



UNIVERSITATEA DIN BUCUREȘTI



FACULTATEA  
DE  
MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

LUCRARE DE DISERTAȚIE  
EXECUȚIA WORKFLOW-URILOR DURABILE ÎN  
CLOUD

Absolvent

Moldovan George-Alexandru

Coordonator științific

Lect. dr. Letiția Marin

București, septembrie 2022

## **Rezumat**

Considerând trend-ul ascendent al arhitecturii orientate spre microservicii și migrarea de la vechiul mod de dezvoltare monolit, această lucrare de disertație propune analiza soluțiilor prezente în piața în ceea ce privește managementul unei arhitecturi pe microservicii în cloud, precum și analiza unor contribuții personale aduse unui framework open-source de gestionare a workflow-urilor în cloud. Sistemele informatice preiau din complexitatea operațiilor de zi cu zi, astfel că arborii de decizie trebuie interpretați și gestionați în mod corect pentru a realiza un sistem care să îndeplinească cerințele de piață curente. Pentru o lungă perioadă de timp, problema gestionării tranzacțiilor distribuite și management-ul workflow-urilor a fost inexistentă, deoarece într-un sistem monolit, caracteristicile tranzacționale ale bazelor de date ce susțineau astfel de sisteme erau îndeajuns pentru a avea toate garanțiile necesare astfel încât sistemul să fie mereu lăsat într-o stare consistentă. În cazul microserviciilor și arhitecturilor distribuite în general, deși vin cu o serie lungă de avantaje, poate cel mai greu de gestionat lucru este management-ul stării unei acțiuni, atunci când aceasta se întinde pe mai multe microservicii, și pe o durată lungă de timp.

## **Abstract**

Considering the upward trend of microservices-oriented architecture and the migration from the old monolithic development mode, this dissertation aims to analyze the solutions present in the market in terms of managing a microservices architecture in the cloud, and the analysis of personal contributions to an open-source framework for managing cloud workflows. Information systems take over the complexity of day-to-day operations, so decision trees must be interpreted and managed correctly to achieve a system that meets current market requirements. For a long time, the problem of distributed transaction management and workflow management was non-existent, because in a monolithic system, the transactional characteristics of the databases that supported such systems were sufficient to have all the necessary guarantees so that the system should always be left in a consistent state. Distributed microservices and architectures in general, although they come with a long list of advantages, perhaps the most difficult thing to manage is the management of the state of an action, when it extends over several microservices, and over a long period of time.

# Cuprins

<b>1</b>	<b>Introducere</b>	<b>4</b>
1.1	Motivație . . . . .	4
1.2	Context . . . . .	5
1.3	Alte analize ale problemelor curente a arhitecturii Serverless . . . . .	6
1.4	Conținutul lucrării . . . . .	7
<b>2</b>	<b>Analiza Tehnologiei Durable Task Framework</b>	<b>8</b>
2.1	Analiză generală . . . . .	8
2.2	Folosirea evenimentelor în DTF . . . . .	9
2.3	DTF în Durable Functions . . . . .	12
2.4	Dificultăți în gestionarea workflow-urilor de lungă durată - Versionarea Orchestrărilor . . . . .	13
<b>3</b>	<b>Workflow-uri de lungă durată - Analiză paralelă a arhitecturii</b>	<b>15</b>
3.1	Descrierea problemei . . . . .	15
3.2	Implementarea arhitecturii în mod clasic . . . . .	16
3.2.1	Implementarea naivă . . . . .	16
3.2.2	Execuție durabilă . . . . .	17
3.2.3	Garanții de execuție. Cel puțin o dată versus exact o dată . . . . .	18
3.2.4	Interacțiunea din exterior cu workflow-ul. Operații de lungă durată . . . . .	19
3.2.5	Cronometre durabile . . . . .	20
3.2.6	Optimizarea numărului de request-uri . . . . .	20
3.2.7	Considerente de deployment și scalabilitate . . . . .	21
3.3	Implementarea arhitecturii folosind Durable Functions . . . . .	22
3.3.1	Noua Arhitectură . . . . .	24
3.3.2	Cronometre Durabile în Durable Functions . . . . .	26
3.3.3	Considerente de deployment și scalabilitate . . . . .	27
3.3.4	Arhitectura non-serverless pentru execuția durabilă a workflow-urilor . . . . .	28

<b>4 Rezultate și concluzii</b>	<b>31</b>
Bibliografie . . . . .	32

# Capitolul 1

## Introducere

### 1.1 Motivație

Unul din principiile de bază ale programării este reutilizarea. Motivația din spatele acestei lucrări o reprezintă dorința de a contribui la un framework care rezolvă o problemă generică, cu care se confruntă toți dezvoltatorii care trebuie să gestioneze tranzacții distribuite. Analiza diferitelor metode pentru rezolvarea problemei de gestiune a workflow-urilor în cloud, în special într-o arhitectură ce se bazează pe funcții în cloud, a fost o prioritate pentru mine în ultimii ani.

Serverless, sau Functions-as-a-Service (FaaS), este o paradigmă din ce în ce mai populară pentru dezvoltarea de aplicații, deoarece oferă scalare infinită implicită și facturare bazată pe consum. Cu toate acestea, garanțiile slabe de execuție și suportul nativ pentru stocare a stării a FaaS creează provocări serioase atunci când se dezvoltă aplicații care necesită stare persistentă, garanții de execuție sau sincronizare. Acest lucru a motivat o nouă generație de soluții serverless care oferă abstractizări ce stochează starea aplicației. Un exemplu în acest sens, noua soluție Azure Durable Functions (DF), o extindere peste deja existentă Azure Functions. Modelul îmbunătățește FaaS cu actori, fluxuri de lucru și secțiuni critice.

În acest context în care dezvoltatorii încearcă să dezvolte aplicații ce gestionează workflow-uri de lungă durată, cu toții rezolvă aceeași problemă și anume lipsa separării între nivelul de execuție și nivelul de stocare a soluțiilor existente FaaS. Cu toții rezolvă o problemă generică, de salvare a stării în anumite puncte ale execuției, pentru a putea relua workflow-ul în eventualitatea în care agentul pe care rulează aplicația pică înainte de finalizarea workflow-ului. Acest lucru era foarte greu de realizat în arhitecturi serverless deoarece toate soluțiile existente până la apariția Azure Durable Functions, suportau doar execuții stateless în mod standard. Deci până la apariția soluțiilor ce oferă garanții de execuție puternice în lumea Serverless, această tehnologie nu era o opțiune populară pentru dezvoltarea aplicațiilor ce aveau nevoie de gestionare a stării și era limitată la execuția unor părți mici ale aplicațiilor pentru care stările intermediare nu erau importante.

## 1.2 Context

Serverless diferă de conceptele tradiționale de cloud computing în sensul că infrastructura și platformele în care serviciile rulează sunt ascunse clienților. În această abordare, clienții sunt preocupați doar de funcționalitatea dorită a aplicației lor, iar restul este delegat furnizorului de servicii.

Scopul serviciilor Serverless este triplu :

- Scutește dezvoltatorii de servicii cloud de la interacțiunea cu infrastructura sau diferite platforme;
- Convertește modelul de facturare din cel clasic la cel bazat pe consum;
- Permite scalarea automată a serviciului în funcție de cererea clienților.

Ca rezultat, într-o aplicație cu adevărat Serverless, infrastructura de execuție este ascunsă clientului, iar clientul plătește doar pentru resursele pe care le utilizează efectiv. Serviciul este conceput astfel încât să poată gestiona rapid creșterile de consum prin scalare automată. Entitățile de bază în calculul fără server sunt funcții. Clientul își înregistrează funcțiile în furnizorul de servicii. Apoi, acele funcții pot fi invocate fie de un eveniment, fie direct prin apelarea acestora la cererea utilizatorilor. Rezultatele execuției sunt trimise înapoi clientului. Invocarea funcțiilor este delegată unuia dintre nodurile de calcul disponibile în interiorul furnizorului de servicii.

Deși conceptul de Serverless este relativ nou, acesta și-a deschis drumul în multe aplicații din lumea reală, de la instrumente de colaborare online la sisteme integrate (IoT), având o creștere în adopție foarte rapidă, lucru vizibil și în figura 1.1. Această creștere este datorată în mare parte ușurinței procesului de dezvoltare a aplicațiilor Serverless și beneficiilor pe care le aduc din punct de vedere al scalării în mod automat, și al gestionării complete a infrastructurii ce stă la baza aplicațiilor.

Unul din lucrurile care poate duce adopția tehnologiei serverless la un cu totul alt nivel, este tocmai capacitatea de a putea dezvolta aplicații întregi, ce se pot baza pe starea sistemului în cadrul execuției, precum și capacitatea de a gestiona long running workflows, o nevoie care este prezentă în mai toate aplicațiile curente.

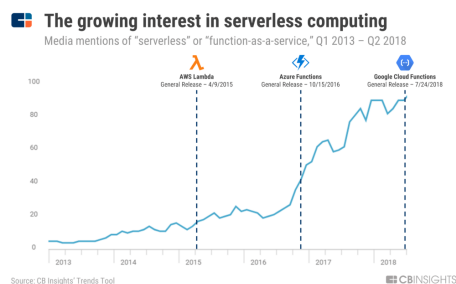


Figura 1.1: Statistică ce evidențiază importanța domeniului cloud computing în ultimii ani

Sursă: <https://www.cbinsights.com>

## 1.3 Alte analize ale problemelor curente a arhitecturii Serverless

Există mai multe provocări cu care se confruntă în prezent serviciile Serverless. Există unele sondaje și recenzii ale literaturii care discută aceste provocări [1, 2, 3, 4, 5].

