TecnoCampus
Escola Superior
Politècnica

Centre adscrit a la
upf. Universitat Pompeu Fabra Barcelona

Grau en Enginyeria Informàtica de Gestió i Sistemes d'Informació

Anàlisis de les arquitectures Serverless

Memòria

Àlex Tello I Vidal

TUTOR: Josep Roure Alcobé

2021/22

TecnoCampus
Mataró-Maresme

Abstract

FaaS and serverless technology is an emerging cloud computing paradigm that allows to allocate the resources to develop an application to what adds value to the company. This is achieved through autoscaling and the fact that the infrastructure where the functions are executed is provided by a cloud provider. The work consists of two parts, contextualization of the technology and its use, as well as, an analysis of functionalities.On the other hand, a proof of concept where the migration of an application based on microservices to a function-based architecture is carried out.

Resum

Les FaaS i la tecnologia serverless és un paradigma de computació en el núvol emergent que permet dedicar els recursos per dur a terme una aplicació únicament a allò que aporta valor a l'empresa. Això, s'aconsegueix mitjançant l'autoescalat i el fet que la infraestructura on s'executen les funcions és proveït per un proveïdor cloud. El treball consta de dues parts una contextualització de la tecnologia i la utilització juntament amb una anàlisi de funcionalitats i, per altra banda, una prova de concepte on es realitza la migració d'una aplicació basada en microserveis a una arquitectura basada en funcions.

Resumen

Las FaaS y la tecnología serverless es un paradigma emergente de computación en la nube que permite dedicar los recursos para llevar a cabo una aplicación únicamente a aquello que aporta valor a la empresa. Esto, se consigue mediante el autoescalado y el hecho que la infraestructura donde se ejecutan las funciones es proveído por un proveedor cloud. El trabajo consta de dos partes una contextualización de la tecnología y la utilización junto con un análisis de funcionalidades y, por otro lado, una prueba de concepto donde se realiza la migración de una aplicación basada en microservicios a una arquitectura basada en funciones.

Índex

Índex de figures

Índex de taules

Glossari de termes

FaaS	Functions as a Service
CI/CD	Continous integration and continous deployment
CDN	Content delivery network
AWS	Amazon Web Services
GCP	Google Cloud Platform
SPA	Single-page application
CPD	Centre de processament de dades
LOPD	Llei orgànica de protecció de dades

1. Introducció

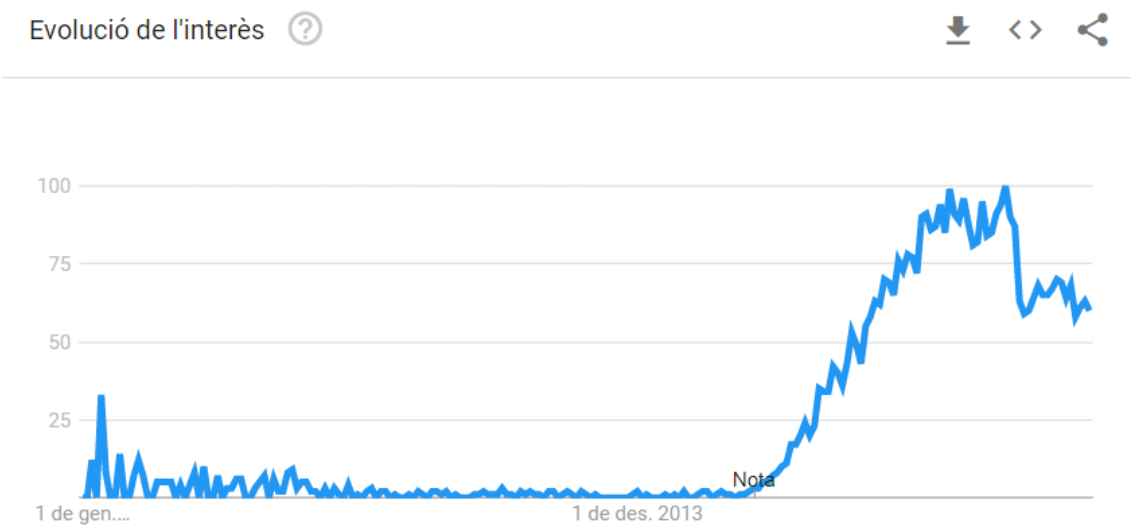
Les FaaS o funcions serverless és una tecnologia emergent que vol redefinir el paradigma de computació cloud. L'objectiu d'aquest estudi és avaluar l'adopció d'aquestes tecnologies i valorar els seus beneficis i inconvenients. Per tal d'aconseguir-ho s'utilitza una metodologia d'anàlisis de cost benefici dividida en dos blocs. El primer bloc és una recerca teòrica definint el marc tecnològic, costos operatius i les implicacions que té desenvolupar amb aquestes tecnologies.

En un segon bloc es desenvoluparà una aplicació fent servir únicament funcions FaaS i serveis serverless per tal d'avaluar la transició d'un sistema basat en microserveis a un sistema basat en events serverless. El desenvolupament de l'aplicació usant FaaS ens permet contrastar la informació teòrica obtinguda amb un cas real d'implementació.

Amb tota aquesta recerca s'assoleix una visió general de les funcions FaaS i permet avaluar un projecte amb l'objectiu d'evolucionar cap a un sistema autoescalable i distribuït.

2. Objecte del projecte

Davant el vigent increment en la computació en el núvol, Il·lustració 1 , i l'increment de la inversió en les tecnologies cloud fem un estudi dels principals avantatges i inconvenients que té aquest nou paradigma de la programació cloud, Function as a Service (FaaS). Així mateix, una investigació en profunditat dels usos i costos que té associat a aquestes aplicacions tenint en compte els tres principals agents com són Amazon Web Services, Google Cloud Platform i Microsoft Azure.



Il·lustració 1: Evolució de l'interès des de 2006

Per altra banda, realitzant una prova de concepte sobre la repercussió que té traslladar un sistema actualment en forma de monòlit o microserveis a aquest nou paradigma com és serverless functions i quins avantatges i inconvenients trobem envers la implementació anterior.

3. Objectius i abast

- Avaluar els pros i contres en l'àmbit econòmic els costos associats al manteniment de les funcions serverless.
- Exposar la disparitat de costos de les diferents plataformes cloud.
- Analitzar la velocitat i la dificultat de portar una idea o funcionalitat a producció usant funcions serverless.
- Avaluar la necessitat d'adaptació de la programació per l'ús de les funcions serverless.
- Estudiar el traspàs des d'una arquitectura basada en microserveis a una basada únicament en funcions serverless.
- Observar l'efecte que tenen les funcions serverless en la escalabilitat del codi.

4. Estudi previ

4.1. Context

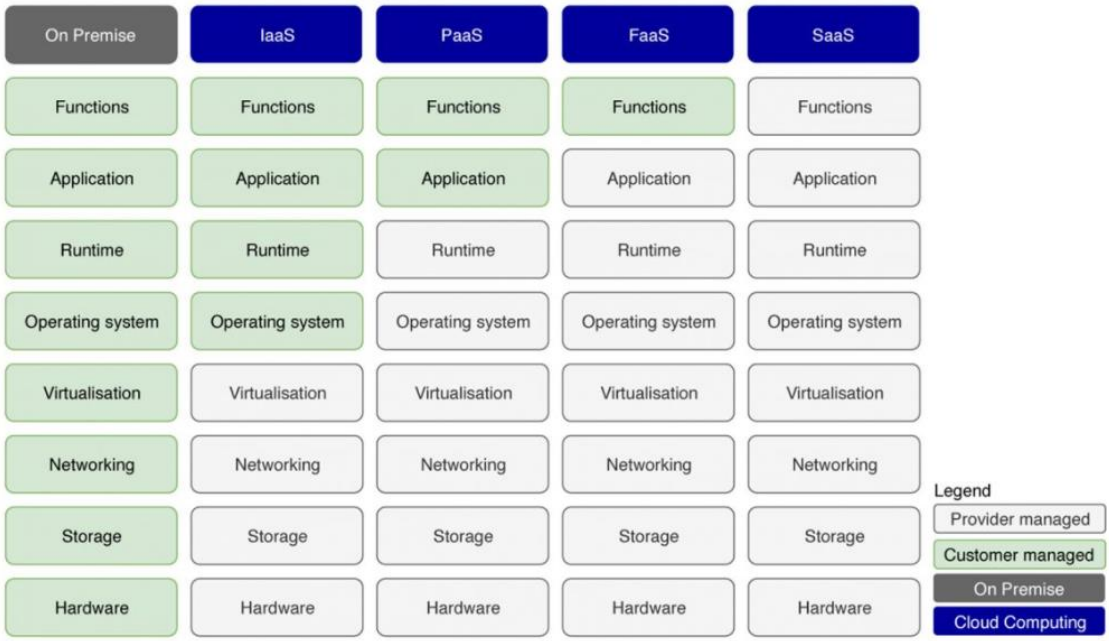
Function as a service (FaaS) és un paradigma de programació cloud que se sosté sobre dos paradigmes de la programació com és la programació orientada a events i la programació funcional.

La programació basada en events es basa en el fet que tot el codi és executat com a reacció síncrona o asíncrona

Per altra banda, la programació funcional es basa que per un valor d'entrada el resultat mai variarà.

Aquests paradigmes juntament amb la característica que les FaaS no s'han de veure afectades pel hardware on s'estan executant produeix que cada execució estigui completament aïllada.

Com es pot veure a la Il·lustració 2 quan parlem de Serverless o Function as a service (FaaS) veiem que la infraestructura per sota de la nostra lògica de negoci és proveïda pel proveïdor d'internet, per tant, únicament es controla la capa de software.



Il·lustració 2: Infraestructura proveïda pel proveïdor per tipus de contractació

En l'actualitat vivim en una situació de canvi constant i això mateix també se li demana al software, per això es tendeix cap a les solucions de software distribuïdes, ja que com explica Sam Newman en el seu llibre" Building Microservices" [1] aquestes solucions donen flexibilitat al software i permeten que s'adapti al canvi, tant cadascuna de les funcions individualment com en conjunt a causa de la seva estructura modular.
Un altre fet rellevant de les FaaS que fa que s'inverteixi en aquesta tecnologia és que les empreses busquen controlar els costos. Això s'aconsegueix mitjançant el millor dimensionament dels recursos de computació que es disposen. Ja que únicament es paga per allò que s'utilitza.

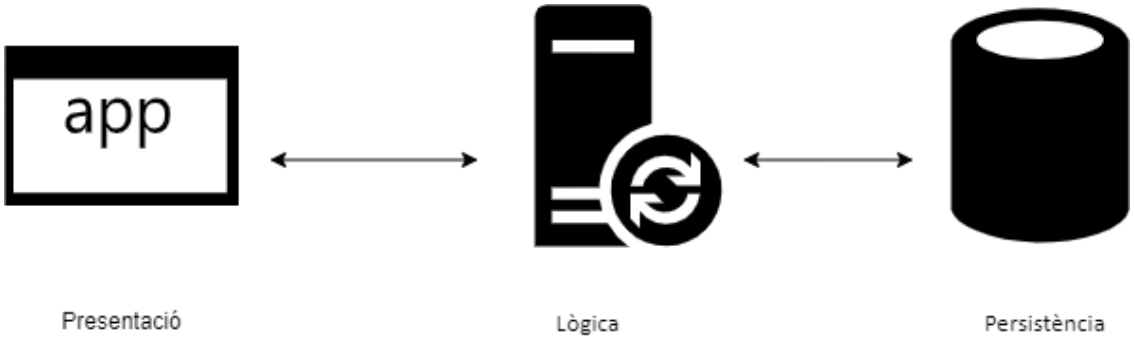
Aquest punt té una afectació en l'àmbit energètic i mediambiental molt important, pel fet que es disminueix l'impacte energètic que té l'empresa pel fet d'aprofitar al màxim els seus recursos.

A més a més la infraestructura que està executant el codi no és tan rellevant la importància real és allò que aporta valor, el seu domini i regles de negoci que formen cadascuna de les funcions.

Per altra banda, l'increment d'utilització de les tecnologies cloud posa sobre la taula una preocupació en l'àmbit d'impacte en el planeta on el consum d'aquests centres de processament de dades i l'eficiència d'aquests és tan important per disminuir l'impacte sobre la petjada de carboni i el consum energètic que tenen aquestes granges de servidors on s'executarà les nostres funcions. Ja que aquestes tenen problemes de dimensionament, on una part dels seus recursos estan sent infrautilitzat, implicant que la infraestructura està consumint recursos que no produeixen un treball en conseqüència baixant l'eficiència dels centres de processament de dades i produint un malbaratament dels recursos energètics. [2]

Amb això en ment ens centrarem en la utilitat de les funcions serverless en les aplicacions web o cloud i en les arquitectures basades en events, per tant, comunicació asíncrona entre els diferents serveis.

