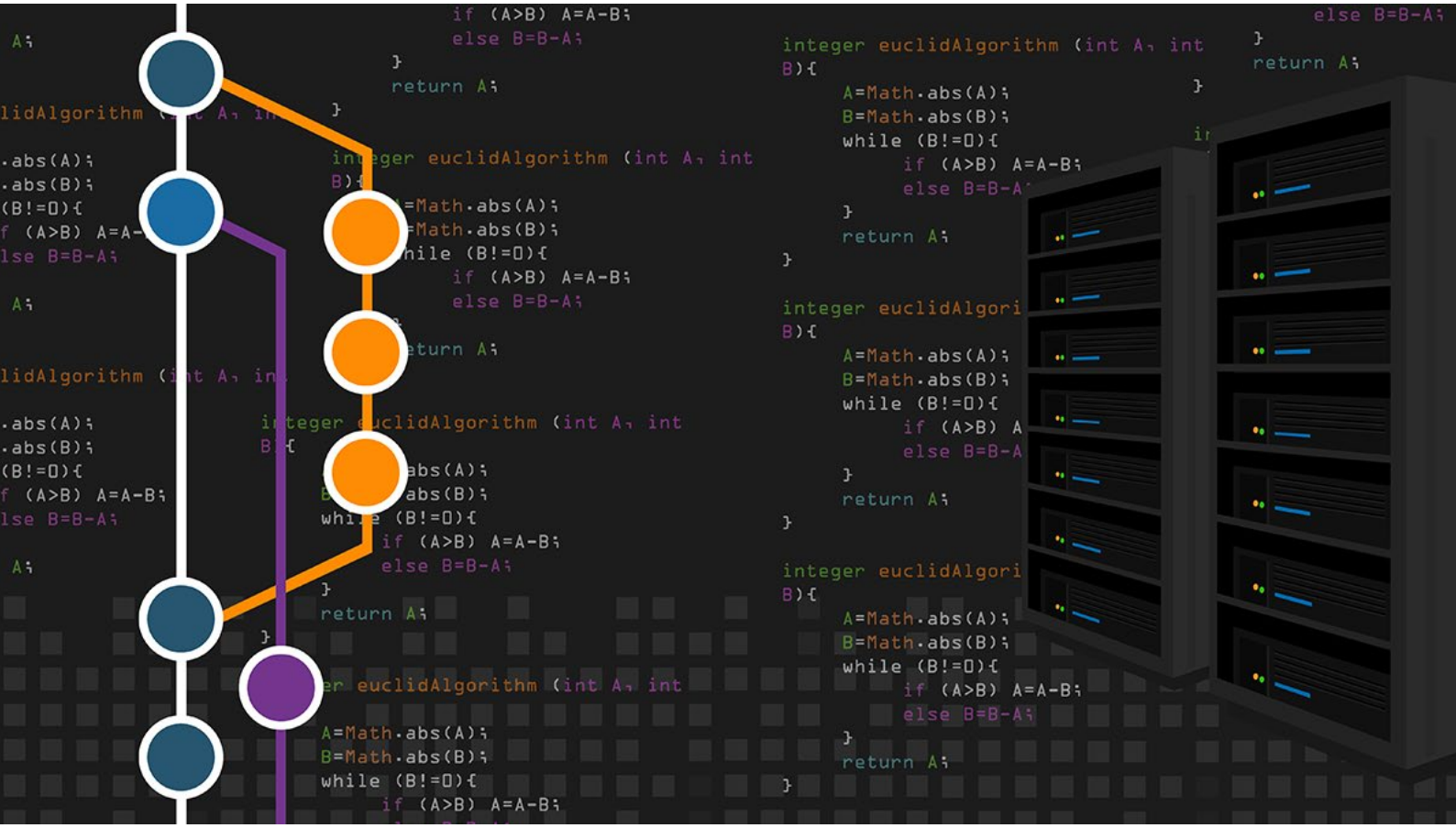
GitOps: l'evolució de la cultura DevOps

Treball Final de Grau



Angel Saldaña López
Grau d'Enginyeria Informàtica
Àrea d'administració de xarxes i sistemes operatius
Curs 2020-2021