*Baldini et al.* [1] enumeră o serie de probleme cu care se confruntă arhitectura serverless, printre care costul, care reprezintă un avantaj pentru această arhitectură doar dacă se execută metode care nu sunt bazate pe prelucrare input-output. Altfel, este mai eficient din punct de vedere al costurilor folosirea soluțiilor clasice în cloud cum ar fi mașini virtuale rezervate sau containere. O altă problemă semnalată este cea a cold-start-ului care poate face o aplicație bazată pe funcții serverless să pară înceată dacă pentru fiecare apel este necesar un cold start, din cauză că traficul nu este indeajuns de mult pentru a împiedica funcția să scaleze la 0.

*Rajkumar et al.* [2] discută despre dificultățile dezvoltării unei arhitecturi serverless din punct de vedere al limitărilor cu care vine această nouă tehnologie și anume limitările de timp al execuției, de memorie a agentului și de management al stării. Acesta crede că este nevoie de o schimbare a mentalității de dezvoltare a aplicațiilor pentru a beneficia la adevaratul potențial al tehnologiilor Serverless, iar momentul în care aplicațiile enterprise complete vor fi migrate sau dezvoltate complet pe o arhitectură Serverless este încă departe. Autorul vede această tehnologie ca pe o unealtă ajutătoare în dezvoltarea aplicațiilor, dar pe viitor poate ajunge să fie nucleul aplicațiilor.

*Castro et al.* [3] ridică problema dezvoltării de aplicații stateful folosind tehnologii Serverless în viitor, o temă ce în acea perioadă era doar o idee, dar după cum urmează să fie prezentat în această lucrare, acum este o realitate. Alte probleme menționate ar fi usurința cu care poate fi portată o aplicație legacy către o arhitectură Serverless, deoarece nu este de dorit să se piardă toate acele ore valoroase care au fost deja investite în aplicațiile existente.

*Hassan et al.* [4] discută despre problema limitării la un singur provider atunci când vine vorba de arhitecturi serverless, deoarece codul pentru un anumit provider de exemplu AWS Lambda, nu e portabil către alt provider, de exemplu Microsoft Azure Functions.

*Jonas et al.* [5] prezintă ineficiențele arhitecturii serverless atunci când vine vorba de procesarea operațiilor care în mod normal ar beneficia de pe urma unui sistem cu mai multe nuclee și implicațiile pe care aceasta limitare (2 nuclee per funcție) o are atunci când vine vorba de paralelizarea acțiunilor pentru a îmbunătăți viteza. Impactul major în acest caz este creșterea semnificativă a datelor transmise pe rețea pentru a obține același grad de paralelism într-un sistem serverless versus unul clasic în cloud.

## 1.4 Conținutul lucrării

Din punct de vedere al structurii, lucrarea va fi organizată în 2 părți :

- Partea teoretică în care va fi analizat Durable Task Framework și modul în care funcționează acesta;
- Partea practică ce va compara aceeași aplicație, dezvoltată în 2 moduri (clasic și folosind DTF).

În prima parte a lucrării va fi analizată tehnologia ce stă la baza soluțiilor de management al workflow-uri în cloud și anume, Durable Task Framework. Vom analiza modul în care este separat domeniul de execuție de domeniul de stocare, care sunt interfețele pe care trebuie să le respecte un provider, care sunt constrângerile care trebuie respectate atunci când se folosește această tehnologie și, bineînțeles, care sunt beneficiile și dezavantajele sale.

În a 2-a parte va fi analizat un exemplu clasic în literatura managementului de workflow-uri și anume cazul de rezervare multiplă în cazul unei călătorii. Această mini-aplicație a fost dezvoltată atât folosind metode clasice, cât și folosind Durable Task Framework. Folosind aceste 2 abordări, vor fi analizate :

- Capacitățile fiecărui sistem și nivelul de reziliență împotriva dezastrelor pe care îl pot demonstra;
- Diferențele de dezvoltare între cele 2 abordări;
- Analiză teoretică a costurilor între cele 2 arhitecturi;
- Performanța celor 2 sisteme.



## Capitolul 2

# Analiza Tehnologiei Durable Task Framework

### 2.1 Analiză generală

Durable Task Framework (DTFx) este o bibliotecă care permite utilizatorilor să gestioneze workflow-uri persistente de lungă durată (denumite orchestrări) în C# folosind sintagme clasice de codare `async/await`. DTF stă la baza soluțiilor Microsoft de gestionare a workflow-urilor durabile în cloud . Folosind această tehnologie, a fost expusă prima soluție serverless ce permite gestionarea stării : Azure Durable Functions.

Din punct de vedere al arhitecturii framework-ului, acesta are 2 părți:

- DTF Core, ce conține nivelul de abstractizare al gestionării, și implementează conceptele de bază ce permit programarea orchestrărilor folosind sintagme `async/await`;
- DTF providers, care implementează interfețele expuse de Core pentru a folosi diferite medii de stocare, în funcție de nevoile dezvoltatorului.

La bază, framework-ul își propune să rezolve problema durabilității acțiunilor într-un sistem distribuit, în care este de așteptat ca orice piesă a sistemului poate pica în orice moment. Arhitectura framework-ului e bazată pe evenimente, spre deosebire de alte framework-uri anterioare ce se bazau pe stocarea întregii stări a aplicației la un moment dat.

Nevoia de garanții de execuție a făcut ca DTF să construiască un sistem ce permite construcția unei structuri arborescente ce reprezintă o mașină de stări finite, în care, frunzele arborelui (activitățile, în terminologia DTF) au garanția că vor fi executate cel puțin o dată. Această garanție vine și cu o constrângere : din cauză că activitățile nu sunt executate **exact o dată** este necesar ca toate acțiunile din cadrul unei activități să fie idempotente.

## 2.2 Folosirea evenimentelor în DTF

Modelul de arhitectură bazat pe evenimente este un model popular de arhitectură asincronă distribuită, utilizat pentru a produce aplicații foarte scalabile. De asemenea, este foarte adaptabil și poate fi utilizat pentru aplicații mici și, de asemenea, pentru cele mari și complexe. Arhitectura bazată pe evenimente este alcătuită din componente de procesare a evenimentelor extrem de decuplate, cu un singur scop, care primesc și procesează evenimentele în mod asincron.

Modelul de arhitectură bazat pe evenimente constă din două topologii principale, mediatorul și brokerul. Topologia mediatorului este folosită în mod obișnuit atunci când există nevoia de orchestrare a mai multor pași în cadrul unui eveniment printr-un mediator central, în timp ce topologia brokerului este utilizată atunci când dorim să înlănțuim evenimente fără a utiliza un mediator central. Deoarece caracteristicile arhitecturii și strategiile de implementare diferă între aceste două topologii, este important ca ambele să fie cunoscute pentru a putea înțelege motivația din spatele alegerii arhitecturii DTF.

Gestionarea workflow-urilor prin mediator central, cunoscută și drept **Orchestrare** în literatura de specialitate (*Megargel et al.*[6]), constă în prezența unei entități centrale ce comunică cu toate celelalte sisteme ce iau parte la un anumit workflow pentru a garanta execuția cu succes a unei mașini de stări. Topologia mediatorului este utilă pentru evenimentele care au mai mulți pași și necesită un anumit nivel de orchestrare pentru a procesa evenimentul. Așa cum vom vedea și în partea practică a acestei lucrări, sunt nenumărate cazuri ce se reduc la gestionarea unei mașini de stări. Un astfel de exemplu este un lanț de aprovizionare, în care, de la momentul achiziției până la momentul livrării sunt numeroase etape ce trebuie îndeplinite cu succes pentru ca operațiunea ca un întreg să fie un succes. Printre aceste etape, s-ar număra pregătirea coletului, predarea lui către firma de curierat, înregistrarea plății și în final, livrarea propriu zisă. Un astfel de sistem poate beneficia de un mediator central, care asigură gestiunea stării unui astfel de workflow.

Gestionarea workflow-urilor printr-o topologie cu broker, cunoscută drept **Coreografie**, constă în distribuirea gestiunii mașinii de stări între toți consumatorii evenimentelor, astfel încât fiecare poate lua o anumită decizie în funcție de contextul în care se află atunci când procesează un eveniment. În această topologie nu există un mediator central, iar fiecare participant în workflow are responsabilitatea deciderii următoarelor acțiuni, în momentul procesării unui eveniment. Astfel, acest gen de arhitectură este potrivită pentru workflow-uri minimale, fără mașini de stări complicate, care nu au nevoie de un loc centralizat pentru asigurarea consistenței stării. Altfel, folosind această arhitectură în sisteme complexe, se poate ajunge foarte repede la haos, deoarece deciziile sunt împărțite peste tot, devenind imposibilă urmărirea firului logic al unei astfel de execuții distribuite.