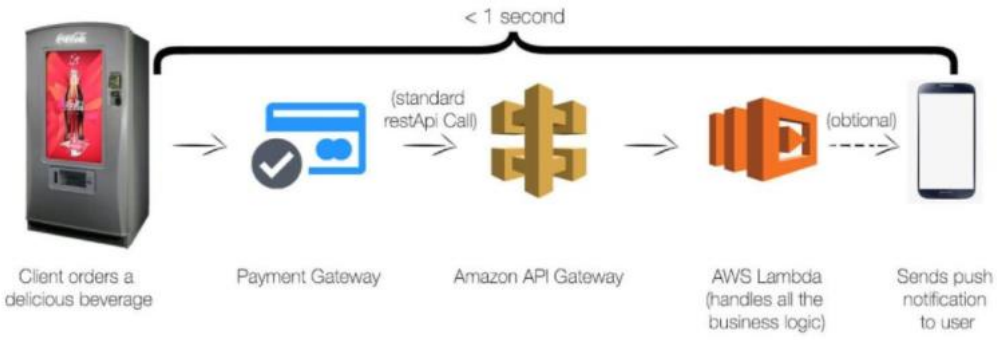
Un exemple de FaaS seria una arquitectura basada en 3 capes, Il·lustració 3, la capa de presentació on és la part més visual i interactiva com potser una pàgina web o un formulari , la capa de persistència que és on s'emmagatzema les dades, en aquest cas trobem dues opcions les serverless que són bases de dades on l'usuari únicament manega l'esquema de dades que vol guardar i les dades i les no serverless que són infraestructura de base de dades on tens el control total del motor de la base de dades. I per últim la capa de la lògica on tenim totes les regles de negoci i és aquí on les funcions serverless són més utilitzades.



Il·lustració 3: Representació de l'estructura 3 capes.

La utilització de les funcions serverless cada cop està més estesa arreu. Un exemple de cas d'èxit d'una companyia que ha fet la transició a utilitzar funcions serverless és Coca-Cola [3]. El cas Coca-Cola es basa en la necessitat que té Coca-Cola de saber quantes llaunes s'han venut a les diferents màquines expenedores per tal de reposar-les.

Malgrat que la tasca és senzilla, Il·lustració 4, simplement s'ha de notificar cada cop que es ven una llauna i sabent la capacitat d'emmagatzemar que té la màquina sabem quan hem d'anar a reposar la màquina. Fins al 2016 estaven fent servir una mitja de sis màquines virtuals amb dues CPU i quatre GB de ram que els suposava uns $13000 a l'any entre què pagaven per les màquines virtuals i manegar tota la infraestructura IT, balancejadors de càrrega, seguretat, actualitzacions, etc.



Il·lustració 4: Funcionament de la infraestructura de Coca-Cola

Coca-Cola va revaluar el nombre de peticions que es realitzaven al llarg de l'any, que eren al voltant de 30 milions i van calcular que la infraestructura que estaven utilitzant actualment estava quantificada per aguantar els pics, però estava sent infrautilitzada, per tant, van decidir fer la inversió en investigar la mateixa solució, però aquest cop serverless on fan que la màquina realitzi una petició a una API i aquesta petició executa una funció serverless que implementa la lògica de negoci que necessiten.

Aquesta transformació permet a Coca-Cola avui en dia dimensionar la infraestructura dinàmicament en funció de les peticions que tenen i a la mateixa manera únicament dediquen els seus recursos al que és important pel seu negoci.

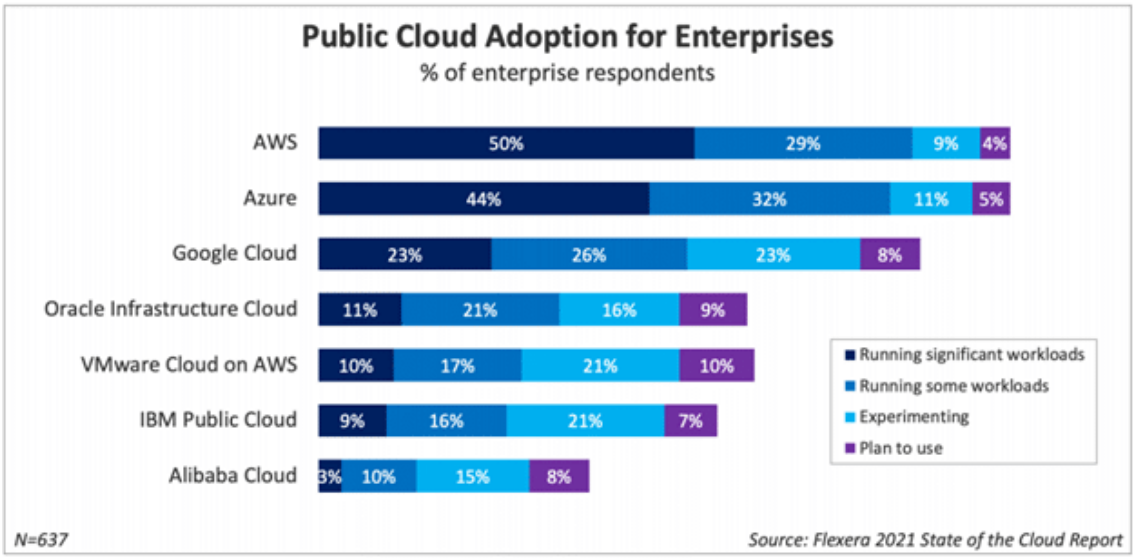
Aquest canvi fa que el cost de la infraestructura de Coca-Cola disminueix de $13000 fins a $4500 per any.

Un altre cas d'ús on les funcions serverless són molt rellevants són aquelles tasques que computacionalment són molt complexes, requereixen molts recursos i a la vegada necessiten tenir un temps d'execució molt baix, ja que l'usuari està esperant la resposta. Aquí és on les funcions serverless tenen l'adopció més gran per part de les empreses. Un referent a nivell d'arquitectures cloud es Netflix sent pioner en infraestructures de microserveis i arquitectures distribuïdes.

En aquest cas es troben que sobre la seva infraestructura de microserveis que té Netflix hi ha parts que no s'executen tan sovint, però que quan ho fan han de donar una resposta immediata independentment de les vegades que es cridi aquesta funció simultàniament i per fer això possible creen la plataforma Cosmos [4].
Cosmos és una plataforma que fa conviure els microserveis que té Netflix dins la seva infraestructura IT amb les funcions serverless, aquesta plataforma té com a objectiu orquestrar la utilització de les funcions serverless dins d'una execució en concret. Per exemple un usuari vol veure una pel·lícula que Netflix no la té preparada per enviar, aquí és on Cosmos entra en acció, cridant aquestes funcions serverless que tenen molt de poder de computació per tal de preparar el vídeo per ser enviat fent que el temps de resposta sigui mínim.

4.2. Principals proveïdors

Seguint l'anàlisi Il·lustració 5 sobre la utilització de les plataformes cloud de les empreses farem una anàlisi de l'oferta dels productes FaaS.



Il·lustració 5: Adopció dels principals proveïdors Cloud

4.2.1. Amazon Web Services

Amazon Web Services (AWS) va ser el primer dels grans clouds en oferir la tecnologia FaaS al gran públic en el tretze de novembre de 2014 va anunciar el seu producte AWS Lambda [5].

AWS ofereix dos productes el primer AWS Lambda que permet l'execució tant de funcions com contenidors sense mantenir l'estat de l'aplicació entre execucions. Té un límit de 15 minuts d'execució [6].

Per altra banda, existeix un altre producte AWS Fargate ofereix únicament l'execució de contenidors accepta mantenir l'estat i ser executades contínuament sense límit del temps d'execució.

4.2.1.1. Costos de execució

4.2.1.1.1. AWS Lambda

Taula 1: Tarifa AWS Lambda

Arquitectura	Duració	Requests
X86	0.0000166667 $/GB per segon	0.00000020 $ per petició
ARM	0.0000133334 $/GB per segon	0.00000020 $ per petició

4.2.1.1.2. AWS Fargate

Taula 2: Tarifa AWS Fargate

Arquitectura	Duració	Peticions
X86	0.00001293 $/vCPU per segon 0.0000014194 $/GB per segon	-
ARM	0.0000014194 $/vCPU per segon 0.0000011361 $/GB per segon	-

4.2.2. Microsoft Azure

Microsoft dins la seva plataforma cloud anomenada Azure ofereix l'execució tant de funcions com contenidors sense preservar l'estat amb un límit d'execució de cinc minuts per defecte fins a un màxim de 10 minuts configurable.

4.2.2.1. Costos d'execució

4.2.2.1.1. Microsoft Azure Functions

Taula 3: Tarifa Microsoft Azure Functions

Arquitectura	Duració	Peticions
X86	0,000016 $/GB per segon	0.00000020 $ per petició

4.2.3. Google Cloud Computing

Google cloud computing (GCC) també ofereix dos productes el primer de tots es Google Cloud Functions que com el mateix nom indica únicament permet executar funcions i ha de ser en una sèrie de llenguatges de programació en concret , Node.js, Python, Go, Java, Net, Ruby i PHP [7].

Per altra banda, hi ha un producte molt similar a AWS Fargate que permet executar els contenidors i les funcions sense límit de temps.

4.2.3.1. Costos d'execució

4.2.3.1.1. Google Cloud Functions

Taula 4: Tarifa Google Cloud Functions

Arquitectura	Duració	Peticions
X86	0.000024 $/vCPU per segon 0.0000025 $/GB per segon	0.00000020 $ per petició

4.2.3.1.2. Google Cloud Run

Taula 5: Tarifa Google Cloud Run

Arquitectura	Duració	Peticions
X86	0.000024 $/vCPU per segon 0.0000025 $/GB per segon	0.00000040 $ per petició

4.2.4. Alternativa de codi lliure

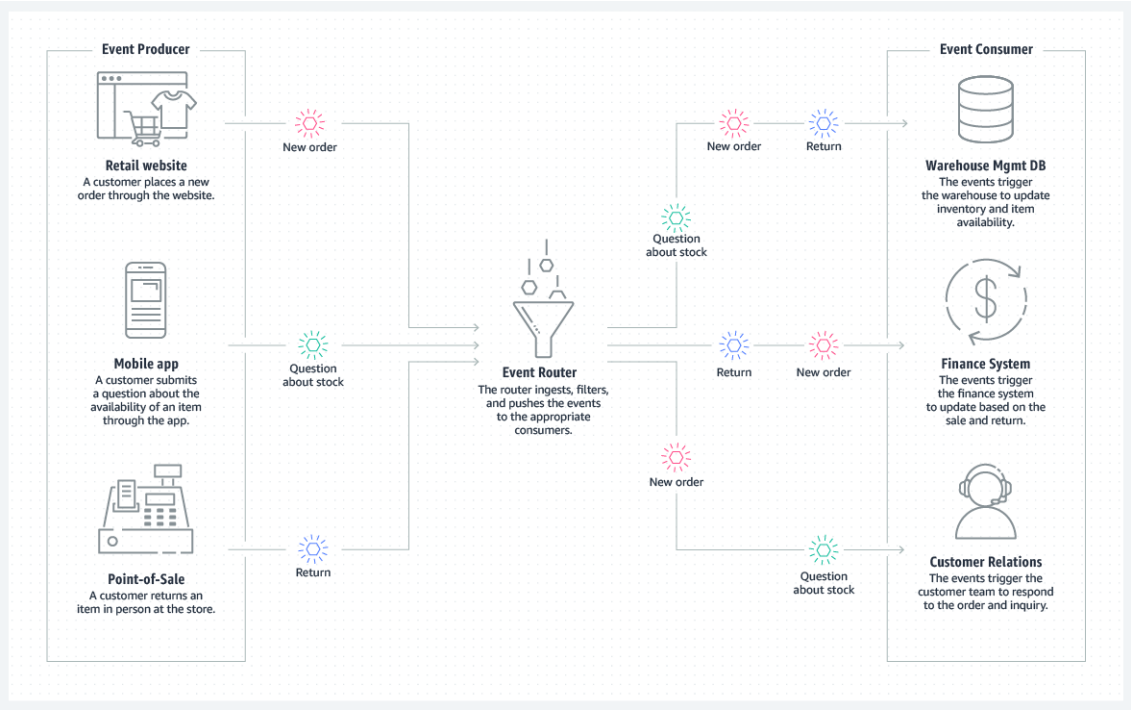
També hi ha alternatives de codi lliure que ens permeten tenir la nostra infraestructura serverless i poder executar-la on vulguem sigui un cloud públic o un cloud privat mantenint el codi completament deslligat del proveïdor que ens ofereix el servei [8]. Hi ha diferents opcions Taula 6.

