile noi de dezvoltare este microfront-end-uri, în cazul acesteia echipele furnizează un UI ce poate fi lansat independent de UI-urile existente. O astfel de abordare le permite echipelor să își creeze UI-ul independent de

celelalte și acestea rămânând să se lanseze aproape independent, dar uneori tot combinat.

Această abordare poate fi implementată în mai multe moduri. Una dintre acestea este decompoziția site-ului pe pagini, fiecare echipă fiind responsabilă de un anumit endpoint de pagini pe site-ul mare. Aceasta le permite să fie independente întrucât livrarea paginilor este independentă de paginile existente pe care server-ul trebuie să le arate.

O abordare mai dificlă de implementat, este separarea unei pagini în funcție de secțiile pe care vrem ca acestea să le cuprindă, astfel fiecare echipă ar fi responsabilă de o componentă ce apare pe o pagină la un anumit moment. Implementarea unei astfel de soluții ar fi ca fiecare echipă să își construiască elementele ce vor să fie disponibile pe website și crearea unei metode de agregare ale tuturor elementelor create. Din acest motiv, deși elementele sunt separate acestea tot trebuie să fie integrate, iar crearea unui comportament împărțit în cadrul mai multor elemente poate fi dificilă.

Unul din dezavantajele a unui front-end monolitic era numărul mare de cereri ce ar fi trebuit să fie făcute către mai multe microservicii pentru o singură pagină. Pentru a rezolva problema aceasta putem să folosim un intermediar ce are ca rol această agregare a datelor, astfel de pe dispozitiv facem un singur call însă în spate sunt făcute mai multe. Dificultățile implementarii ar consta în coordonarea echipelor, întrucât acest intermediar nu ar avea o echipă directă ci toți ar contribui la el. În același stil ar exista dificultăți de tratare al numărului de informații pe care tipuri de dispozitive le-ar folosi dar și de autentificare, întrucât totul devine agregat de un intermediar ce ar trebui să aibă acces la toate resursele pe care le oferim.

Oferirea de asistență pentru diferitele platforme ne-a dus la crearea conceptului de Backend for Frontend, astfel pentru fiecare tip de dispozitiv ce ar avea nevoie să ne acceseze sistemul, am crea un backend specializat ce s-ar ocupa doar de o anumită experiență. Această abordare poate avea sens dacă se fac multe apeluri peste rețea și dacă echipele de front end sunt decuplate de cele de backend. Implementarea poate fi făcută de exemplu folosind GraphQL.

3.10.2 Structura

Capitolul 4

Orchestrare

4.1 Istoric deployment

4.2 Tipuri de deployment

4.2.1 Baremetal

4.2.2 Mașină virtuală

4.2.3 Container

4.2.4 Function as a Service

4.2.5 Platform as a Service

4.2.6 Container as a Service

4.3 Docker

4.4 Docker Swarm

4.5 Kubernetes

4.6 Red Hat OpenShift

4.7 HashiCorp Nomad

4.8 Infrastructure as a Service

Capitolul 5

Îmbunătățirea metodelor de dezvoltare software

5.1 Schimbari in livrarea produselor

5.2 DevOps și aspecte ale comunitații

5.3 Ce înseamnă să aplici DevOps

5.4 Unelte

Capitolul 6

Aplicație

Capitolul 7

Concluzii

Bibliografie

[1] Bahrim Dragoș, „Temă „Principii SOLID și CI/CD în Microservicii" pentru cursul Metode de dezvoltare software", în (Iunie, 2022).

[2] Albert Endres și H. Dieter Rombach, *A Handbook of Software and Systems Engineering: Empirical observations, laws, and theories*, Pearson/Addison Wesley, 2003.

[3] Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley Professional, August 20, 2003.

[4] Hector Garcia-Molina și Kenneth Salem, „Sagas", în *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*, ed. de Umeshwar Dayal și Irving L. Traiger, ACM Press, 1987, pp. 249–259, DOI: 10.1145/38713.38742, URL: https://doi.org/10.1145/38713.38742.

[5] Troy Hunt, *Passwords evolved: Authentication guidance for the modern era*, Aug. 2017, URL: https://www.troyhunt.com/passwords-evolved-authentication-guidance-for-the-modern-era/.

[6] David Lorge Parnas, „Information Distribution Aspects of Design Methodology", în *Information Processing, Proceedings of IFIP Congress 1971, Volume 1 - Foundations and Systems, Ljubljana, Yugoslavia, August 23-28, 1971*, ed. de Charles V. Freiman, John E. Griffith și Jack L. Rosenfeld, North-Holland, 1971.