Durable task framework, implementează o arhitectură de tip mediator central, și

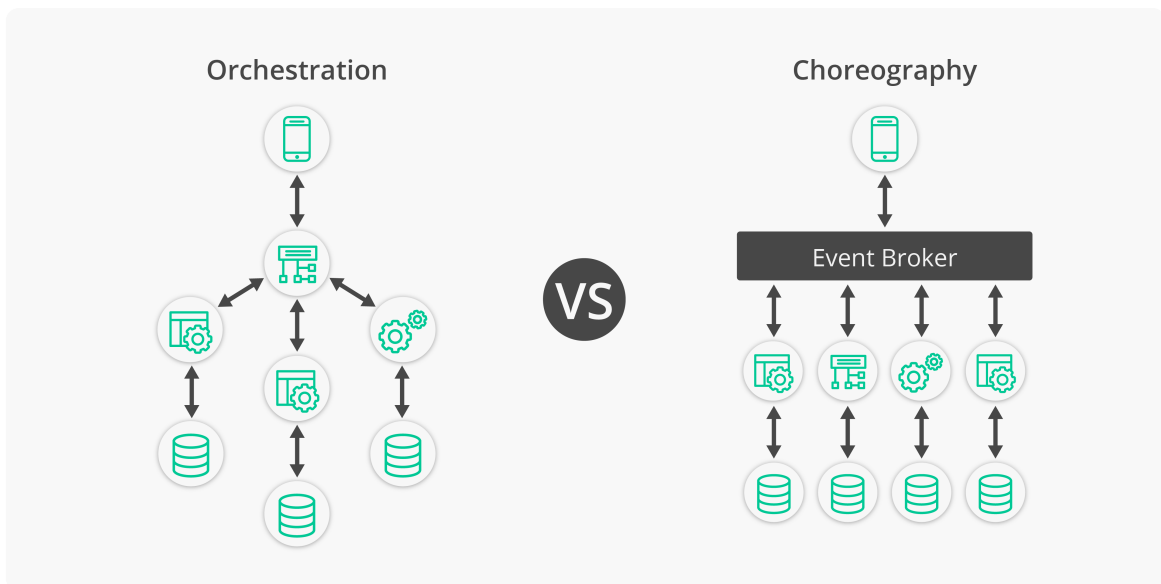


Figura 2.1: Orchestrare versus Coreografie

Sursă: <https://solace.com/blog/microservices-choreography-vs-orchestration/>

expune bazele pentru a putea crea propriul mediator pentru gestiunea unui workflow de lungă durată. Pentru a analiza implementarea conceptului de orchestrare bazată pe evenimente, vom analiza implementarea de DTF împreună cu Sql Server Provider, pentru a face o transpunere a conceptelor teoretice în lumea DTF. În figura 2.2 se poate observa structura schemei ce susține framework-ul atunci când este rulat împreună cu un SQL Server Provider. Acesta implementează conceptele teoretice într-un nivel de stocare bazat pe SQL server.

Există patru tipuri principale de componente de arhitectură în topologia mediatorului:

- cozi de evenimente
- mediatorul de evenimente
- canale de evenimente
- procesatoarele de evenimente.

Fluxul de evenimente începe cu un client care trimite un eveniment la o coadă de evenimente, care este utilizată pentru a transporta evenimentul la mediatorul de evenimente. Mediatorul de evenimente primește evenimentul inițial și orchestrează acel eveniment prin trimiterea de evenimente asincrone suplimentare către canalele de evenimente pentru a executa fiecare pas al procesului. Procesatoarele de evenimente, care ascultă pe canalele de evenimente, primesc evenimentul de la mediatorul de evenimente și execută o logică de afaceri specifică pentru a procesa evenimentul. Vom analiza fiecare componentă teoretică a procesului pentru a stabili paralela către structura din DTF, urmărind schema prezentată în figura 2.2.

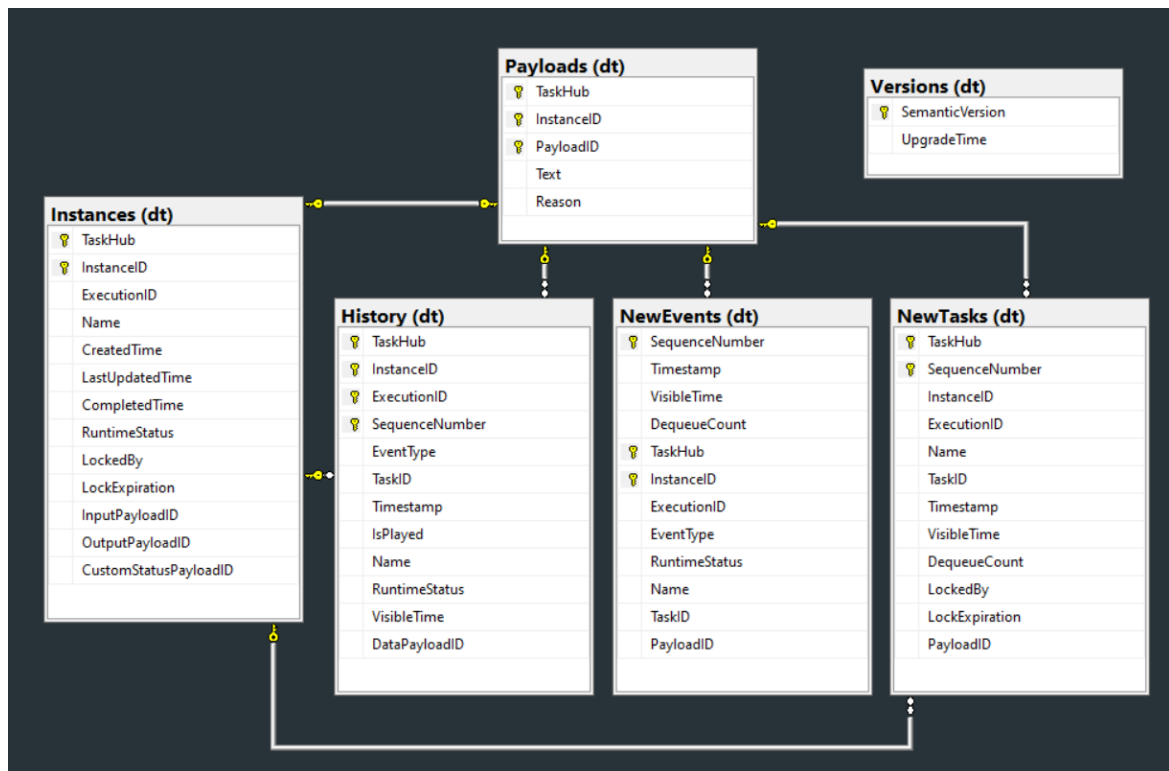


Figura 2.2: DTF Schema în DTF Sql Server provider

Sursă: <https://microsoft.github.io/durabletask-mssql/>

În cazul nostru, coada de evenimente este tabela 'New Events' iar modalitatea prin care un eveniment ajunge în aceasta, este prin folosirea unui DTF Client pentru a înregistra un eveniment pentru o anumită orchestrare.

Mediatorul de evenimente este 'Orchestrarea' respectiv logica definită de dezvoltator pentru a gestiona workflow-ul. Această logică este rulată de fiecare dată când un eveniment este primit pentru un anumit workflow. Deși codul este rulat de mai multe ori, operațiune denumită și 'Replay', acțiunile care deja au mai fost executate o dată nu sunt re-executate, ci rezultatele lor sunt direct încărcate direct din memorie, folosind istoricul orchestrării. Istoricul este salvat în tabela 'History'.

Canalele de evenimente sunt diferitele Task-uri înregistrate în tabelul 'NewTasks'. Acestea sunt folosite de pentru a declanșa execuția unor Activități, ce reprezintă frunzele arborelui de execuție a unui workflow. Task-urile sunt înregistrate pe canalele de evenimente de către mediatorul de evenimente, pentru a fi mai apoi procesate de procesatorul de evenimente. La finalul procesării unui task din canalul de evenimente, procesatorul de evenimente emite un nou eveniment pe coada de evenimente, pentru ca mediatorul să poată continua execuția workflow-ului.

Procesatoarele de evenimente sunt Workerii de DTF și reprezintă logica din DTF Core care monitorizează în mod constant canalele de evenimente pentru a prelua eventualele noi Task-uri ce sunt disponibile pentru execuție. Astfel, se obține o

arhitectură foarte decuplată în care înregistrarea de evenimente, gestionarea stării workflow-ului, execuția propriu zisă a diverselor task-uri și întoarcerea răspunsurilor înapoi către mediator sunt toate operațiuni decuplate, independente una de cealaltă.

Cel mai important aspect ce reiese din analiza implementării conceptelor de orchestrare bazată pe evenimente, este că datorită operațiilor independente, eșecurile care pot apărea atunci când această logica este executată într-un mediu distribuit nu vor afecta starea workflow-ului ca un întreg, fiindcă eșecul într-o parte a arhitecturii nu afectează alte părți, iar fiecare unitate este independent reîncercabilă. Astfel, revenirea în urma unui eșec (eroare de rețea, deconectare completă fie a workerilor fie a client-ului) sunt gestionate într-un mod elegant, fără efecte secundare.

## 2.3 DTF în Durable Functions

Conceptele prezentate în capitolul anterior, au fost reimplementate în DTF pentru a putea executa această tehnologie folosind soluții native în cloud, și pentru a expune o soluție pentru care există o mare nevoie în piață : Statefull Serverless functions. Astfel, a fost dezvoltat un nou provider pentru DTF, bazat pe Azure Storage tables de această dată, pentru a expune aceeași funcționalitate și în cloud.

Logica de bază a framework-ului a rămas aceeași, în schimb implementarea celor 4 concepte de bază într-o arhitectură bazată pe evenimente (cozi de evenimente, mediatorul de evenimente ,canale de evenimente, procesatoarele de evenimente) este schimbată.

Această adaptare a dus la lansarea unei soluții noi pe piață ce a creat un nou domeniu în lumea serverless: funcții ce sunt capabile să gestioneze starea, și să ruleze o perioadă foarte îndelungată de timp, dar pentru care se plătește doar timpul de execuție propriu zis. Acest lucru vine cu beneficii foarte mari din punct de vedere al costurilor, deoarece într-un workflow ce conține și pași manuali, cum ar fi aprobarea de către un operator a unei tranzacții, pe perioada în care se așteaptă acest eveniment, codul orchestrării nu se execută, deci practic nu există un cost asociat acestei operații.

Pe lângă funcționalitățile ce există în mod standard în toți providerii de DTF, Durable Functions vine cu o caracteristică suplimentară numită Durable Entities. Durable Entities definesc operațiuni pentru citirea și actualizarea unor mici părți de stare, cunoscute sub numele de entități durabile. La fel ca funcțiile de orchestrator, funcțiile de entitate sunt funcții cu un tip de declanșator special, declanșatorul de entitate. Spre deosebire de funcțiile de orchestrator, funcțiile de entitate gestionează starea unei entități în mod explicit, mai degrabă decât să reprezinte implicit starea prin fluxul de control. Entitățile oferă un mijloc de extindere a aplicațiilor prin distribuirea lucrării în mai multe entități, fiecare cu o stare de dimensiuni modeste. Practic, acest lucru oferă suport automat pentru gestiunea unei stări independente de cea a orchestrării, dar într-un mod distribuit și sigur, deoarece oferă garanții de access

serializat la aceasta.