Taula 6: Popularitat alternatives de codi lliure

Projecte	Col·laboradors	Estrelles
OpenFaaS	161	21.2k
Apache Open Whisk	195	5.6k
Knative	384	3.3k
Kubeless	105	6.8k

4.3. Arquitectura basada en events

La utilització de les FaaS promou una adopció de les arquitectures basada en events [9] on les funcions reaccionen als events permetent l'aplicació escalar horitzontalment.



Il·lustració 6: Exemple d'arquitectura basada en events

L'arquitectura està dividida en dos grans blocs com podem veure en Il·lustració 6.

4.3.1. Productors d'events

Els productors d'events són les funcions encarregades de generar els events que notifiquen que ha succeït un esdeveniment rellevant en el cas de les FaaS pot ser un nou article a la base de dades, un formulari que ha estat emplenat o un accés a un URL.

4.3.2. Consumidors d'events

Per altra banda, les funcions consumidores d'events són les que reben la notificació que s'ha realitzat una acció i reaccionen executant el codi, fent la funció designada com per exemple pot ser actualitzar el balanç del compte de l'usuari o enviar la foto que estan accedint amb la petició.

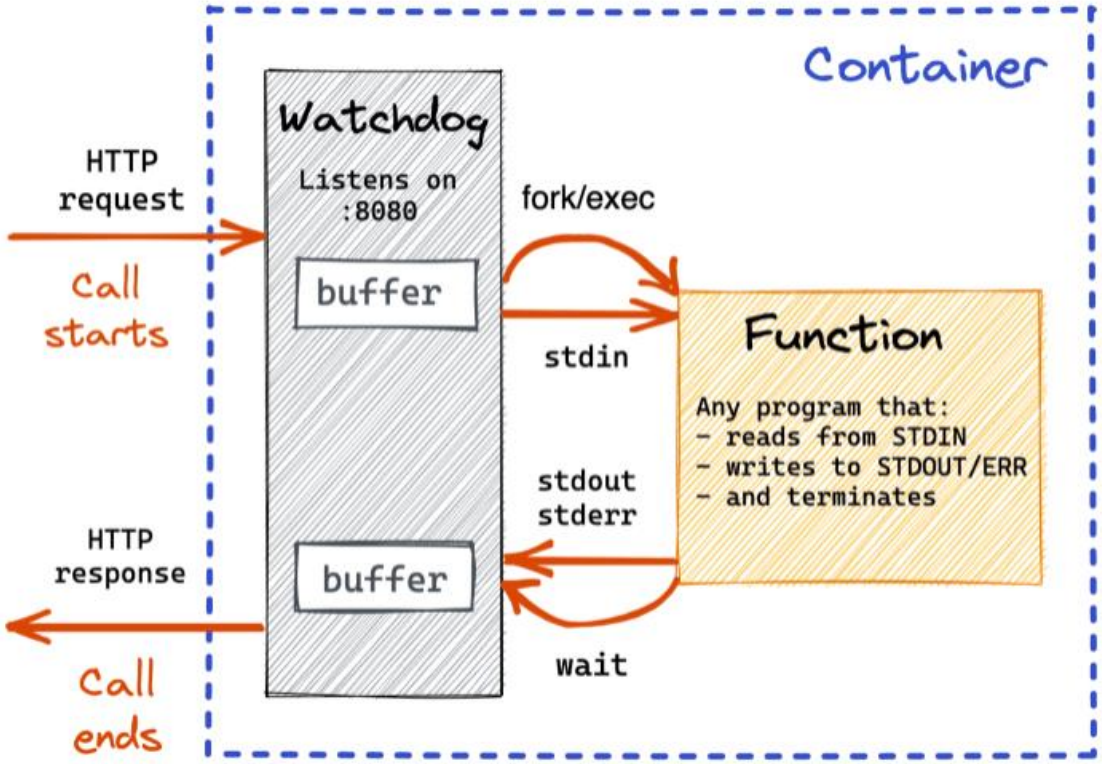
4.4. Arquitectura de les funcions

Perquè les funcions siguin serverless i puguin reaccionar als events, ja siguin síncrons com seria una petició HTTP a un endpoint d'una API Gateway o asíncrons com serien els missatges als quals reacciona una funció quan s'ha pujat un fitxer nou al repositori, han de complir una estructura concreta que està dividida en dos blocs.

4.4.1. Handler

El Handler és la part de la funció encarregada de gestionar l'esdeveniment que se li subministra a l'acció. Depenent de la plataforma on despleguem aquest forma part de les llibreries o SDK que ofereixen els proveïdors cloud, en el cas d'AWS utilitza una llibreria específica de la plataforma que gestiona la informació que se li subministra a l'acció.

En el cas d'Open FaaS, per altra banda, extreu aquesta complexitat a una altra capa Il·lustració 7 on com podem llegir en el cas d'estudi [10] on el whatchdog executa un servei similar a un web server en aquest cas que intercepta les peticions i traspassa a la funció per arguments la informació d'entrada y espera a rebre la informació de sortida responent a la petició HTTP. Això permet que un mateix watchdog executi diverses funcions simultàniament.



Il·lustració 7: Representació de watchdog

4.4.2. Regles de negoci

En aquesta part de la funció és on realment existeix el nostre domini i les regles de negoci que produiran que la nostra aplicació faci allò per què està dissenyada.

Però s'han de tenir diferents factors en compte amb les funcions amb el fi de rebre sempre els resultats que esperem i és que han de ser idempotents, independentment de quantes vegades s'executi la mateixa funció amb els mateixos paràmetres i ordre ha de finalitzar amb el mateix resultat. També han de ser stateless, és a dir, no poden dependre de variables globals que puguin produir efectes secundaris en l'execució de la funció. I per últim no han de quedar bloquejades, no es pot gestionar la concurrència amb bloquejos, ja que com hem vist en seccions anteriors en molts proveïdors hi ha un límit de temps que la funció s'està executant i això fa que un sistema de bloqueig pugui produir efectes inesperats.

4.5. Events

Una peça clau de les arquitectures basades en events són els events que a diferència de les funcions on no hi ha un estàndard clar la Cloud Native Fundation ha creat un estàndard mitjançant el projecte de Cloud event [11]. Això permet que hi hagi un consens en el tractament dels events universal. Aquesta especificació ja ha estat adoptada per la gran majoria dels proveïdors cloud. Dos dels principals proveïdors cloud com són Google Cloud amb el producte Eventarc [12] o Azure que implementa l'especificació dels events JSON [13] i els events HTTP [14].

El fet que hi hagi un estàndard significa que hi ha un SDK global que permet que els llenguatges de programació interactuïn amb aquests events d'una manera controlada generant o capturant-los.

Una de les principals característiques dels events és que han de ser llegits o generats per infinitat de funcions diferents en llenguatges diferents per això estan pensats per ser serialitzats a JSON.

Seguint l'estàndard un exemple de de event seria:

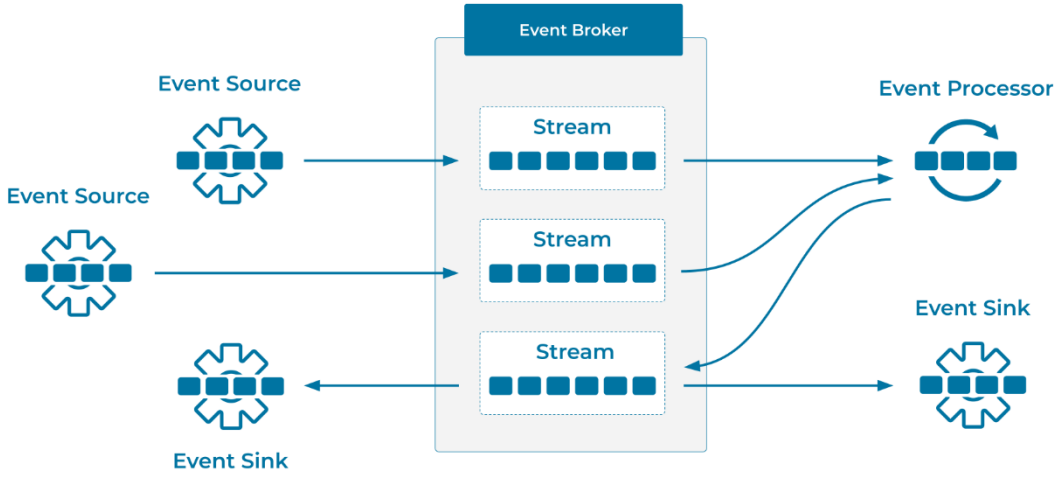
```
{
    "specversion" : "1.0",
    "type" : "com.github.pull_request.opened",
    "source" : "https://github.com/cloudevents/spec/pull",
    "subject" : "123",
    "id" : "A234-1234-1234",
    "time" : "2018-04-05T17:31:00Z",
    "comexampleextension1" : "value",
    "comexampleothervalue" : 5,
    "datacontenttype" : "text/xml",
    "data" : "<much wow=\"xml\"/>"
}
```

Il·lustració 8: Exemple de event

On podem veure un event Il·lustració 8 generat per la creació d'una pull request al github com posa en el camp type, el source que ha generat el event, el moment de creació i les dades i el tipus del contingut del event.

4.6. Event Broker

Hi ha un agent opcional en les arquitectures basades en events i és el event broker, aquest agent permet la comunicació asíncrona entre les diferents funcions així com encaminar els diferents events a les funcions que els consumeixen permetent que les diferents funcions estiguin desacoblades entre si seguint el patró del message broker central [15] .



Il·lustració 9: Event broker

Com podem veure en la Il·lustració 9 els productors generen els events i se'ls envia al event broker qui llavors emmagatzema els missatges fins que el processador de events els demana o en el cas que no sigui asíncron li envia directament el missatge.

Els diferents proveïdors cloud ofereixen diferents implementacions dels event brokers. En AWS existeix EventBridge [16] , en Azure comercialitzen Event Grid [17]. També existeixen productes que ofereixen aquesta funcionalitat com serien Apache Kafka i RabbitMQ que poden ser desplegats en qualsevol plataforma per tal de tenir una solució més universal.

5. Metodologia

La metodologia d'aquest treball serà una metodologia d'anàlisis de cost benefici [18] que constarà de dos blocs, el primer bloc és una recerca en profunditat teòrica definint el marc tecnològic actual, els costos operatius de la tecnologia, les implicacions de les FaaS tenen sobre les pipelines de CI/CD, el testing i el manteniment del codi.

Que constarà de dos blocs, el primer bloc és una recerca en profunditat teòrica definint el marc tecnològic actual, els costos operatius de la tecnologia, les implicacions de les FaaS tenen sobre les pipelines de CI/CD, el testing i el manteniment del codi.

Conjuntament, farem una anàlisi dels riscos que suposa desenvolupar una FaaS des del punt de vista del vendor lock que és quan desenvolupes una aplicació que únicament pot funcionar en aquella plataforma cloud.

El segon bloc serà un cas pràctic, una transició des d'una arquitectura de microserveis cap a una arquitectura totalment serverless utilitzant el cas d'estudi descrit en el llibre [19] Aquest bloc ens permetrà poder extreure conclusions de la dificultat que té configurar tot l'entorn de desenvolupament, eines que es fan servir, frameworks ...

En conjunt ens permetrà poder fer una comparativa entre les arquitectures de microserveis i serverless podent així analitzar els pros i contres per definir quina beneficia més el nostre negoci a l'hora de perdre una decisió fonamentada.

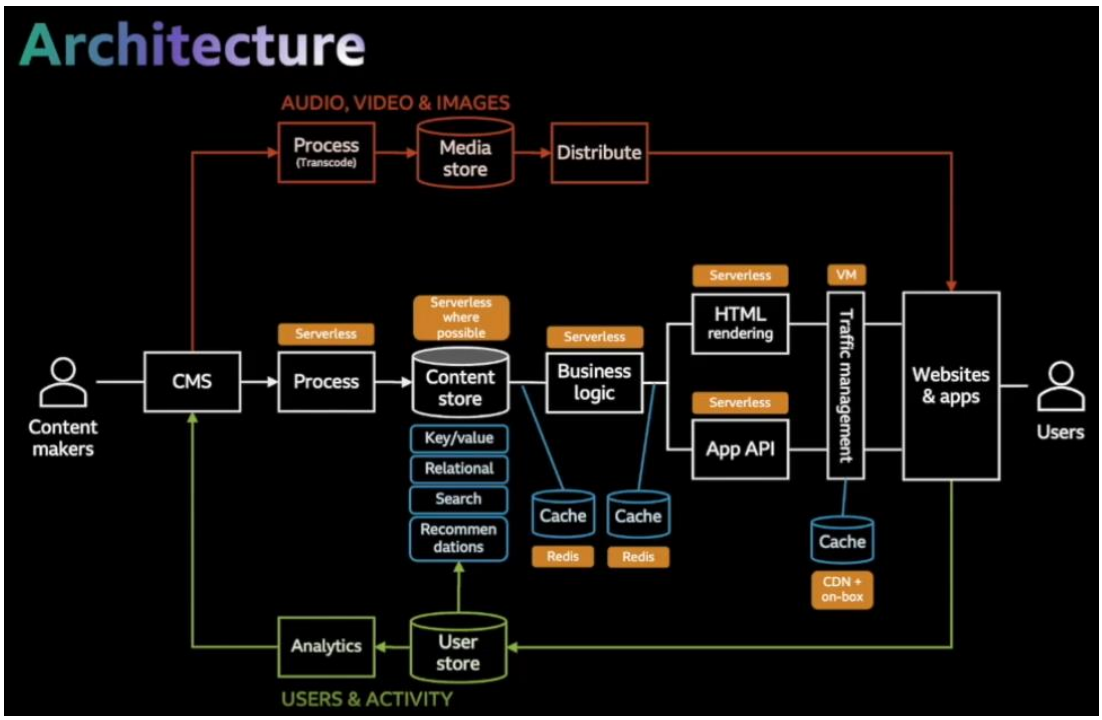
6. Anàlisis de casos reals

6.1. BBC Online

El cas d'estudi de la BBC online representa la transició d'una arquitectura plantejada el 2010 [20] basada en dos grans centres de dades a prop de Londres hostejant i servint les més de 200 pàgines en 44 idiomes diferents que componen la presència digital de la BBC.