FITXA DEL TREBALL

Títol del treball:	*GitOps: l'evolució de la cultura DevOps*
Nom del autor:	*Angel Saldaña López*
Nom del consultor:	*Miguel Martín Mateo*
Data d'entrega:	*06/2021*
Titulació:	*Grau d'Enginyeria Informàtica*
Àrea:	*Administració de xarxes i sistemes operatius*
Idioma:	*Català*
Paraules clau:	*GitOps, DevOps, Microservices, CI/CD, Kubernetes, Cloud, Automatització, Infrastructure as Code, Golang*

Resum:

Aquest Treball de Final de Grau té com objectiu principal aprofundir i investigar els beneficis que ofereix la metodologia GitOps als cicles de desenvolupament de programari.

El projecte està dividit en dues parts ben diferenciades. En la primera, es realitza una exposició de caire més teòric on s'estudia el flux de desplegament d'una aplicació seguint els principis GitOps així com els components requerits per poder implementar aquesta solució. En canvi, la segona part consisteix en dissenyar i implementar una plataforma que permeti demostrar com funcionaria l'automatització de tots els processos de CI/CD per desplegar tant infraestructura com una aplicació web seguint el mateix criteri.

La solució tècnica està hostatjada en Google Cloud Platform i les eines principals que s'han utilitzat són Kubernetes, Terraform, GitHub, Golang, Cloud Build i ArgoCD.

Abstract:

The main goal of this Bachelor's Degree Final Project is to learn and investigate the benefits of applying GitOps methodologies to the software development life cycles.

The project is divided into two main sections. In the first one, are exposed the theoretical concepts about how is an application deployment workflow following GitOps principles and also which components are required to put in place this solution. In contrast, the second section is more about a practice use case where is designed and implemented a platform that allows demonstrating how would work the automated CI/CD process to deploy infrastructure and a web application following the same pattern.

The technical solution is hosted in Google Cloud Platform and the most important toolset used to complete this project are Kubernetes, Terraform, GitHub, Golang, Cloud Build, and ArgoCD.

Índex

1. Introducció

En els darrers anys, l'accelerada digitalització dels hàbits de la societat i del processos de les organitzacions, està provocant que cada vegada més hi hagi major dependència amb les aplicacions informàtiques i els serveis web, convertint-se en la majoria de casos, en eines fonamentals pel desenvolupament de la majoria de tasques que es realitzen tant en l'àmbit individual com en el professional.

En aquest context, les empreses tecnològiques i els propis departaments TIC de qualsevol corporació, es troben davant un repte i unes peticions cada cop més exigents per a satisfer les necessitats de les persones usuàries que utilitzen el seus productes a diari per a realitzar multitud d'operacions quotidianes. En molts casos, un problema en una solució digital, pot impactar directament a l'operativa d'un negoci provocant pèrdues econòmiques importants durant el període de temps en què l'aplicació ha estat indisponible. Per no esmentar la repercussió mundial que té quan és alguna de les plataformes tecnològiques més esteses (Google, WhatsApp, Netflix, Twitter, Instagram, etc...) la que reporta alguna incidència.

Les característiques que ha de tenir una aplicació per a poder donar resposta a les demandes anteriors són principalment les que es detallen a continuació:

- **Alta disponibilitat** per garantir la continuïtat del servei.
- **Escalabilitat** per suportar un major número concurrent de peticions sense que es vegi afectat el rendiment.
- **Seguretat i privacitat** de les dades.
- **Desplegament freqüent** de noves versions de les aplicacions per **generar valor per les persones usuàries.**

Per a poder complir amb els requisits anteriors, les organitzacions s'han hagut de transformar per adoptar noves metodologies i tecnologies que han esdevingut clau en la revolució digital que ha patit el sector informàtic durant la última dècada, on en destaquen principalment els següents punts:

- **Cultura DevOps:** aquesta filosofia de treball és un moviment que va aparèixer entre els anys 2007-2008 derivat de les metodologies Lean i Agile. La seva principal característica radica en potenciar la col·laboració entre els equips de desenvolupament i d'operacions per alinear els seus interessos particulars en benefici dels objectius globals de l'organització, eliminant els *silos*[1] entre departaments que existien en el passat.