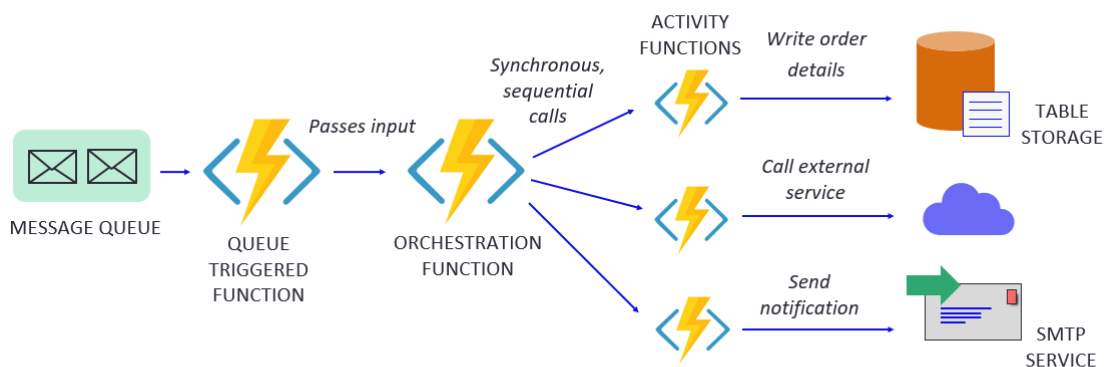


Figura 2.3: Exemplu arhitectură folosind Durable Functions

Sursă: <https://thisiszone.medium.com>

Așa cum se poate observa și în figura 2.3, conceptele clasice de DTF sunt transpuse în funcții individuale, de diferite tipuri. Astfel, o orchestrare este reprezentată de un Orchestration Function, o activitate este reprezentată de un Activity Function, în timp ce starea globală a workflow-ului este stocată în tabele în Azure Storage. Obținem astfel o arhitectură nativă în cloud, fără costuri de mentenanță a infrastructurii și de scalare, având ca beneficiu principal faptul că gestionarea stării într-un mediu distribuit nu mai este responsabilitatea explicită a dezvoltatorului, ci este rezolvată implicit de către providerul de soluții cloud.

## 2.4 Dificultăți în gestionarea workflow-urilor de lungă durată - Versionarea Orchestrărilor

Pe lângă beneficiile pe care le are o arhitectură bazată pe un mediator pentru gestiunea workflow-urilor, această centralizare are și anumite limitări și dezavantaje. Printre acestea, poate cel mai dificil lucru de gestionat este versionarea acestora.

Natura îndelungată a acestor workflow-uri face ca acestea să aibă următoarele particularități:

- Un workflow poate dura mai mult de un ciclu de dezvoltare (Mai multe versiuni ale aceluiași workflow pot exista până când o anumită instanță a unei anume versiuni finalizează execuția).
- Codul de orchestrare trebuie să fie determinist deoarece acesta va fi executat de mai multe ori pe perioada de viață a unui workflow.

Prima problema se rezumă la dificultatea de versionare a unei orchestrări. Atunci cand va apărea o versiune nouă a mașinii de stări gestionată de o anume orchestrare, cea veche nu poate fi direct înlocuită, deoarece încă există instanțe ale acelui tip de workflow care încă rulează. Astfel, versiunile noi de orchestrare trebuie mereu rulate în paralel cu cele vechi, până când toate instanțele ce rulează versiunea veche a workflow-urilor finalizează execuția. În acel moment versiunea veche a orchestrării va putea fi ștearsă.

Cea de-a doua problemă este mai mult doar o limitare și anume, din cauza logicii de replay pe perioada de viață a unei orchestrări, dezvoltatorul trebuie sa se asigure ca tot codul sursă din cadrul unei orchestrări va returna același rezultat indiferent de momentul în care se rulează. Pe scurt, comportamentul unei orchestrări trebuie sa fie determinist și constant indiferent de timp.

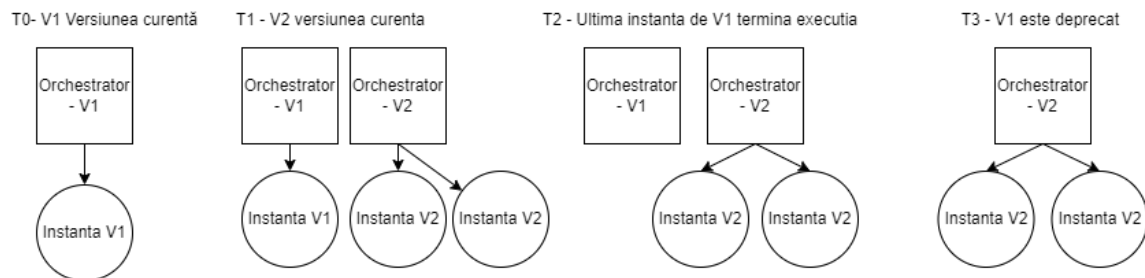


Figura 2.4: Ciclu de versionare a codului de orchestrare

# Capitolul 3

## Workflow-uri de lungă durată - Analiză paralelă a arhitecturii

### 3.1 Descrierea problemei

Atunci când vine vorba de managementul unui workflow de lungă durată, sau a unei tranzacții distribuite, un exemplu clasic în literatură este cel al unei rezervări de călătorie. O rezervare de călătorie presupune mai multe etape :

- Rezervarea unui taxi;
- Rezervarea hotelului;
- Rezervarea avionului.

Într-o lume monolitică, în care totul ar fi deținut de un singur serviciu ale cărei arhitecturi s-ar fi bazat pe o bază de date relațională, garantarea rezervării cu succes ar fi fost o problemă ușor de rezolvat având în vedere caracteristica de atomicitate a unei tranzacții. În schimb, într-o lume distribuită, în care fiecare serviciu este complet independent (Taxi Pelicanul, Hotel Capșa și Blue Air ) sistemul de gestiune a unei rezervări trebuie să fie rezilient la toate erorile ce pot apărea la diferite niveluri ale aplicației. Astfel sistemul nostru trebuie să garanteze nu numai că rezervarea se face cu succes, dar în cazul în care acest lucru nu se face, toate sistemele să fie aduse la starea inițială de dinaintea începerii tranzacției distribuite.

Recuperarea dintr-un eșec într-o tranzacție distribuită nu se poate face în mod nativ, deoarece aplicațiile ce iau parte la aceasta sunt complet independente, astfel că fiecare sistem ce participă la o astfel de tranzacție trebuie să expună și o funcționalitate de roll-back ce va anula o operație executată anterior. Astfel, putem concluziona că managementul unei tranzacții distribuite rezultă întotdeauna în managementul unui workflow de tip SAGA.

Saga este un concept arhitectural teoretic ce garantează execuția unui workflow într-un mod predictibil: Fie  $A_1, A_2, \dots, A_n$  acțiuni ce participă la o execuție Saga și



$RA_1, RA_2, \dots, RA_n$  acțiunile de rollback corespunzător fiecărui pas din workflow. Un workflow de tip Saga garantează fie că toți pașii  $A_1, A_2, \dots, A_n$  au fost executați cu succes, fie  $A_1, A_2, \dots, A_t, RA_1, RA_2, \dots, RA_t$  au fost executați cu succes. Astfel, se garantează fie că toate acțiunile au fost executate cu succes, fie că toate acțiunile ce au apucat să fie executate până la un punct de eroare au fost anulate cu succes. În final, sistemul va fi mereu într-o stare consistentă.

Scopul părții practice a acestei lucrări este implementarea unui coordonator de tranzacții distribuite de lungă durată implementând o arhitectură Saga. Vor fi analizate atât dezvoltarea acestei aplicații folosind un framework care facilitează dezvoltarea orchestratoarelor, cât și dezvoltarea aplicației folosind tehnologii clasice (.net Web API combinat cu o bază de date relațională).

## 3.2 Implementarea arhitecturii în mod clasic

În mod clasic, implementarea unui serviciu ce oferă garanțiile unei arhitecturi SAGA, trebuie să fie rezistent la următoarele potențiale probleme :

- Posibilitatea ca procesul să fie închis chiar în timpul execuției;
- Posibilitatea ca unul din pași să eșueze din motive neimputabile sistemelor ce iau parte la acțiune (probleme de rețea).

### 3.2.1 Implementarea naivă

Vom porni de la următoarea secvență de cod :

```

1  var step = ''Started'';
2  try {
3      await sagaService.bookHotel(booking);
4      step = ''BookedHotel'';
5      await sagaService.bookTaxi(booking);
6      step = ''BookedTaxi'';
7      await sagaService.bookFlight(booking);
8      step = ''BookedFlight'';
9  } catch (Exception){
10     if (step == ''BookedHotel''){
11         await sagaService.cancelHotel(booking);
12     } else if (step == ''BookedTaxi''){
13         await sagaService.cancelHotel(booking);
14         await sagaService.cancelTaxi(booking);
15     }
16 }
```

Într-un sistem monolit această secțiune de cod ar fi o implementare corectă a unui SAGA, dar ce se poate întâmpla dacă analizăm cazurile la care trebuie să fie rezilient un astfel de sistem, într-o configurație distribuită ?

Analizând prima potențială problemă și anume posibilitatea ca procesul ce execută Saga să fie oprit în mijlocul execuției, putem evidenția potențialele probleme cu această abordare. Practic, dacă serverul este oprit oriunde între linia 4 și 14, va lăsa execuția într-o stare inconsistentă, cu posibilitatea de a avea consecințe serioase asupra utilizatorului. Ce se întâmplă dacă se execută doar rezervarea de Hotel, dar utilizatorul nu este notificat ? Acestuia îi va fi retrasă suma de pe card în mod abuziv, rezultând în posibile probleme legale pentru deținătorul aplicației. Astfel, garanția atomicității într-o tranzacție distribuită este foarte importantă.