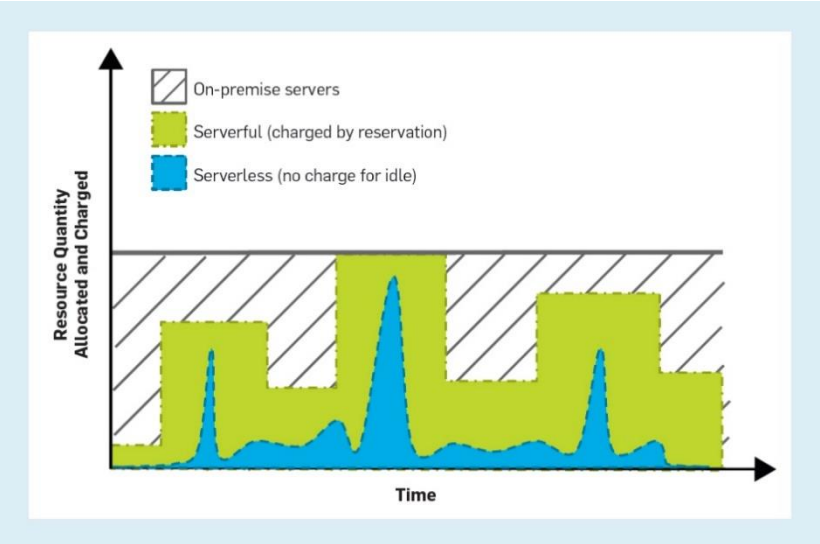
Els objectius d'aquesta transició eren [21] primer de tot ser capaç d'escalar a un gran tràfic, a causa de la gran afluència de públic que tenen en l'actualitat i la previsió de creixement on a més a més el canvi dels continguts a un sistema menys de premsa escrita i més de contingut audiovisual requereix una infraestructura diferent. Aquest canvi en el contingut que ofereix la BBC fa que un dels requeriments sigui poder servir un ventall de continguts molt més extens i creat per diferents equips. L'últim dels propòsits és l'eficiència en costos, però no només en la factura de la solució cloud sinó tenint en compte que el cost més gran per la majoria d'empreses és el cost humà, la infraestructura ha de ser fàcil de mantenir i implementar.

Aquests punts van fer que la BBC es decantés per una arquitectura Il·lustració 16 basada en gran part en FaaS evitant manteniment i disminuint la complexitat de la seva infraestructura marcant estàndards però permetent afegir millores o processos paral·lels amb facilitat.



Il·lustració 10: Esquema de l'arquitectura cloud de la BBC Online

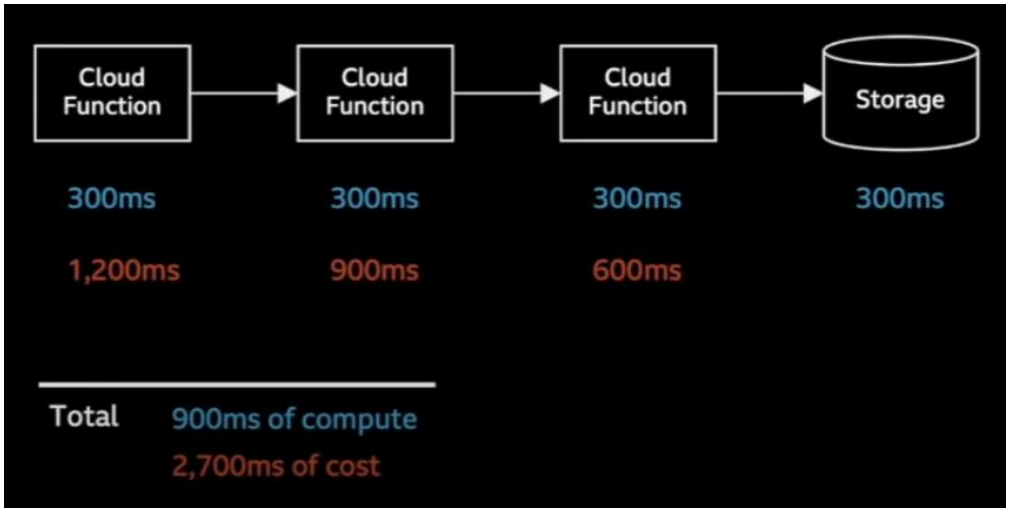
Quan la BBC feia l'anàlisi de costos de la nova solució, comparaven el cost de tres possibles solucions, els servidors On-premise que tenien fins ara, envers una infraestructura hostejada en contenidors o màquines virtuals i serverless. Avaluant les dades van treure les conclusions que malgrat que el cost de la funció és entre 3 i 5 cops el preu d'una infraestructura basada en contenidors o màquines virtuals l'escalat era lineal a la demanda Il·lustració 11, ja que, per altra banda, hi ha un interval de recursos que es perden implicant un cost superior en el gran volum.



Il·lustració 11: Recursos reservats envers als utilitzats

No tot són avantatges l'ús de les funcions serverless hi ha coses a tenir present, una d'elles es que la facturació de les FaaS és per recursos envers temps d'execució independentment que la funció estigui o no bloquejada per una altra funció.

En el cas de tenir una cadena bloquejant com aquesta Il·lustració 12 els desenvolupadors de la BBC van implementar sistemes de cache disminuint al mínim el bloqueig produït per sistemes més lents com poden ser les bases de dades relacionals o inclús reduint peticions al backend mitjançant sistemes CDN.



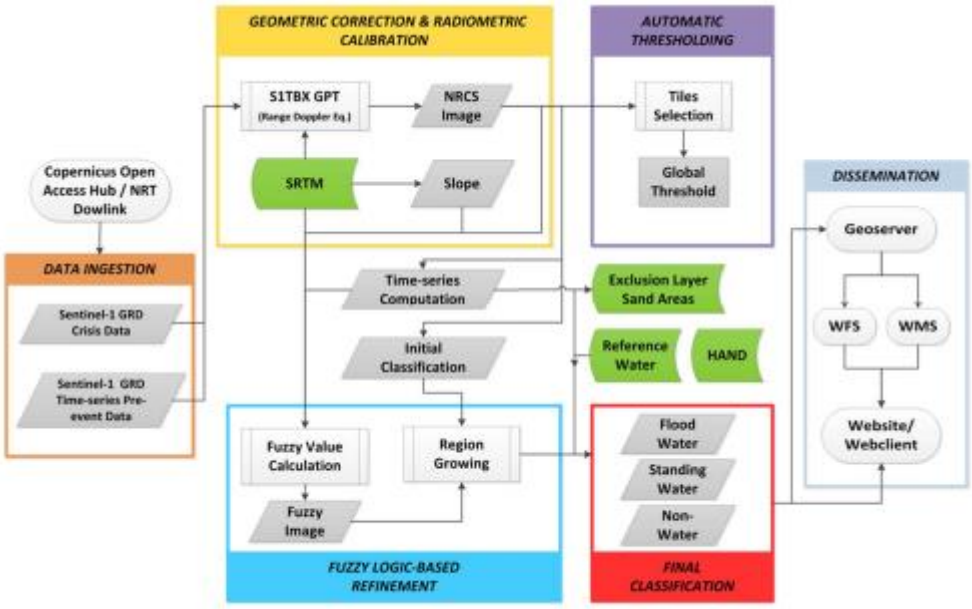
Il·lustració 12: Cadena de FaaS

6.2. Anàlisi científica del moviment hidrogràfic

El següent cas d'estudi es tracta d'un sistema d'anàlisis científic dels moviments hidrogràfics processant dades que provenen de la constel·lació de satèl·lits Sentinel-1 que forma part del programa Copèrnic de la Unió Europea. Aquest programa [22] té com a objectiu aportar dades gratuïtes i totalment obertes captades a partir de la col·laboració entre les diferents agències amb l'objectiu que les administracions públiques i d'altres serveis puguin millorar la qualitat de vida de la ciutadania Europea.

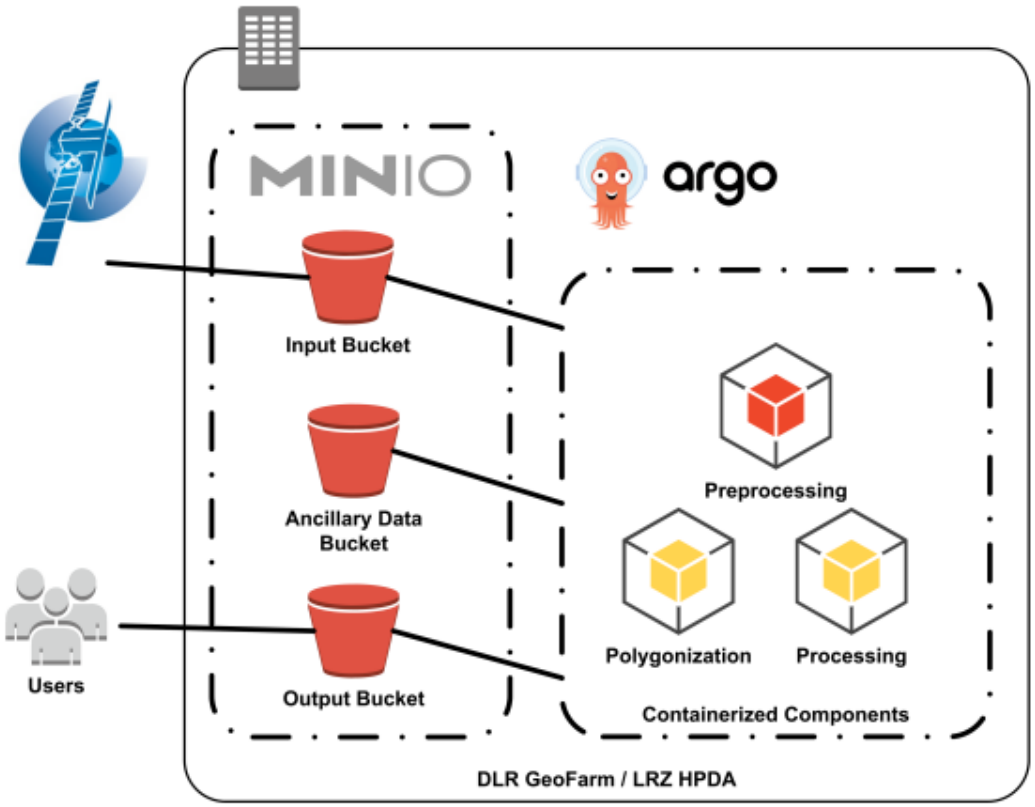
Aquestes dades estan formades per aproximadament 1200 escenes per dia que creen una ingesta d'informació de l'ordre de 800 GB. Per tant, per tal de produir resultats que es puguin consumir d'una forma senzilla es requereix diverses fases de tractament i anàlisis de la informació [23].

Com indica el següent diagrama Il·lustració 13 descrit en l'article [24] ens trobem un sistema d'anàlisis i classificació de les imatges generades amb la finalitat de prevenir inundacions alertant.



Il·lustració 13: Fases del tractament de les dades de Sentinel-1

Per tal d'assolir els requeriments es planteja una arquitectura basada en un programari d'orquestració de contenidors de codi lliure anomenat Argo Workflows [25] que amb l'ajut d'un sistema de fitxers de codi lliure i compatible amb les funcions com és Minio [26] confeccionar la següent arquitectura:



Il·lustració 14: Arquitectura de tractament de les dades Sentinel-1

L'objectiu d'aquesta arquitectura Il·lustració 14, a diferència de la del cas anterior, consta de tractar un gran volum de dades paralitzant el procediment i permetent aprofitar al màxim els recursos computacionals dels quals disposa la infraestructura.

7. Característiques de les FaaS

Utilitzant la informació adquirida en l'anàlisi dels casos d'èxit i contrastant-la amb articles com "Arquitectures serverless" de Mike Roberts [27] analitzem les característiques de les FaaS

7.1. Estat de la Funció

Una de les principals restriccions de les FaaS és el fet que entre execucions poden no mantenir l'estat, és a dir, les variables, fitxers o altres configuracions no estan garantits que estiguin encara en la funció. Això implica que qualsevol dada que necessiti ser reutilitzada o persistida ha de ser emmagatzemada fora de la funció. Sigui una base de dades o un sistema centralitzat de logs per exemple.

La millor manera de programar aquestes funcions és utilitzar com el mateix nom indica programació funcional i una guia a seguir és "The Twelve Factor App" [28].

7.2. Duració de la funció

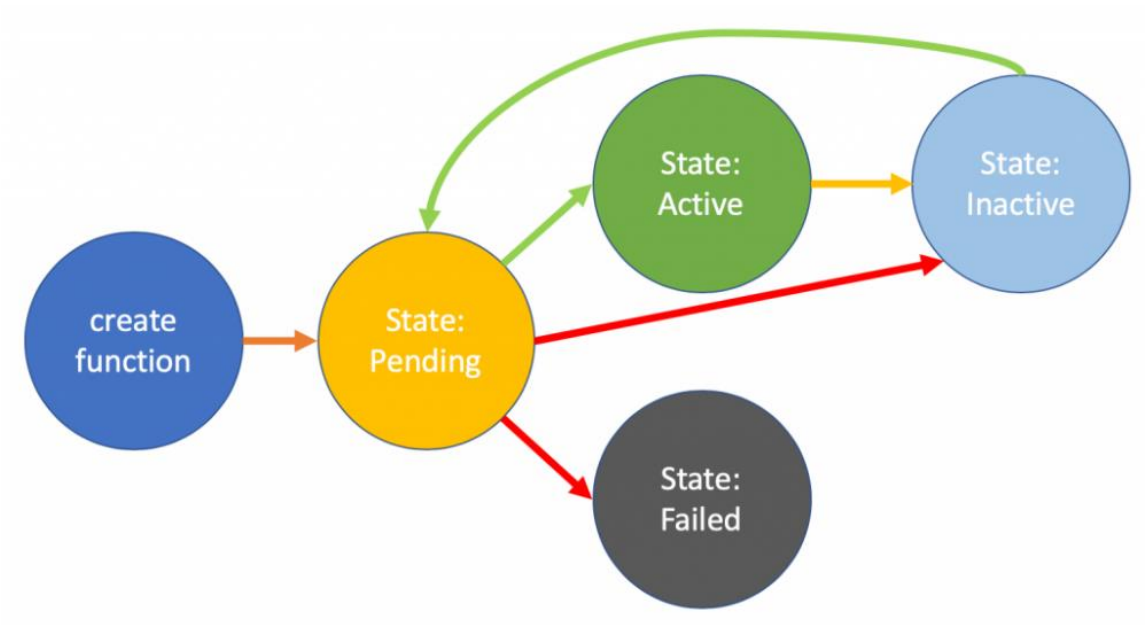
Com ja hem vist en l'apartat d'anàlisis dels principals proveïdors de funcions ens trobem que la gran majoria limiten el temps màxim d'execució d'una funció. Aquesta restricció provoca que els processos que es passin d'aquests temps d'execució hagin de ser dividits en fases. Com seria l'exemple de l'anàlisi de dades del satèl·lit Sentinel-I on cada fase guarda el resultat en un emmagatzemament extern. Això afegeix complexitat, però té avantatges pel que fa a l'aplicació està més segmentada en funcionalitats concretes.

7.3. Latència

Quan avaluem el potencial que tenen les FaaS en entorns productius ens trobem una inquietud envers el temps de resposta que tenen les funcions.

Per entendre la latència primer hem de conèixer el cicle de vida que tenen les funcions.

7.3.1. Cicle de vida de les funcions



Il·lustració 15: Cicle de vida de les funcions

Com podem veure a la Il·lustració 15 el cicle de vida de les funcions es basa en 4 estats. En el moment que es demana una funció automàticament adquireix l'estat de pendent, en aquest període la funció està aixecant-se, descarregant les dependències necessàries i adquirint connexió a aquells recursos que necessita. Des d'aquest punt poden passar tres coses, primer de tot que algun d'aquests recursos no es puguin carregar i la funció falla. La segona opció és que la funció s'ha aixecat correctament, però l'event es cancel·la al cap d'uns minuts passarà de pendent a inactiva. I per últim el que és el cicle normal, un cop inicialitzada la funció respon a un dels events passant a estar activa i processant l'event, un cop acabat l'event passa a estat inactiu on només tornarà a estar pendent en el cas que hi hagi una nova petició, en el cas que no hi hagi cap aquesta funció serà destruïda.

7.3.2. Cold Start

Com hem pogut veure prèviament hi ha un punt crític en aquest cicle de vida que és quan s'han de crear les funcions això és conegut com a cold start i aquest procés pot durar entre 50 ms fins a cinc segons fent que en processos on el temps de resposta és crític aquesta limitació pot implicar que no es pugui utilitzar aquesta tecnologia.

Hi ha diversos factors que ajuden a disminuir aquests temps d'inicialització, disminuir el nombre de dependències que tenim en el codi, fer servir llenguatges de programació que inicialitzin el més ràpid possible i optimitzar-ho per inicialitzar i carregar les variables d'entorn el més eficient possible. I limitar l'accés que ha de tenir la funció a recursos externs ja siguin bases de dades com sistemes de fitxers.

7.4. Tecnologia

Les FaaS estan molt lligades a ser utilitzades en clouds públics, això també implica que aquests desenvolupaments estaran lligats a moltes eines que aquests proveïdors subministren, com pot ser el sistema gestor d'events, les bases de dades propietàries en les quals es guardaran les dades o els sistemes de fitxers per emmagatzemar arxius. Aquesta dependència crea un altre problema de les FaaS que s'anomena vendor Lock.

7.4.1. Vendor Lock

El vendor lock o la dependència d'un proveïdor cloud és deguda a la falta d'especificacions obertes i la falta de consens entre els diferents proveïdors i les comunitats per definir un estàndard. Això provoca que en la majoria de casos desenvolupes una aplicació amb les tecnologies oferides per proveïdor i adaptant les relacions a les imposades.

8. Avantatges i inconvenients que ofereixen les funcions

8.1. Avantatges

8.1.1. Reducció dels costos de manteniment de la infraestructura

Les FaaS majoritàriament estan hostejades en servidors dels proveïdors cloud aprovisionades i actualitzades per ells, per tant, des d'un punt de vista de manteniment de la infraestructura, s'està subcontractant aquest manteniment.

8.1.2. Reducció del cost d'escalat

Com bé mostra el cas d'èxit de la BBC malgrat que les funcions serverless tenen un cost individual més gran al que tindria el mateix volum de computació adquirit per exemple en una màquina virtual, l'escalat pot ser molt més ajustat al creixement real que té la nostra aplicació, per tant, evita un sobredimensionament de la infraestructura fent que estalviem costos.

8.1.3. Disminució del temps de desenvolupament

Una de les principals característiques per les quals moltes empreses emprenedores utilitzen el model serverless més enllà de què els costos incrementen linealment amb la quantitat de servei que ells estan aportant és poder portar una idea a mercat en un curt període de temps, ja que no han de configurar cap plataforma i a més a més poden aprofitar moltes funcions ja fetes arreu i centrar-se únicament en allò que aporta valor.

8.1.4. Sostenibilitat

Les funcions Serverless no només tenen impacte en temes de desenvolupament i econòmics també impacten mediambientalment, perquè segons la revista Forbes [29] un 30% dels servidors estan sent infrautilitzat o simplement parats. I aquí és on les funcions serverless tenen un punt a favor, pel fet que gràcies al seu cicle de vida permet alliberar un recurs que d'una altra manera estaria reservat i sense utilització.

8.2. Inconvenients

8.2.1. Estar restringit a un proveïdor

En utilitzar tecnologies FaaS la nostra infraestructura s'adapta a la del proveïdor en qüestió, fent que tant el codi com el com les nostres eines de desenvolupament estiguin lligades a ell. Això produeix que la barrera de cara a migrar entre proveïdors sigui gran.

8.2.2. Fer debug i captar logs

Tenint en compte que el sistema operatiu on les nostres funcions s'executen fer servir sistemes de captació d'errors i logs com a serveis en background no és possible. Per tant, es depèn de la integració que faci el proveïdor que estem fent servir.

Aquesta aproximació dificulta molt la feina de saber que està passant a l'aplicació i el perquè d'una fallida. A més a més estem parlant d'una arquitectura distribuïda que ja de per si són complicades de testejar, però a això se li ha d'afegir el fet que les funcions reaccionen a events que es generen en el cloud i que no són fàcils de replicar localment per tal de poder desenvolupar.

8.2.3. Seguretat

A diferència de les arquitectures monolítiques les funcions estan repartides en diferents servidors inclús diferents centres de dades, exposades en gran manera a internet provocant una superfície d'atac major i a més a més no tenim control sobre la infraestructura, per tant, no podem utilitzar sistemes d'anàlisi de comportament o protecció de la infraestructura.

D'altra banda, la nostra aplicació està distribuïda en diferents funcions fent que a mesura que creix l'aplicació sigui més complex comprovar que la infraestructura és segura i que no hi ha possibles vulnerabilitats en les aplicacions que la comprometen.

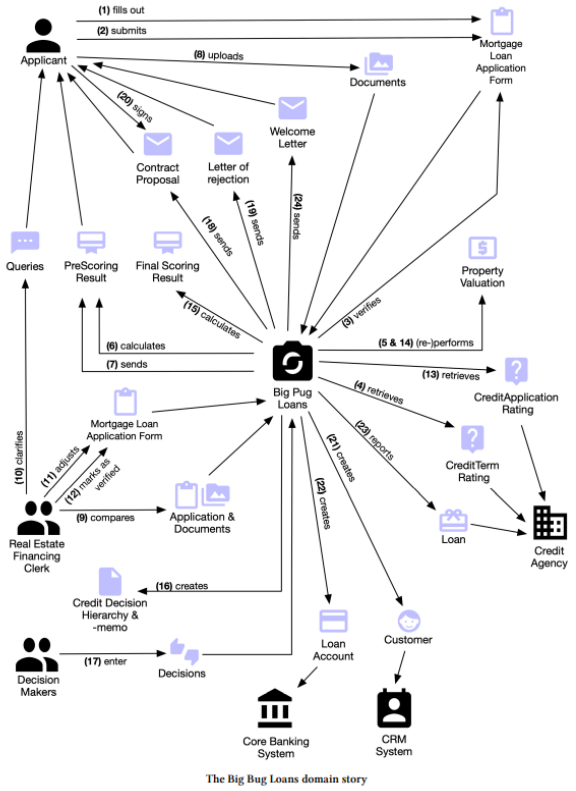
.

9. Definició de requisits funcionals i tecnològics

Aquest projecte és de caire teòric per tal de fer una anàlisi dels beneficis que aporta aquesta tecnologia, les funcions serverless, malgrat això hi ha una part que consta en transformar una aplicació real basada en microserveis a una aplicació que únicament utilitza serverless.

Per fer aquest desenvolupament em baso en el cas d'estudi mostrat en el llibre "Hands-on Domain-driven Design - by example" [19] . En aquest llibre s'estudia la implementació d'un software bancari complex on l'usuari ha de poder demanar un préstec bancari mitjançant un formulari i el banc ha de poder sol·licitar a l'agència de crèdit la puntuació d'aquest usuari i els deutes que té i a l'agència d'habitatges el valor de l'immoble per tal de poder prendre una decisió sobre el préstec.

Mitjançant tècniques com són el domain story telling es té un diagrama del domini complet de l'aplicació.



Il·lustració 16: Model del Domini

Tenint en compte el model de domini anterior Il·lustració 16 plantejarem en una primera fase amb el fi de comprovar la viabilitat del projecte la implementació dels passos 1-6. I tenint clara la implementació ja existent en una arquitectura orientada a microserveis [30]. Podem plantejar una sèrie de requisits tècnics:

- Utilitzar sempre que es pugui funcions serverless.
- Aconseguir que tota l'aplicació estigui testejada, per tant, de tenir un 100% de code coverage en els testos.
- Realitzar comunicació asíncrona entre les diferents parts de la nostra aplicació.
- Evitar l'encadenament de funcions.
- Fer deployment de les funcions en un proveïdor cloud.