Motivele pentru care această întrerupere a execuției practic face imposibilă garantarea atomicității întregii operații, este că toată acțiunea se execută ca parte a unei singure încercări, și orice întrerupere a firului de execuție face ca operația să nu poată fi reîncercată și eventual continuată, pentru a garanta execuția completă a strategiei Saga.

### 3.2.2 Execuție durabilă

O altă problemă cu codul inițial este că nu acoperă nici posibilitatea ca unul din pași să eșueze din motive de probleme de rețea, iar astfel, toată operațiunea poate rămâne într-o stare inconsistentă, dacă pe perioada roll-back-ului (între linia 10 și 14) unul din pași eșuează. Pentru a acoperi ambele probleme, codul ar trebui să fie transformat după cum urmează :

```
1  var booking = await sagaService.RegisterIfNotPresent(booking); // returneaza starea
2  // inregistrata in baza de date a rezervării sau inregistreaza rezervarea
3  // în baza de date, daca este prima execuție.
4  try {
5      if (!booking.isHotelBooked){
6          await sagaService.bookHotel(booking);
7          await sagaService.saveState(booking);
8      }
9      if (!booking.isTaxiBooked){
10         await sagaService.bookTaxi(booking);
11         await sagaService.saveState(booking);
12     }
13     if (!booking.isFlightBooked){
14         await sagaService.bookFlight(booking);
15         await sagaService.saveState(booking);
16     }
```

```

17 } catch (Exception){
18     try{
19         if (booking.isHotelBooked){
20             await sagaService.cancelHotel();
21             await sagaService.saveState(booking);
22         } else if (booking.isTaxiBooked){
23             await sagaService.cancelHotel();
24             await sagaService.cancelTaxi();
25             await sagaService.saveState(booking);
26         }
27     }
28     catch (Exception){
29         await sagaService.markAsFailed(booking);
30     }
31 }

```

Odată cu modificarea secvenței de cod, putem observa următoarele modificări :

- Introducerea unui nivel de decuplare, la nivelul bazei de date, ce permite execuția repetată a secvenței de cod fără a pierde starea precedentă, pentru a evita trimiterea mai multor request-uri prin rețea decât este necesar;
- Introducerea unui nou nivel de tratare a excepțiilor, care prinde eventualele probleme de rețea la nivelul acțiunii de rollback, și care marchează rezervarea ca fiind eșuată, pentru a putea fi reparată în mod manual. (atunci când execuția Saga nu poate fi executată pe flow-ul de rollback, acțiunea trebuie fixată manual).

Utilizând această nouă strategie, codul nu va mai fi executat ca parte dintr-un singur context, ci aplicația va încerca să consume toate rezervările ce nu sunt într-o stare finală, pana când acestea fie sunt executate și au ajuns într-o stare finală corectă, fie sunt marcate ca eșuate și necesită o recuperare manuală.

Pentru ca acest lucru să fie posibil, trebuie implementat un mecanism specializat care să ruleze la intervale predefinite și să încerce să consume rezervările ce așteaptă să fie procesate. Această decuplare ajută din punct de vedere al rezilienței sistemului, dar vine cu un cost suplimentar de dezvoltare deoarece această tehnică nu este standard în arhitectura unui Web API.

### 3.2.3 Garanții de execuție. Cel puțin o dată versus exact o dată

Deși o parte din probleme au fost rezolvate, încă rămân în picioare anumite vulnerabilități, deoarece sistemul încă poate suferi de pe urma unei întreruperi spontane între momentul în care se execută o acțiune, și momentul în care aceasta este salvată în baza de date.

Din cauza acestui lucru, sistemul obținut oferă doar garanția că toate sistemele ce iau parte la un workflow, vor fi apelate cel puțin o dată, dar nu exact o dată. Problema garanției de execuție exact o dată în cadrul unui sistem distribuit este intens dezbătută și în multe cazuri, imposibil de obținut. Atât timp cât endpoint-urile sistemelor care iau parte la workflow sunt idempotente, garanția că aceste endpoint-uri vor fi apelate cel puțin o dată este suficientă.

În varianta finală a acestei implementări componentele implicate sunt :

- Serviciul ce execută mașina de stări a workflow-ului (partea de cod discutată mai sus)
- Serviciul ce se ocupă de preluarea rezervărilor din baza de date și pornirea serviciului de mai sus.
- Baza de date, ce se ocupă de persistarea stărilor pentru a permite o execuție durabilă

### **3.2.4 Interacțiunea din exterior cu workflow-ul. Operații de lungă durată**

Până acum, implementarea a analizat doar problemele de durabilitate a workflow-ului pe cazul rapid, atunci când toate sistemele pot răspunde în mod sincron la apelurile trimise de sistemul ce implementează Saga. Ce se întâmplă însă dacă unul din sisteme poate oferi un răspuns foarte lent, spre exemplu în 24 de ore ? Implementarea curentă nu permite interacțiuni cu workflow-ul din exterior, deci toată arhitectura trebuie modificată pentru a cuprinde și acest caz.

Pentru a permite interacțiunea cu workflow-ul din exterior, sistemul va expune și endpointuri externe pentru a modifica starea unei rezervări. Astfel, sistemul ce implementează Saga doar va anunța sistemul hotelier legat de cererea de rezervare a camerei, iar sistemul hotelier va veni cu un apel către sistemul central odată ce finalizează rezervarea (posibil peste o perioadă lungă de timp) și va modifica în mod direct starea rezervării. La nivelul serviciului ce implementează mașina de stări, aceasta va suferi o modificare în sensul că, dacă la un moment dat nu se află într-o stare ce permite continuarea ( de exemplu după trimirea request-ului de rezervare a camerei, camera nu este rezervată) aceasta va termina execuția (return) și va aștepta următoarea procesare (ce va fi inițiată de Serviciul ce se ocupă de preluarea rezervărilor din baza de date) pentru a verifica din nou starea și pentru a trece eventual la etapa următoare.

Serviciile ce interacționează cu rezervarea vor putea modifica starea acestuia accesând endpoint-urile dedicate fiecărei operații, folosind id-ul rezervării primit pe request. De exemplu, dacă sistemul de rezervare a zborurilor expune un endpoint de rezervare de zbor `POST /api/flights` , în sistemul central va exista un endpoint pereche `POST`

/api/bookings/{bookingId}/book\_flight ce va permite tranziția rezervării în starea FlightBooked. Astfel, workflow-ul propriu zis nu va fi executat pe perioada în care se așteaptă finalizarea unei acțiuni din partea unui alt sistem.

### 3.2.5 Cronometre durabile

Deși sistemul în această formă permite interacțiunea cu alte servicii, există o potențială vulnerabilitate și anume, dacă un serviciu, din diverse motive, nu mai apelează endpoint-ul corespunzător pentru a trece workflow-ul în starea corectă. În acest caz, workflow-ul va rămâne pentru totdeauna într-o stare intermediară. Pentru a rezolva această problemă, trebuie să adăugăm suport pentru un cronometru durabil.

Cronometrul durabil este un concept teoretic într-o arhitectură orientată pe microservicii, și se referă la capacitatea unui cronometru de a rezista eventualelor întreruperi ale procesului. În cadrul sistemului nostru, pentru a obține un cronometru durabil elementar, pe entitatea rezervării se va salva și data la care a fost creat și data la care s-a făcut ultimul update. Astfel, orice cronometru durabil se obține prin compararea datei la care a fost creat/updatat plus un anumit delta, cu data curentă de la momentul execuției. Următoarea secvență de cod ilustrează cum poate fi verificat în mod durabil dacă workflow-ul a tranziționat în mod corect către o stare intermediară, în timp util.

```
1      var relativeTime = booking.CreationTime.AddMinutes(10);
2      if (!booking.isHotelBooked){
3          if (DateTime.Compare(relativeTime, DateTime.UtcNow) < 0 ){
4              throw new TimeoutException('Workflow-ul nu a trecut
5                  la pasul următor destul de rapid');
6          }
7          await sagaService.bookHotel(booking);
8          await sagaService.saveState(booking);
9      }
10     }
```

Codul de mai sus garantează că dacă workflow-ul nu tranziționează către o anumită stare (HotelBooked în acest caz) workflow-ul va porni strategia de rollback printr-o excepție. Astfel, sistemul devine rezilient la eventualele erori ce apar în serviciile ce iau parte la workflow și este capabil să readucă workflow-ul într-o stare consistentă fără intervenții din exterior.

### 3.2.6 Optimizarea numărului de request-uri

Deși garanția de cel puțin o dată ne permite ca request-urile către celelalte servicii să fie trimise de mai multe ori, beneficiind de faptul că acestea sunt idempotente, sistemul nu ar trebui să abuzeze și atât timp cât totul funcționează corect, sistemul ar trebui

să trimită request-urile exact o dată. Însă, analizând codul de mai sus și extinzând pattern-ul și la celelalte acțiuni, putem observa o problemă. Presupunând ca acțiunea de rezervare a unei camere de hotel durează 24 de ore, iar rezervarea noastră este preluată și procesată de către serviciul ce se ocupă de preluarea rezervărilor din baza de date o dată pe minut, atunci se vor trimite 1440 de request-uri către sistemul de rezervare a camerelor de hotel, în loc de unul, acest lucru fiind inacceptabil chiar și într-un sistem ce garantează execuția cel puțin o dată a acțiunilor.

Pentru a rezolva această problemă vor fi introduse stări intermediare, ce vor evidenția natura de lungă durată a operațiunilor și vor ajuta sistemul de gestiune al Saga să obțină execuția exact o dată a acțiunilor, atât timp cât nu apare nici o eroare în toate sistemele implicate.

```
1      var relativeTime = booking.CreationTime.AddMinutes(10);
2      if (!booking.isHotelBooked){
3          if (DateTime.Compare(relativeTime, DateTime.UtcNow) < 0 ){
4              throw new TimeoutException('Workflow-ul nu a trecut
5                  la pasul următor destul de rapid');
6          }
7          if (booking.status != ''BookingHotel''){
8              await sagaService.bookHotel(booking);
9              await sagaService.saveState(booking, 'BookingHotel');
10         }
11     }
```

Salvarea stărilor de tranziție în entitatea rezervării permite sistemului să decidă să nu mai execute o anumită acțiune pe perioada în care așteaptă un răspuns din exterior, datorită naturii de lungă durată a operațiunilor.