Els punts 1-6 de la Il·lustració 16 constituiran l'aplicació desenvolupada formada per un formulari per aplicar pel préstec, aquest serà un SPA per tal de disminuir al màxim el tràfic de les lambdes. Per altra banda, s'ha de crear una API que pugui rebre la informació del formulari. Com l'objectiu és que tot siguin lambdes i que el codi estigui el mes descoplat possible separarem el codi necessari per demanar el score l'agència de crèdit. Aquesta agència de codi per tal que realment sigui extern la simularem amb una API de nombres aleatoris que ens donarà un valor de score aleatori.

Finalment, per tal que tot funcioni en conjunt és necessari la implementació d'una infraestructura de cues, servir contingut estàtic i funcions interconnectades.

10. Desenvolupament

10.1. Elecció del proveïdor cloud

Per poder dur a terme el projecte es necessita un proveïdor. Ja que com s'ha vist en seccions anteriors el vendor lock és una cosa a tenir en compte.
La proposta inicial és utilitzar un dels grans 3 proveïdors, Google Cloud, AWS o Azure, per aprofitar al màxim els avantatges de les tecnologies serverless i evitar problemes i preocupacions a l'hora de muntar la infraestructura i mantenir-la.
Com hem vist en l'apartat d'anàlisis dels proveïdors les característiques són pràcticament iguals en tots, per tant, preseleccionarem Azure, ja que té un programa de formació especialitat en cloud molt extens d'on treure exemples i a més a més donen un crèdit de 100 $ gratuïts per ser estudiants universitaris.

10.2. Preparar el entorn de desenvolupament

10.2.1. Sistema de versions

Per un projecte que busca emular el desenvolupament d'una aplicació fins a arribar a producció és necessari tenir un sistema de control de versions

En el nostre cas hem utilitzat el sistema de versions més utilitzat en desenvolupament, git i hostejat en GitHub. Tot el codi es pot trobar en el repositori públic https://github.com/alexindris/TFG2022-ALEX-TELLO-SERVERLESS

10.2.2. Framework serverless

Quan es desenvolupa una funció serverless no es necessita cap framework, es pot simplement compilar el codi i comprimir-ho en un ZIP tot junt i pujar-ho a la plataforma cloud. Aquest procediment no és sostenible en un entorn on tindràs més de 5 funcions i cada cop que s'actualitzi una dependència has de compilar, comprimir i pujar el codi.

Un altre avantatge que té d'utilitzar frameworks serverless és la gestió de permisos, quan es treballa amb proveïdors cloud es necessita atorgar permisos a les funcions per accedir als diferents recursos. Gestionar això fa créixer la complexitat de mantenir l'aplicació. Els frameworks fan aquesta feina per nosaltres, especifiquem en un fitxer de configuració els diferents recursos on tenen accés i, per altra banda, les regles que han de seguir per complir-ho.

Pel que fa a frameworks serverless hi ha poca diversitat d'opcions i no totes accepten totes les plataformes cloud.

10.2.2.1. Serverless framework

Aquest framework ens permet dissenyar la nostra infraestructura mitjançant un fitxer yaml Il·lustració 17 on especifiquem els recursos

```yaml
service: serverless-http-api
frameworkVersion: '3'

provider:
  name: aws
  runtime: nodejs14.x

functions:
  hello:
    handler: handler.hello
    events:
      - httpApi:
          path: /
          method: get
```

Il·lustració 17: Fitxer de configuració serverless

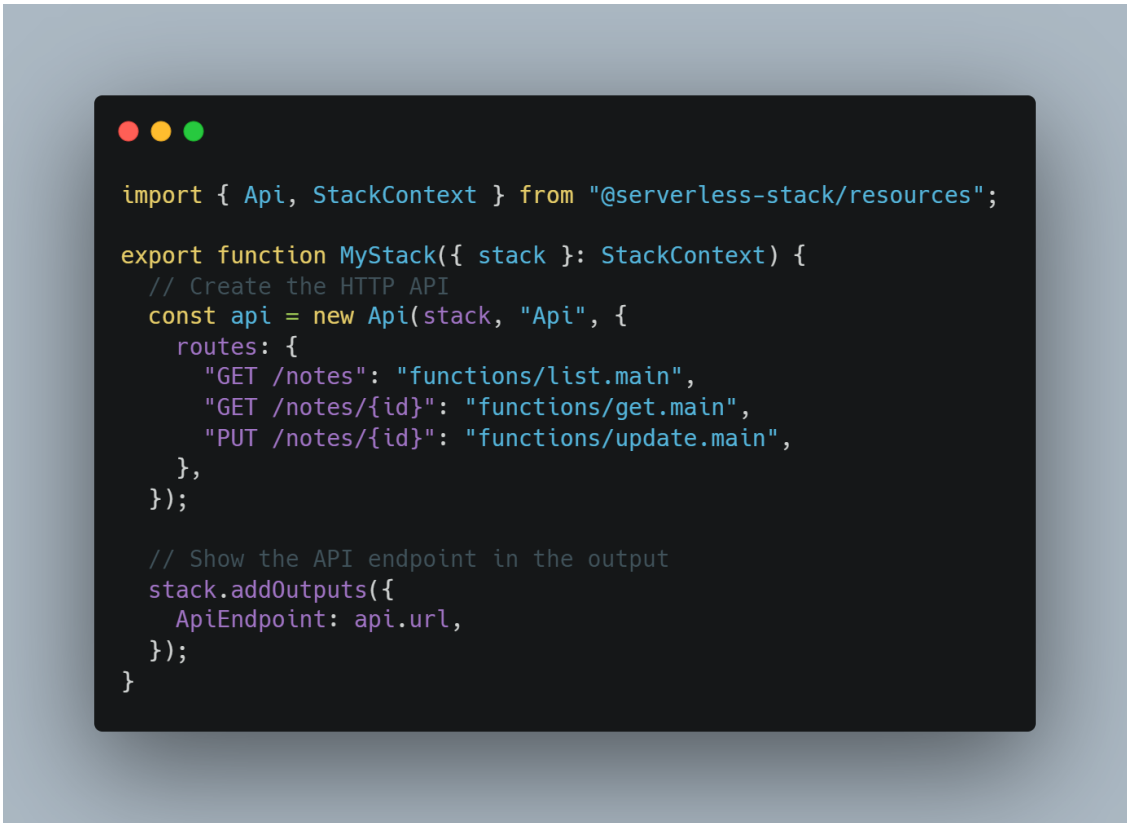
És parcialment multiplata forma, és a dir tot l'ecosistema, està llest per funcionar amb aws i amb la resta de proveïdors ha d'adaptar-se. Provocant una falta de compatibilitat amb altres sistemes.

10.2.2.2. SAM

SAM, Serverless Application Model és un framework de codi obert proveït per Amazon amb una sintaxi pràcticament igual que serverless framework. És molt més nou que serverless framework, però creix molt ràpidament impulsat per AWS que és per qui està destinat a funcionar.

10.2.2.3. Serverless-Stack

Serverless-Stack és un framework que segueix la mateixa filosofia que els anteriors, però s'aproxima molt més al qual seria infraestructura com a codi Il·lustració 18.



```
import { Api, StackContext } from "@serverless-stack/resources";

export function MyStack({ stack }: StackContext) {
  // Create the HTTP API
  const api = new Api(stack, "Api", {
    routes: {
      "GET /notes": "functions/list.main",
      "GET /notes/{id}": "functions/get.main",
      "PUT /notes/{id}": "functions/update.main",
    },
  });

  // Show the API endpoint in the output
  stack.addOutputs({
    ApiEndpoint: api.url,
  });
}
```

Il·lustració 18: fitxer de configuració Serverless-Stack

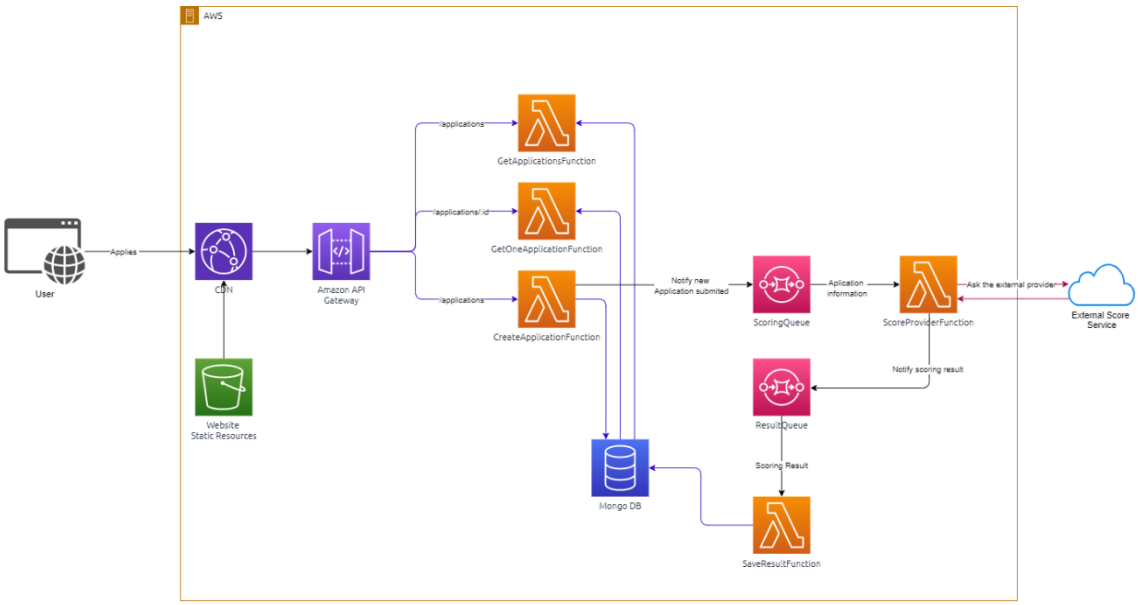
Aquest framework abstreu la infraestructura a objectes que s'instancien a l'hora de fer deployment simplificant la utilització dels proveïdors cloud. En l'actualitat únicament és compatible amb AWS.

A més a més proporciona una consola per poder desenvolupar les funcions localment i fer debug directament en el teu navegador.

10.3. Parts de la aplicació

10.4. Arquitectura de l'aplicació

Tenint en compte el que hem determinat prèviament la arquitectura que implementarem serà la següent:



Il·lustració 19: Arquitectura de l'aplicació

El usuari accedirà a la nostra aplicació mitjançant el contingut estàtic servit per el CDN.

Un cop el usuari completi el formulari s'envia a la API que crida la lambda de creació de la aplicació, aquesta tindrà la responsabilitat de guardar l'aplicació de la base de dades i sol·licitar que es calculi el seu score. Per fer això el que fa es crear un esdeveniment a la cua ScoringQueue.

Un cop es rep aquest event per part de la ScoringProviderFunction aquesta, s'encarrega de demanar el score al proveïdor i enviar a la ResultQueue el resultat.

Aquest event invoca una funció que la seva responsabilitat es actualitzar la puntuació rebuda per part del proveïdor a la aplicació.

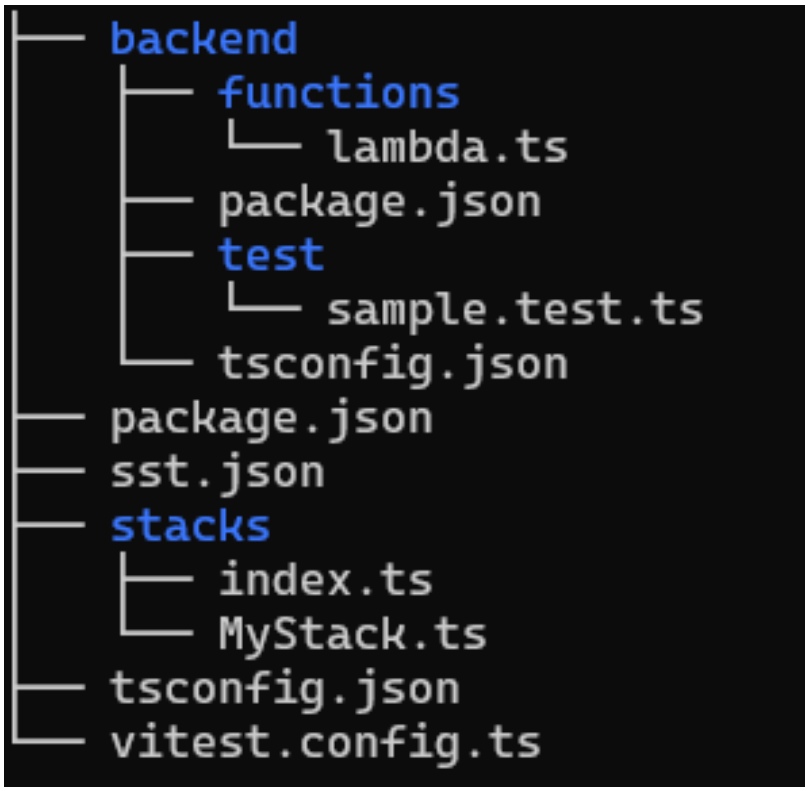
10.5. Creació del projecte

Tenint en compte el que hem avaluat anteriorment el proveïdor Azure no ens permetria utilitzar cap dels frameworks que hem contemplat anteriorment, per tant, haurem de canviar a AWS amb l'objectiu de poder utilitzar allò que més se'ns adapti al nostre projecte.

Doncs, com el propòsit de la nostra aplicació és fer servir diferents lambdes que es comuniquin asíncronament i no volem gestionar permisos. Farem servir el framework Serverless-stack.

Seguint la documentació oficial inicialitzem el projecte amb la plantilla typescript-starter usant la comanda "npm init sst typescript-starter nom-del-projecte".

Crea una estructura de fitxers com la següent Il·lustració 20



Il·lustració 20: Estructura de carpetes inicial

10.6. Stack

Primer de tot hem de definir els punts d'entrada de l'aplicació i els recursos que necessitarem. Per tal de fer això el que s'ha de fer és crear un fitxer dins la carpeta stacks, en el nostre cas es dirà "BigPugBank.ts"

Dissenyant tota l'arquitectura anterior en forma de funcions.

Primer de tot les cues amb les funcions que consumiran els events de les cues.

```
// Create ResultQueue
const resultQueue = new Queue(stack, "resultQueue", {
  consumer: {
    function: {
      handler: "workers/resultWorker.save",
      timeout: 10,
      environment: {
        DATABASE_URL: process.env.DATABASE_URL ?? "default_uri",
        DATABASE_NAME: process.env.DATABASE_NAME ?? "test",
      },
    },
  },
}
```

Il·lustració 21: Creació de la cua de resultats

En la Il·lustració 21 podem veure com creem una cua anomenada "resultQueue" i els seus events són consumits per una funció lambda que té un handler. Com aquesta lambda ha de poder escriure a la base de dades també necessita les variables d'entorn amb la connexió a la base de dades.

En el cas de la cua de la puntuació Il·lustració 22 a diferència de l'anterior que s'ha de comunicar a la base de dades, aquesta ha de comunicar-se amb la cua de resultats, per tant, els permisos i les variables d'entorn són de l'altra cua.

```
// Create ScoringQueue
const scoringQueue = new Queue(stack, "scoringQueue", {
  consumer: {
    function: {
      handler: "workers/scoringWorker.score",
      timeout: 10,
      environment: {
        RESULT_QUEUE_NAME: resultQueue.queueName,
      },
      permissions: [resultQueue],
    },
  },
});
```

Il·lustració 22: Creació cua de la puntuació

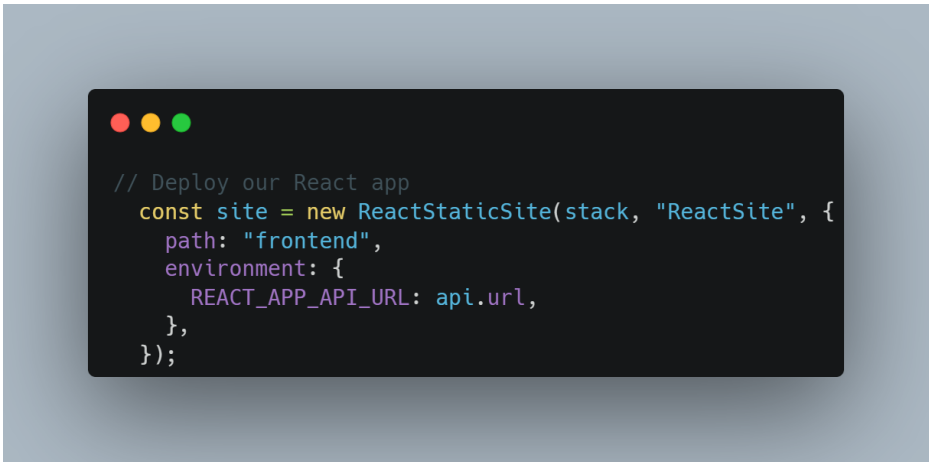
La següent part a instanciar és l'API per fer-ho, crearem totes les rutes i farem que apuntin a la funció en qüestió Il·lustració 23 i a més a més afegirem permisos a l'API per accedir tant a la base de dades com a la cua de la puntuació.

```
// Create the HTTP API
  const api = new Api(stack, "Api", {
    defaults: {
      function: {
        environment: {
          SCORING_QUEUE_NAME: scoringQueue.queueName,
          DATABASE_URL: process.env.DATABASE_URL ?? "default_uri",
          DATABASE_NAME: process.env.DATABASE_NAME ?? "test",
        },
      },
    },
    routes: {
      "GET /applications": "./api/applications.getAll",
      "POST /applications": "./api/applications.create",
      "GET /application/{id}": "./api/applications.getOne",
    },
  });

  api.attachPermissions([scoringQueue]);
```

Il·lustració 23: Creació API

I per últim ens falta el front-end Il·lustració 24 per fer que utilitzi un CDN i serveixi el contingut estàtic utilitzarem la construcció predefinida de "ReactStaticSite" on únicament hem de dir a quina carpeta tenim els arxius del front-end i les variables d'entorn que vulguem utilitzar en el nostre cas únicament serà l'URL de l'API.



```
// Deploy our React app
  const site = new ReactStaticSite(stack, "ReactSite", {
    path: "frontend",
    environment: {
      REACT_APP_API_URL: api.url,
    },
  });
```

Il·lustració 24: Crear Plana Estàtica

Un cop definit tot el stack modificarem el fitxer "index.ts" Il·lustració 25 per afegir el nostre nou stack i per especificar que totes les funcions utilitzaran 128 MB de memòria ram i estaran dins la carpeta backend.

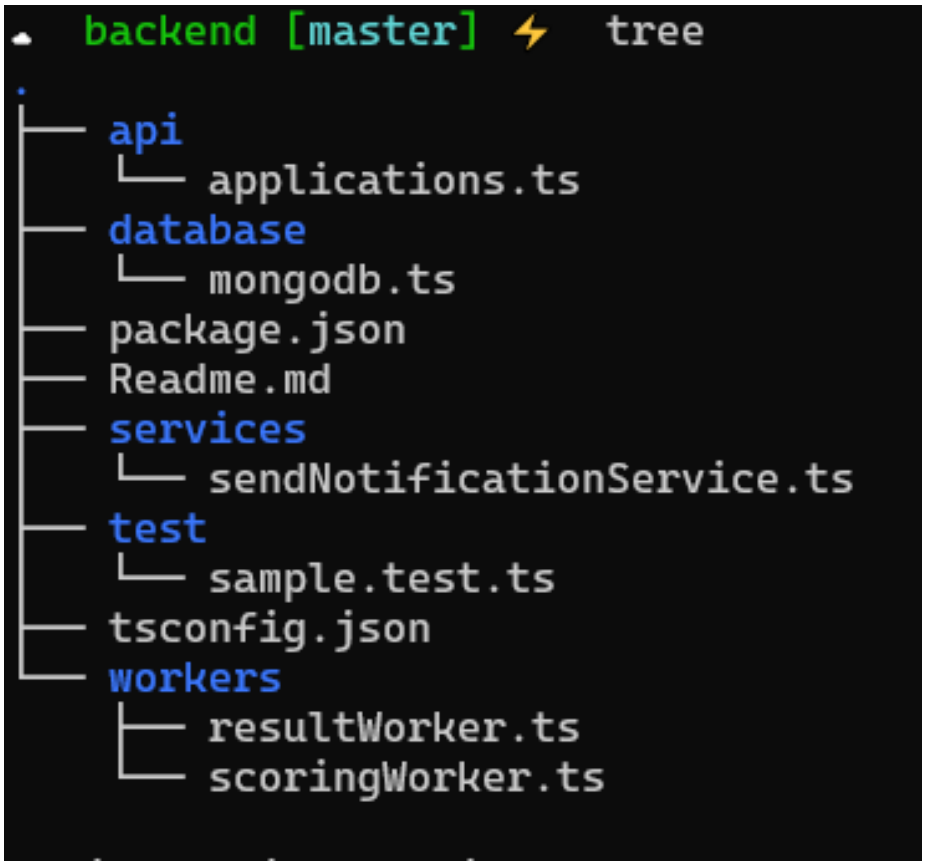
```
import { BigPugBank } from "./BigPugBank";
import { App } from "@serverless-stack/resources";

export default function (app: App) {
  app.setDefaultFunctionProps({
    runtime: "nodejs16.x",
    srcPath: "backend",
    memorySize: 128,
    bundle: {
      format: "esm",
    },
  });
  app.stack(BigPugBank);
}
```

Il·lustració 25: Definició del nou Stack

10.7. Backend

Primer de tot en el backend crearem la següent estructura Il·lustració 26.



Il·lustració 26: Estructura Backend

L'estructura consta de les següents parts:

- Api → Tots els fitxers on apunten les lambdes que responen a peticions HTTP/HTTPS
- Database → Configuració de la base de dades
- services → Funcions genèrica que tenen com a objectiu facilitar la integració de l'aplicació amb la infraestructura.
- Workers → Fitxers on apunten les lambdes consumidores d'events asíncrons

10.7.1. Funcions API

Quan hem definit prèviament les rutes de l'API Il·lustració 27 hem dit que la ruta GET /applications està a la carpeta api en el fitxer Applications.ts en la funció getAll.

```
routes: {
      "GET /applications": "./api/applications.getAll",
      "POST /applications": "./api/applications.create",
      "GET /application/{id}": "./api/applications.getOne",
    },
```

Il·lustració 27: Definició rutes API

Exportem la funció Il·lustració 28 on connectem a la base per aconseguir el client de base de dades i agafem totes les aplicacions que tenim. I retornem la resposta que donarà l'API.

```
export async function getAll(event, context) {
  context.callbackWaitsForEmptyEventLoop = false;

  // Get all applications from the database
  const db = await connectToDatabase();

  const applications: Application[] = (await db
    .collection("Application")
    .find()
    .toArray()) as unknown as Application[];

  return {
    statusCode: 200,
    body: JSON.stringify(applications),
    headers: { "Content-Type": "application/json" },
  };
}
```

Il·lustració 28: Funció getAllApplications

10.7.2. Serveis

En el cas dels serveis els utilitzarem principalment per enviar events a la cua com és el cas de la següent Il·lustració 29.

```
import AWS from "aws-sdk";

export async function sendNotification(queueName: string, data: Object) {
  const sqs = new AWS.SQS();

  const queueUrl = await sqs.getQueueUrl({ QueueName: queueName }).promise();

  await sqs
    .sendMessage({
      QueueUrl: queueUrl.QueueUrl,
      MessageBody: JSON.stringify(data),
    })
    .promise()
    .catch((err) => {
      throw err;
    });
}
```

Il·lustració 29: Servei de Notificacions

A la funció hi passem per paràmetres el nom de la cua i l'objecte que volem serialitzar per enviar, en aquest exemple utilitzem la llibreria d'AWS per enviar el missatge a la cua.