### 3.2.7 Considerente de deployment și scalabilitate

Având în vedere că arhitectura descrisă în această secțiune este bazată pe un server web și o bază de date relațională, atunci când vine vorba de deployment, avem multiple alegeri. În primul rând această arhitectură este pretabilă atât pentru a rula on-prem cât și în cloud.

On-prem avem posibilitatea să găzduim propriul nostru web-server și propriul server de baze de date pe propria infrastructură. Însă, acest lucru poate afecta scalabilitatea deoarece infrastructura proprie are unele limitări, una dintre cele mai mari fiind faptul că este greu de scalat, deoarece sunt implicate resurse fizice. Astfel, scalarea pe verticală pentru a obține performanțe mai mari de pe urma acestui serviciu este îngreunată dacă alegem să rulăm totul pe propria infrastructură. În schimb scalarea pe orizontală poate fi obținută relativ simplu adaugând mai multe servere și rulând mai multe instanțe ale aplicației noastre. Însă, în acel caz vor exista complicații

la nivel de rutare și gestiune a traficului ce este preluat de către aplicație. În contextul în care aplicația noastră gestionează workflow-uri de lungă durată, acestea nu pot fi executate în cadrul unui singur apel HTTP astfel încât să putem returna rezultatul ca răspuns. În schimb, utilizatorul care trimite primul request, de pornire a unui workflow, va trebui să revină către serviciul nostru cu un alt request, mai târziu, pentru a putea afla rezultatul final. Deoarece rezultatul va fi stocat la nivelul bazei de date, în cazul în care serviciul nostru este scalat orizontal și avem mai multe instanțe separate în mod fizic între ele, este foarte important unde vor ajunge apelurile viitoare ale unui consumator ce dorește să afle rezultatul unui workflow pe care deja l-a pornit. Acest lucru este critic deoarece datele despre un workflow nu există pe toate instanțele într-un setup scalat pe orizontală. Un utilizator care dorește să afle statusul workflow-ului cu id-ul 24566789 poate primi un răspuns doar dacă request-ul acestuia ajunge pe instanța care a și pornit workflow-ul 24566789. Acest lucru adaugă un nivel suplimentar de complexitate, ce trebuie gestionat la un nivel superior aplicației (de rutare) care să trimită request-urile către instanța corespunzătoare.

În cloud avem posibilitatea să găzduim aplicația noastră la oricare dintre providerii deja cunoscuți în industrie (AWS, Azure, Google Cloud) deoarece toți au suport pentru a găzdui un web-server și baza de date în numeroase modalități. Deși problema scalabilității pe orizontală rămâne validă și în Cloud, aici în schimb putem alege varianta scălării pe orizontală doar a web-server-ului, în timp ce baza de date poate fi scalată pe verticală, fără a suferi întreruperi ale aplicației. Astfel, evităm complexitatea managementului de rutare către instanța corectă, deoarece toate instanțele serverului web vor fi conectate la aceeași bază de date și implicit vor putea accesa datele tuturor workflow-urilor. Astfel, execuția în cloud a acestei arhitecturi poate veni atât cu avantaje de performanță și scalabilitate, cât și cu reduceri semnificative de costuri, deoarece toate resursele pot fi scalate în mod automat în funcție de nevoile aplicației la un anumit moment dat, rezultând într-o utilizare mult mai eficientă a resurselor.

### 3.3 Implementarea arhitecturii folosind Durable Functions

În capitolul anterior am putut observa dificultățile în a implementa un serviciu care să respecte conceptul de execuție Saga pentru a gestiona o tranzacție distribuită. Pornind de la aceeași problemă, în acest capitol va fi analizată dezvoltarea unui serviciu care folosește de această dată Durable Functions, pentru a analiza care sunt problemele pe care această tehnologie le rezolvă.

Pentru a putea înțelege mai bine arhitectura execuției unei Saga folosind Durable functions, vor fi analizate conceptele ce stau la baza dezvoltării unei soluții folosind Durable functions :

**Funcțiile de orchestrare** descriu modul și ordinea în care sunt executate acțiunile.

Funcțiile orchestrator descriu orchestrarea în cod (C# sau JavaScript). O orchestrare poate avea multe tipuri diferite de acțiuni, inclusiv funcții de activitate, sub-orchestrări, așteptare pentru evenimente externe, HTTP și cronometre durabile. Funcțiile de orchestrator pot interacționa și cu entități durabile. Funcțiile orchestrator sunt scrise folosind cod obișnuit, dar există cerințe stricte privind modul de scriere a codului. Mai exact, codul funcției de orchestrator trebuie să fie determinist. Nerespectarea acestor cerințe de determinism poate face ca funcțiile orchestratorului să nu ruleze corect. Codul funcției orchestrator este rulat de mai multe ori, asemănător cu modalitatea în care în capitolul anterior am ales un mecanism de polling care rulează codul de orchestrare până orchestrarea ajunge într-o stare finală. Deși implementarea propriu zisă diferă în multe zone, conceptul de bază este același, iar aceasta vine cu nevoia de a scrie cod determinist, pentru a nu avea rezultate diferite pentru aceeași orchestrare, în funcție de momentul în care a fost executată.

**Funcțiile de activitate** sunt unitatea de bază de lucru într-o orchestrare durabilă a funcțiilor. Funcțiile de activitate sunt funcțiile și sarcinile care sunt orchestrate în proces. De exemplu, în cazul analizat în cadrul acestei lucrări, pentru a procesa o rezervare este necesară execuția unor sarcini. Aceste sarcini implică comunicarea cu serviciul de taxi, cu cel hotelier și cu cel aviatic. Fiecare dintre aceste sarcini individuale poate fi modelat ca o activitate. Aceste funcții de activitate pot fi executate în serie, în paralel sau o combinație a ambelor. Spre deosebire de funcțiile de orchestrator, funcțiile de activitate nu sunt restricționate în ceea ce privește tipul de lucru pe care îl putem face în ele. Funcțiile de activitate sunt frecvent utilizate pentru a efectua apeluri în rețea sau pentru a rula operațiuni cu consum intensiv de CPU. O funcție de activitate poate, de asemenea, să returneze date înapoi către funcția de orchestrator. Durable Task Framework garantează că fiecare funcție de activitate apelată va fi executată cel puțin o dată în timpul execuției unei orchestrări.

**Funcțiile entitate** definesc operațiuni pentru citirea și actualizarea unor părți mici de stare. Adesea ne referim la aceste entități cu stare ca entități durabile. La fel ca funcțiile de orchestrator, funcțiile de entitate sunt funcții cu un tip de declanșator special, declanșator de entitate. Ele pot fi, de asemenea, invocate din funcțiile client sau din funcțiile orchestrator. Spre deosebire de funcțiile de orchestrator, funcțiile de entitate nu au constrângeri specifice de cod. Funcțiile de entitate gestionează starea în mod explicit, mai degrabă, decât să reprezinte implicit starea prin fluxul de control. Acestea nu sunt re-rulate în mod constant, în schimb, garantează accesul în serie la datele interne, astfel este sigur că aceste entități durabile să fie accesate din multiple activități sau orchestrări în mod concurent, fără a ne îngrijora legat de integritatea datelor.

**Cronometrele Durabile** din Durable Functions implementează același concept descris și în capitolul anterior, și oferă o abstractizare peste implementarea standard, pentru o utilizare rapidă.



**Evenimentele Externe** de asemenea implementează un concept similar cu problema interacțiunii cu procesele în curs, descris anterior, și facilitează comunicarea din exterior către o orchestrare care încă rulează. Combinația dintre suportul pentru evenimente externe și cronometre durabile rezultă într-un suport foarte complex, care acoperă multe situații reale, ce necesită comunicări asincrone între serviciul de orchestrare și celelalte servicii terțe care iau parte la acțiune.

### 3.3.1 Noua Arhitectură

Folosind conceptele tocmai definite, vom mapa problemele ce trebuie rezolvate pentru a implementa serviciul de gestiune a rezervărilor către soluțiile puse la dispoziție de Durable Functions.

Analizând 3.1 putem observa următoarea corespondență între conceptele de bază expuse de Durable Functions, și necesitățile noastre :

- Nucleul serviciului nostru de gestiune al operațiunii de rezervare este plasat într-o funcție de orchestrare, ce garantează execuția durabilă a logicii, astfel încât Saga să fie executată cu succes indiferent de eventualele întreruperi mecanice sau excepții neașteptate în timpul execuției.
- Fiecare interacțiune cu serviciile externe este izolată într-o funcție de activitate, pentru a permite execuția lor conform unor politici de retry și pentru a izola comportamentul nedeterminist (apeluri asincrone prin rețea) de logica principală de orchestrare.
- Starea generală a rezervării este stocată prin intermediul unei funcții entitate ce asigură capacitatea de a citi și updata starea rezervării în mod distribuit, pe măsura ce serviciile externe procesează request-urile trimise de către serviciul principal.

Analizând strict codul de orchestrare putem observa că acesta nu este cu mult mai complex, deoarece conceptele de durabilitate sunt abstractizate sub interfețe ușor de folosit.

```
1 var entityGuid = context.GetInput<string>();
2 var entityId = new EntityId(nameof(Booking), entityGuid);
3 var booking = await context.CallEntityAsync<Booking>(entityId, "Get");
4 var isHotelBooked= await context.CallActivityAsync<bool>("BookHotel", null);
5 booking = await context.CallEntityAsync(new EntityId(nameof(Booking),
    entityId),"UpdateHotel", isHotelBooked);
6 if (booking.Hotel)
7 {
8     var isTaxiBooked = await context.CallActivityAsync<bool>("BookTaxi",
        null);
```

```

9      booking = await context.CallEntityAsync(new EntityId(nameof(Booking),
      entityGuid),"UpdateTaxi", isTaxiBooked);
10     if (booking.Taxi)
11     {
12         var isFlightBooked = await context.CallActivityAsync<bool>("
            BookFlight", null);
13         booking = await context.CallEntityAsync(new EntityId(nameof(Booking)
            , entityGuid),"UpdateFlight", isFlightBooked);
14     }
15     else // hotel booked, taxi not booked - cancel hotel
16     {
17         await context.CallActivityAsync<bool>("CancelHotel", null);
18         booking = await context.CallEntityAsync(new EntityId(nameof(Booking)
            , entityGuid),"UpdateHotel", false);
19     }
20     // hotel booked, taxi booked, flight not booked - cancel hotel & taxi
21     if (booking.Hotel && booking.Taxi && !booking.Flight)
22     {
23         await context.CallActivityAsync<bool>("CancelHotel", null);
24         booking = await context.CallEntityAsync(new EntityId(nameof(
            Booking), entityGuid),"UpdateHotel", false);
25         await context.CallActivityAsync<bool>("CancelTaxi", null);
26         booking = await context.CallEntityAsync(new EntityId(nameof(
            Booking), entityGuid), "UpdateTaxi", false);
27     }
28 }

```

Folosind Durable Functions, codul de orchestrare se simplifică considerabil, deoarece excepțiile ce apar în cadrul unei orchestrări declanșează în mod automat marcarea acelei orchestrări ca fiind *Failed*, ceea ce poate fi monitorizat în mod automat pentru a fixa eventualele cazuri de eșec pe flow-ul de rollback.

Din punct de vedere al structurii codului, acesta descrie în mod procedural mașina de stări ce este implementată de serviciu, iar framework-ul asigură execuția acestuia în mod durabil, printr-un mecanism de execuție al orchestrării bazat pe event-sourcing. Practic, orchestrarea este re-executată de fiecare dată când un eveniment este înregistrat pentru aceasta. Evenimentele pot avea diferite surse, interne sau externe. Evenimentele externe sunt cele inițiate de utilizatori sau alte servicii, de exemplu când serviciul de orchestrare este anunțat printr-un eveniment că o anumită acțiune s-a terminat. Evenimentele interne sunt cele care determină re-execuția unei orchestrări când un cronometru durabil care îi se adresează expiră, sau o sub-orchestrare sau o activitate ce este înlănțuită în mod direct este finalizată.

Atunci când o orchestrare este re-executată, aceasta sare în mod automat peste pașii care au fost executați deja. De fiecare dată când prin intermediul contextului

este înregistrată o sub-acțiune a unei orchestrări, o intrare în tabela de istoric este generată care înregistrează statusul și eventualul rezultat al acelei sub-acțiuni. Atunci când orchestrarea este re-executată, aceasta încarcă din baza de date toți pașii care au fost deja executați și înregistrează doar acțiunile noi ce trebuie înregistrate. Astfel, este practic suportată în mod standard o strategie care minimizează request-urile către serviciile terțe și care oferă o garanție de execuție cel puțin o dată a activităților, fără însă a abuza în mod activ de aceasta. În cazul în care pe perioada execuției nici o excepție neașteptată nu apare, fiecare activitate este executată exact o dată. Acest suport care în mod clasic necesită destul de mult design și efort de implementare este oferit de către Durable Functions, acesta fiind un alt avantaj pentru a alege această tehnologie pentru a manageria workflow-uri de lungă durată în cloud.

La nivelul Activităților, acestea gestionează comunicarea cu serviciile externe. Din punct de vedere al execuției acestea sunt rulate urmând o garanție de cel puțin o dată, iar din acest motiv implementarea lor este necesar să fie idempotentă, pentru a evita potențialele acțiuni duplicate, în cazul unei execuții multiple. Pe de altă parte, din cauză ca acestea nu sunt re-executate de mai multe ori în mod recurent, acestea nu trebuie să respecte aceleași restricții ca orchestrările, și pot avea comportament nedeterminist. Rezultatul acestora este returnat către orchestrări, iar acestea la rândul lor updatează entitatea care menține starea rezervării.

### 3.3.2 Cronometre Durabile în Durable Functions

Cronometrele Durabile sunt suportate în mod standard de către Durable Functions prin intermediul contextului de execuție. Conceptul din spate e asemănător cu cel propus în capitolul anterior, dar diferă deoarece cronometrul odată înregistrat, este gestionat ca un eveniment de sine stătător, și nu este strâns legat de execuția orchestrării.

```
1 DateTime expireAt = context.CurrentUtcDateTime.AddHours(48);
2 var cts = new CancellationTokenSource()
3 var timeoutTask = context.CreateTimer(expireAt, cts);
4 if (!booking.isHotelBooked){
5     var hotelBookingTask = context.CallActivityAsync<bool>("BookFlight",
6         null);
7     Task winner = await Task.WhenAny(hotelBookingTask, timeoutTask);
8     if (winner == hotelBookingTask)
9     {
10         cts.Cancel();
11         booking = await context.CallEntityAsync(new EntityId(nameof(Booking),
12             entityGuid),"UpdateFlight", isFlightBooked);
13     }
14 }
```

```

14         throw new TimeoutException('The Timer expired before receiving a
           response from the hotel activity');
15     }
16 }
17 }

```

Putem observa că timpul mereu este calculat folosind timpul curent extras din contextul de execuție. Acesta este un mod determinist de a extrage timpul curent, fiindcă va returna mereu aceeași valoare, chiar dacă orchestrarea este re-executată la diverse momente în timp. După crearea inițială, acest timer este înregistrat în baza de date, iar în momentul în care acesta expiră, va declanșa o re-execuție a orchestrării. În acel moment timer-ul va fi fost finalizat și, urmărind codul de mai sus, se va arunca o excepție. Dacă până în momentul în care timer-ul expiră este finalizată altă acțiune care concursa cu timer-ul, atunci timer-ul trebuie să fie explicit închis folosind CancellationToken-ul pentru a preveni re-execuții inutile ale orchestrării. În cazul în care timer-ele rămân deschise chiar și atunci când nu mai sunt așteptate, acest lucru duce la o degradare a performanței funcțiilor noastre durabile.

### 3.3.3 Considerente de deployment și scalabilitate

Din punct de vedere al deployment-ului soluția ce folosește Durable Functions suferă de "vendor lock-in". Acest lucru înseamnă că soluția aceasta nu este generică din punct de vedere al deployment-ului, și dezvoltatorul are o singură soluție: deployment-ul în Azure Cloud folosind Durable Functions. Execuția soluției on-prem sau portarea acesteia către un alt provider de soluții cloud nu este posibilă.

În schimb, soluția disponibilă de a rula aplicația folosind Durable Functions, vine cu 2 mari avantaje :

- Scalare pe orizontală infinită, fără restricții de performanță și fără complexitate suplimentară în nivelul de rutare, deoarece acest lucru este deja gestionat de Azure.
- Mod de calcul al costurilor foarte avantajos, deoarece se plătește doar timpul de execuție, nu și timpul de așteptare. Iar într-un sistem în care timpul de execuție reprezintă mai puțin de 2% din timpul total de viață a workflow-ului (deoarece majoritatea timpului este petrecut așteptând timere sau evenimente externe) acest lucru este foarte valoros și duce la o reducere considerabilă a costurilor de execuție a unui astfel de sistem.

Deși această limitare a opțiunilor de deployment poate fi o vulnerabilitate pe viitor, pentru moment această soluție este cea mai bună deoarece în piață nu există un competitor care să ofere tehnologie stateful serverless într-un mod mai bun.

### 3.3.4 Arhitectura non-serverless pentru execuția durabilă a workflow-urilor

Arhitectura serverless vine cu numeroase beneficii, mai ales în zona de management a deploymentului și a scalabilității automate. În schimb, așa cum a fost punctat și în secțiunea anterioară, alegerea acestei modalități de execuție suferă de "vendor lock-in", iar acest lucru poate fi un impediment pentru anumite companii, care preferă flexibilitate pe termen lung.

Alternativa la a lansa serviciul de management a workflow-urilor ca funcții serverless durabile, este să fie folosit framework-ul ce stă la baza acestora (Durable Task Framework), într-un mod self-hosted. Acest lucru înseamnă că din punct de vedere al infrastructurii, aplicația va semăna cu varianta inițială analizată, care nu folosea nici un framework. Serviciul va rula pe un server web clasic, având ca suport la nivelul stocării datelor, o bază de date relațională (aici DTF oferă mai multe opțiuni, dar pentru exemplul curent vom analiza varianta în care DTF providerul este bazat pe SQLServer). Din punct de vedere al scalabilității, această configurație are aceleași avantaje și dezavantaje ca orice aplicație web clasică. În schimb, din punct de vedere al managementului de workflow-uri de lungă durată, folosind DTF, toate primitivele necesare sunt deja implementate reducând semnificativ timpul de dezvoltare al unui astfel de serviciu. Astfel, putem pune DTF în categoria serviciilor ce îmbunătățesc productivitatea dezvoltatorilor.

Așa cum se poate observa și în figura 3.2, DTF realizează același potențial de scalabilitate ca sistemul clasic analizat inițial, prin posibilitatea de a adauga mai mulți workeri (instanțe ale serviciului ce folosește DTF) conectați la aceeași bază de date. Acest lucru este posibil, printr-un mecanism de locking la nivelul bazei de date, ce previne mai mulți workeri să consume aceeași muncă în mod concurent. Fiecare worker execută un mecanism de polling la nivelul bazei de date, ce încearcă să rezerve un work-item pentru acesta. Atât timp cât logica de selecție a următorului work-item este rezistentă la execuții paralele, atunci un număr nelimitat de workeri pot fi adaugați împotriva unei baze de date.