10.7.3. Workers

Aquestes són les funcions que consumeixen els events de les cues aquestes a diferència de les de l'API hem dit que es troben sota la carpeta workers en el cas de la funció de resultats es troba sota el fitxer "scoringWorker.ts" Il·lustració 30 rep per paràmetre un event SQS, el mapeja en aquest cas al model Application i per cada aplicació sol·licita a un servei extern la seva valoració. En aquest cas utilitzarem una API mock que crearem posteriorment.

```typescript
import { Application } from "shared/models/Application";
import { SQSEvent } from "aws-lambda";
import { sendNotification } from "../services/sendNotificationService";
import fetch from "node-fetch";

export async function score(event: SQSEvent) {
  const applications: Application[] = event.Records.map((record) =>
    JSON.parse(record.body)
  );

  applications.forEach(async (application: Application) => {
    //  Make a request to the mock API to get the score
    const result = await fetch(
      "https://62a46144259aba8e10e750c0.mockapi.io/scoring/api/v1/apply",
      {
        method: "POST",
        body: JSON.stringify(application),
      }
    );

    const response = (await result.json()) as response;

    application.score = response.score;
    sendNotification(process.env.RESULT_QUEUE_NAME, application);
  });
}

type response = {
  createdAt: string;
  score: number;
  id: string;
};
```

Il·lustració 30: Implementació del Score Worker

10.8. Front-end

Per al front-end també canviarem l'estructura de les carpetes Il·lustració 31.

```
frontend [master] tree
.
├── node_modules
├── package.json
├── package-lock.json
├── public
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
│   ├── manifest.json
│   └── robots.txt
├── README.md
└── src
    ├── assets
    │   ├── css
    │   │   └── index.css
    │   └── img
    │       ├── BigPugLoans.jpg
    │       ├── favicon.svg
    │       └── logo.svg
    ├── components
    │   ├── ApplicantForm.jsx
    │   ├── FinancingForm.jsx
    │   ├── HouseHoldForm.jsx
    │   └── LoanForm.jsx
    ├── index.js
    ├── pages
    │   ├── Application.jsx
    │   └── Index.jsx
    ├── reportWebVitals.js
    └── setupTests.js

8 directories, 22 files
```

Il·lustració 31: Estructura carpetes Frontend

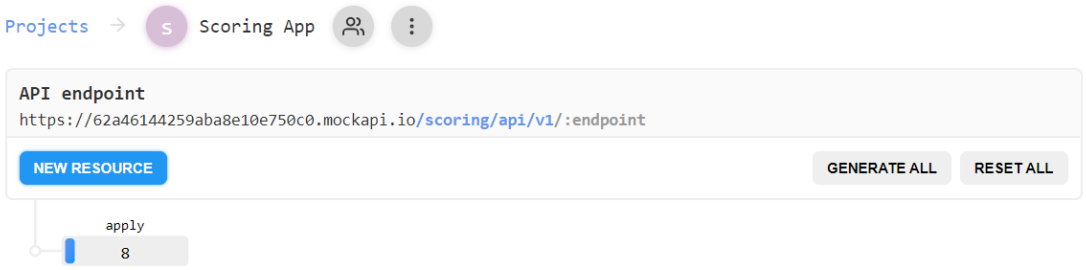
L'estructura consta de les següents parts:

- node_modules → Dependències de node
- public→ Contingut estàtic.
- assets→Son les imatges i css utilitzats en l'aplicació
- components→Fraccions de l'aplicació que constitueixen una pàgina.
- pages→Diferents parts de l'aplicació enllacades per una ruta

El resultat es pot trobar al següent enllaç https://dd5np5vytmwec.cloudfront.net/

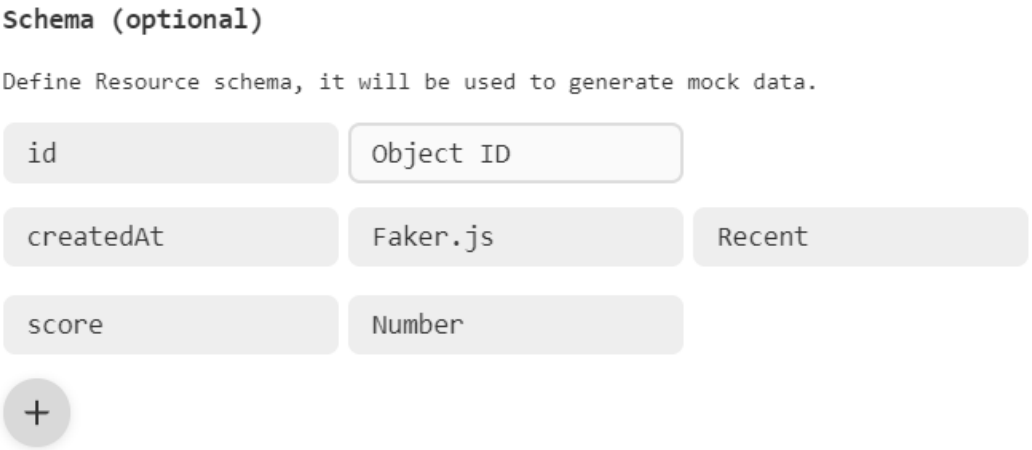
10.9. Mock API

Amb l'objectiu de simular l'aplicació externa es crea una API amb el servei mockapi.io. Crearem un projecte i especificarem la ruta /scoring/api/v1/ Il·lustració 32



Il·lustració 32: Projecte a MockAPI.io

Un cop aquest punt s'ha completat crearem l'endpoint de l'API , en el nostre cas es diu apply. I crearem l'esquema de dades que esperem que retorni l'API Il·lustració 33 . L'API retornarà una id del objecte, una data de creació i per últim el score aleatori que necessitem.



Il·lustració 33: Esquema API Scoring

11. Conclusions

L'objectiu d'aquest treball era analitzar la tecnologia serverless i validar la viabilitat de desenvolupar una aplicació fent ús únicament de la tecnologia serverless.

Segons el que s'ha analitzat, la tecnologia serverless té molts pros en aplicacions pensades per funcionar en arquitectures basades en events o de manera distribuïda. Així mateix, també se'n beneficien aquelles aplicacions amb tràfic variable on la seva infraestructura i capacitat de computació ha de créixer i disminuir segons el tràfic o les prestacions necessàries en un moment temporal precís.

El FaaS permet disminuir el temps de desenvolupament, ja que el prototipatge i el desplegament d'infraestructura queda simplificat a una configuració inicial i el desenvolupament de cadascuna de les funcions es tracta com una part independent de l'aplicació .

Per altra banda, la tecnologia serverless té falles en entorns on el temps de resposta és crític o no es pot conviure amb la fallida de cap funció aquesta tecnologia no és la ideal per ser desenvolupada. Això és a causa de diversos factors, un d'ells el cold start que com hem indicat anteriorment deteriora el temps de reacció cada cop que l'aplicació s'executa quan no hi ha execucions prèvies en un interval de temps reduït.

A més a més un altre desavantatge que té la tecnologia principalment degut a la falta de maduresa i a ser un sistema distribuït és la captació de logs i poder fer debug. Això afecta el desenvolupament, dificultant la tasca d'integració entre les diferents funcions.

Durant el desenvolupament aquestes dificultats i avantatges s'han manifestat, la falta de debug va provocar un endarreriment de més de dues hores quan s'implementaven les cues. Perquè no es rebien els events ni tan sols hi havia un error visible. Per altra banda, un cop la infraestructura ja estava configurada expandir l'aplicació i provar les diferents funcions era senzill

En conclusió malgrat que la tecnologia serverless és una molt nova i encara té mancances a nivells d'estàndard que unifiquin totes les plataformes i permetin una fàcil transició entre proveïdors. Aporta grans avantatges en entorns productius i de desenvolupament fins al punt que empreses amb grans sistemes transicionen parts de la seva infraestructura cloud.

12. Possibles ampliacions

L'objectiu del projecte era demostrar que es podia desenvolupar una aplicació com la del cas d'estudi únicament amb tecnologies serverless. Malgrat que s'ha aconseguit el propòsit hi ha diferents punts de millores.

12.1. Testing

Pel que fa a testing únicament s'ha fet un test d'integració on es valida que el front-end realment funciona i es pot omplir el formulari com s'ha especificat, però ha mancat realitzar l'unit testing tant del front-end com del back-end i testejar amb el framework serverless la infraestructura de back-end.

12.2. CI/CD

Un dels punts on els sistemes distribuïts realment destaquen envers sistemes monolítics és el desenvolupament continu i aprofitant l'ampliació anterior seria una bona millora pel desenvolupament i manteniment del codi.

12.3. Seguretat

Actualment, l'aplicació no disposa de cap mena de seguretat ni validació d'usuaris, per tant, una millora necessària implementar un sistema d'usuaris, restringir accés a alguns endpoints de l'API mitjançant permisos i rols. Una de les implementacions seria utilitzar Amazon Cognito.

12.4. Notificació del resultat

I per últim l'usuari actualment no disposa de feedback del score realitzat i seria convenient que ho tinguis mitjançant un sistema de notificacions com podria ser email, SMS o push notifications.

13. Bibliografia

[1] S. Newman, Building Microservices, O'Reilly Media, Inc., 2015, p. 280.

[2] P. Michael, A. S. Varde, S. A. Robila i A. Ranganathan, «A call for energy efficiency in data centers,» *Association for Computing Machinery,* vol. 43, núm. 1, 2014.

[3] AWS, *Coca-Cola - Running Serverless Applications with Enterprise Requirements,* 2016.

[4] Netflix, «The Netflix Cosmos Platform,» Netflix, 01 Març 2021. [En línia]. Available: https://netflixtechblog.com/the-netflix-cosmos-platform-35c14d9351ad. [Últim accés: 06 Abril 2022].

[5] AWS, «Release AWS Lambda,» 13 11 2014. [En línia]. Available: https://aws.amazon.com/releasenotes/release-aws-lambda-on-2014-11-13/. [Últim accés: 07 02 2022].

[6] AWS, «AWS Lambda Quotas,» [En línia]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. [Últim accés: 28 02 2022].

[7] Google, «Writing Cloud Functions,» Google Cloud, [En línia]. Available: https://cloud.google.com/functions/docs/writing. [Últim accés: 28 02 2022].

[8] A. K. a. S. C. Andrei Palade, «An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge,» School of Computer Science and Statistics, Trinity College Dublin, Ireland, 2019.

[9] AWS, «What is an Event-Driven Architecture,» [En línia]. Available: https://aws.amazon.com/event-driven-architecture/. [Últim accés: 08 03 2022].

[10] I. Velichko, «OpenFaaS - Run Containerized Functions On Your Own Terms,» 06 Desembre 2021. [En línia]. Available: https://iximiuz.com/en/posts/openfaas-case-study/. [Últim accés: 15 Març 2022].

[11] Cloud Event, «CloudEvents Spec - Version 1.0.1,» Cloud Fundation, 03 05 2021. [En línia]. Available: https://github.com/cloudevents/spec/blob/v1.0.1/spec.md. [Últim accés: 13 03 2022].

[12] Google Cloud, «Eventarc overview,» Google, 07 03 2022. [En línia]. Available: https://cloud.google.com/eventarc/docs/overview. [Últim accés: 13 03 2022].

[13] Cloud Event, «JSON Event Format for CloudEvents - Version 1.0,» Cloud Native Fundation, 11 Enero 2020. [En línia]. Available: https://github.com/cloudevents/spec/blob/v1.0/json-format.md. [Últim accés: 13 Marcç 22].

[14] Cloud Event, «HTTP Protocol Binding for CloudEvents - Version 1.0,» Cloud Native Fundation, 11 Gener 2020. [En línia]. Available: https://github.com/cloudevents/spec/blob/v1.0/http-protocol-binding.md. [Últim accés: 13 Març 2022].

[15] W. B. Hohpe Gregord, «Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions,» de *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Martin Flowers, 2022, pp. 286-290.

[16] AWS, «Eventbridge,» AWS, [En línia]. Available: https://aws.amazon.com/es/eventbridge/. [Últim accés: 06 Abril 2022].

[17] Azure, «Azure Event Grid,» Microsoft, [En línia]. Available: https://azure.microsoft.com/es-es/services/event-grid. [Últim accés: 06 Abril 2022].

[18] URENIO Research Unit, «Cost Benefit Analysis Methodology,» 2001. [En línia]. Available: https://www.urenio.org/newventuretools/cba/cba_components.html. [Últim accés: 07 02 2022].

[19] M. Plöd, Hands-on Domain-driven Design - by example, Michael Plöd, 2020.

[20] M. Clark, «Moving BBC Online to the cloud,» *BBC Product & Technology*, 29 Octubre 2020.

[21] M. Clark, «Architecting for scale with the cloud and serverless. A case study of BBC Online,» de *QCon+*, 2021.

[22] Copernicus, «Copernicus,» European Union's Earth Observation Program, [En línia]. Available: https://www.copernicus.eu/es/sobre-copernicus. [Últim accés: 12 Abril 2022].

[23] W. C. S. P. ,. S. M. André Twele, «Sentinel-1-based flood mapping: a fully automated,» *International Journal of Remote Sensing*, vol. 37, núm. 13, pp. 2990-3004, 2016.

[24] S. P. K. C. Sandro Martinis, «The Use of Sentinel-1 Time-Series Data to Improve,» *Remote Sensing*, pp. 1-13, 9 Abril 2018.

[25] Argo Workflows, «What is Argo Workflows,» Argo Workflows, 04 Febrer 2022. [En línia]. Available: https://argoproj.github.io/argo-workflows/. [Últim accés: 12 Abril 2022].

[26] Minio, «Multi-Cloud Object,» Minio, [En línia]. Available: https://min.io/. [Últim accés: 12 Abril 2022].

[27] M. Roberts, «Serverless Architectures,» Martin Fowler, 22 Maig 2018. [En línia]. Available: https://martinfowler.com/articles/serverless.html. [Últim accés: 19 Abril 2022].

[28] A. Wiggins, The Twelve-Factor App, Twelve-Factor App methodology, 2011.

[29] B. Kepes, «30% Of Servers Are Sitting "Comatose" According To Research,» *Forbes*, p. 1, 3 Juny 2015.

[30] M. Plöd, «ddd-with-spring,» 25 Març 2018. [En línia]. Available: https://github.com/mploed/ddd-with-spring. [Últim accés: 02 Febrer 2022].