Abilitatea de a scala orizontal numărul de workeri conectați la o bază de date, evidențiază în același timp și vulnerabilitatea acestui sistem: Baza de date poate limita performanța sistemului, dacă numărul de workeri conectați la aceasta este prea mare. Singura soluție pentru a îmbunătăți performanța sistemului atunci când limitele bazei de date sunt atinse, este scalarea pe verticală a acesteia. Însa, scalarea pe verticală nu poate fi infinită, din cauza limitărilor din punct de vedere hardware care există acum în industrie.

Pentru a scala pe orizontală și la nivelul bazei de date, trebuie implementat un mecanism de sharding, ce conține și logică de rutare, asemănător celui descris în secțiunile anterioare. Acest lucru este necesar pentru a garanta că evenimentele transmise către un anumit workflow ajung către instanța care a și creat acel workflow.

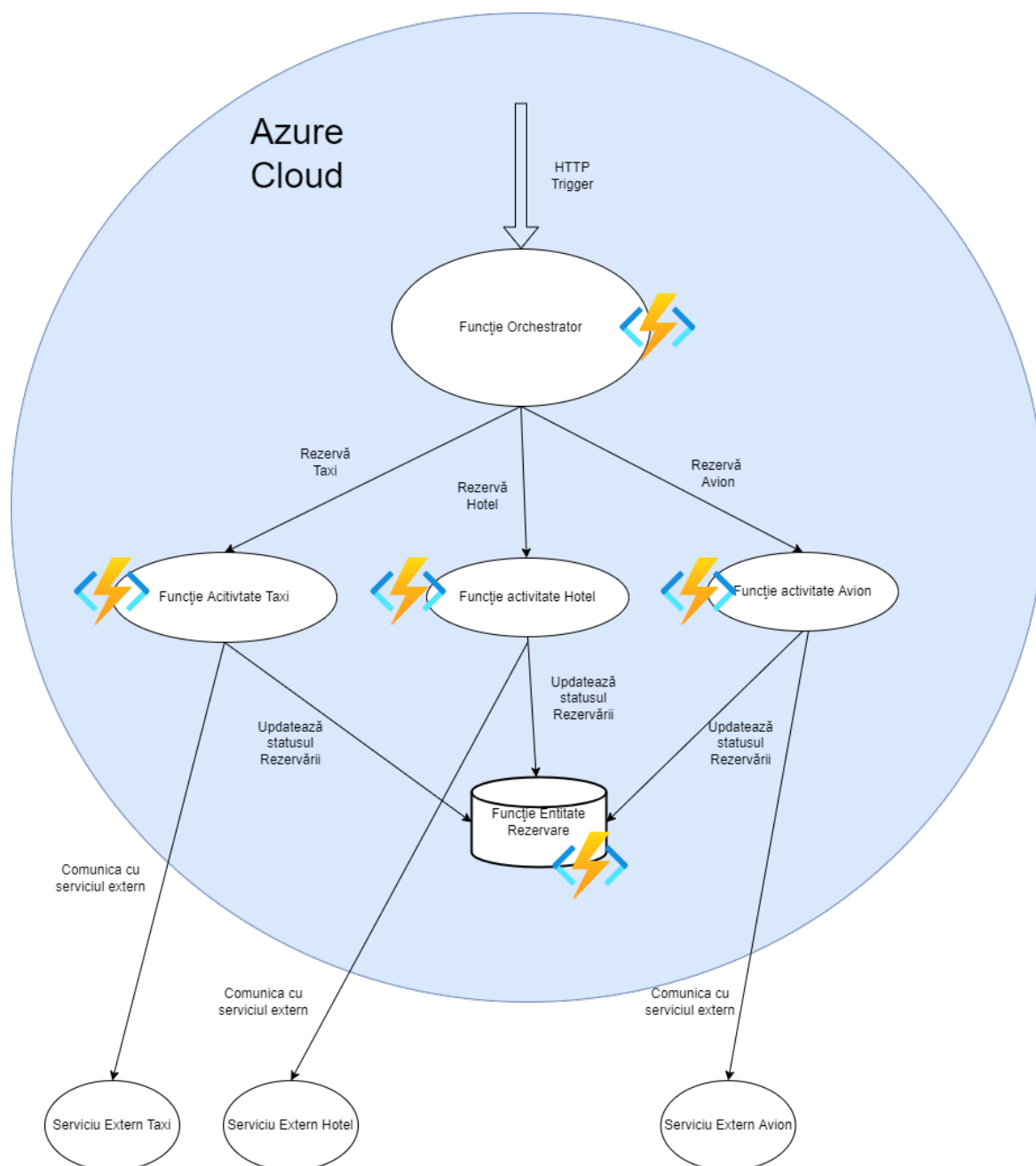


Figura 3.1: Rezervare distribuită folosind Durable Functions

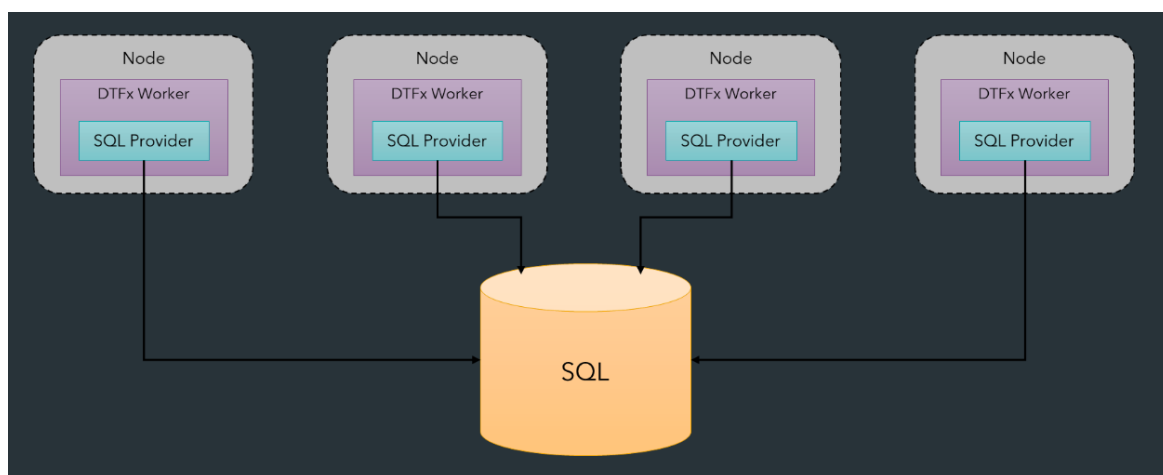


Figura 3.2: Scalare automată a workerilor DTF

Sursă: <https://microsoft.github.io/durabletask-mssql>

# Capitolul 4

## Rezultate și concluzii

Rezolvarea unei probleme comune a fost abordată în două moduri diferite. În urma acestui efort, au fost identificate numeroase probleme generice de care dezvoltatorii se lovesc atunci când doresc să dezvolte sisteme durabile, ce pot orchestra workflow-uri de lungă durată, într-un sistem distribuit.

Dezvoltarea unui sistem de management al workflow-urilor de lungă durată într-un mediu distribuit este o problemă cu foarte multe cazuri speciale, mai ales în zona durabilității sistemului și garanțiilor de execuție pe care acesta le oferă.

Observând caracterul repetabil al acestor cazuri speciale, în piață au aparut diverse framework-uri care rezolvă în mod generic aceste probleme și expun interfețe ușor de consumat, printre care cel mai matur fiind DurableTaskFramework. Folosind această tehnologie, a fost dezvoltat un serviciu demonstrativ, ce rezolvă problema rezervării unei excursii și care expune o metodă interactivă de a vizualiza managementul unei tranzacții distribuite. Aplicația este accesibilă în mod public la <https://durable-saga-front-end.web.app/>.

Primitivele expuse de Durable Functions au înjumătățit timpul de dezvoltare a soluției și au generat un serviciu cu costuri aproape de 0, deoarece sistemul este în totalitate serverless.

Pe viitor vor fi abordate eventuale îmbunătățiri ale DTF în zonele în care au fost descoperite vulnerabilități în cadrul acestei lucrări, respectiv în adăugarea suportului pentru throttling în DTF.



# Listă de figuri

1.1	Statistică ce evidențiază importanța domeniului cloud computing în ultimii ani . . . . .	5
2.1	Orchestrare versus Coreografie . . . . .	10
2.2	DTF Schema în DTF Sql Server provider . . . . .	11
2.3	Exemplu arhitectură folosind Durable Functions . . . . .	13
2.4	Ciclu de versionare a codului de orchestrare . . . . .	14
3.1	Rezervare distribuită folosind Durable Functions . . . . .	29
3.2	Scalare automată a workerilor DTF . . . . .	30

# Bibliografie

- [1] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter. Serverless computing: Current trends and open problems. *CoRR*, abs/1706.03178, 2017.
- [2] R. Buyya, S. N. Srirama, G. Casale, R. N. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L. M. Vaquero, M. A. S. Netto, A. N. Toosi, M. A. Rodriguez, I. M. Llorente, S. D. C. di Vimercati, P. Samarati, D. S. Milojevic, C. A. Varela, R. Bahsoon, M. D. de Assunção, O. F. Rana, W. Zhou, H. Jin, W. Gentzsch, A. Y. Zomaya, and H. Shen. A manifesto for future generation cloud computing: Research directions for the next decade. *CoRR*, abs/1711.09123, 2017.
- [3] P. C. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. The server is dead, long live the server: Rise of serverless computing, overview of current state and future trends in research and industry. *CoRR*, abs/1906.02888, 2019.
- [4] H. Hassan, S. Barakat, and Q. Sarhan. Survey on serverless computing. *Journal of Cloud Computing*, 10, 07 2021.
- [5] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.
- [6] A. Megargel, C. M. Poskitt, and V. Shankararaman. Microservices orchestration vs. choreography: A decision framework. In *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 134–141, 2021.