  Amb la introducció d'aquesta metodologia de treball es pretén arribar a tenir equips multidisciplinaris i autònoms que assumeixin la responsabilitat total sobre cada un dels cicle de desenvolupament d'un producte: començant per la fase de codificació, continuant pel desplegament de la solució en cada un dels entorns i acabant amb el manteniment del servei a producció.

[1] *Concepte que defineix aquelles organitzacions on hi ha aïllament entre els departaments on manca la col·laboració, la comunicació i la transferència de coneixement.*

L'aparició del terme DevOps va venir acompanyat d'un seguit de pràctiques i eines que faciliten l'estandardització de processos dins de les organitzacions, entre les quals podem destacar els següents conceptes:

- **Automatització:** un dels canvis més importants que ha introduït la filosofia DevOps en l'administració de sistemes tradicional ha sigut l'automatització de tasques repetitives que anteriorment requerien d'intervenció manual com, per exemple, la configuració dels servidors o de les aplicacions. A més a més, aquest codi generat per automatitzar l'aprovisionament dels sistemes ha d'estar emmagatzemat en eines de control de versions (Git principalment) per tenir traçabilitat sobre els canvis que es realitzen en la plataforma i fomentar la col·laboració entre diferents equips.

  Tanmateix, l'automatització no ha afectat únicament l'administració de sistemes sinó que també ha transformat la manera en què es desenvolupa i es prova el software introduint, dins d'aquestes àrees, l'aplicació de conceptes com **TDD** (*Test Driven Development*) i **BDD** (*Behave Driven Development*)

- **CI/CD:** aquestes sigles introdueixen dos conceptes fonamentals que han contribuït especialment en augmentar la freqüència i la qualitat en què es distribueixen les aplicacions.

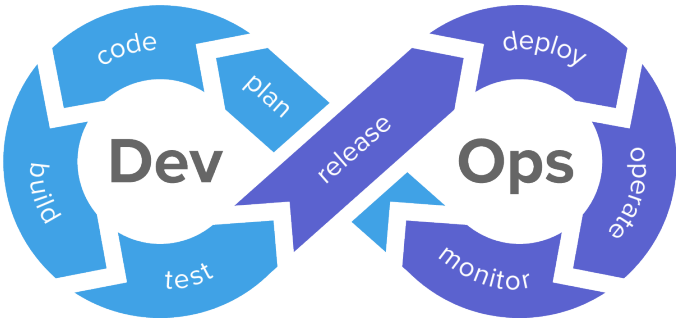
  D'una banda, tenim el terme **CI** (*Continuous Integration*) que està orientat en automatitzar els tests que s'executaran sobre un codi nou afegit per l'equip de desenvolupament. Aquests tests, retornaran en un curt període de temps, si els canvis proposats no introdueixen cap defecte ni incompatibilitat amb el codi actual, oferint garanties i seguretat per incorporar-los.

  En canvi, el concepte **CD** (*Continuous Delivery/Continuous Deployment)*, succeeix una vegada els canvis proposats i comprovats pel procés d'integració continua s'han afegit al codi actual generant així una nova versió de l'aplicació. Aquest procés, genera un nou artefacte que es prepara per a ser entregat automàticament a l'entorn que correspongui gràcies a l'execució de cada una de les fases definides en les *pipelines*[2] de desplegament.

- **Monitoratge i observabilitat:** per fer un correcte seguiment de l'estat de la infraestructura i de les aplicacions és essencial disposar de dades en temps real sobre el seu rendiment. Per aquesta raó, és fonamental disposar d'eines que permetin als equips tècnics identificar els problemes actuals abans que siguin reportats per les persones usuàries dels sistemes i, encara més important, anticipar-se a que aquests succeeixin. Tenir en bon estat la plataforma, permet augmentar la productivitat de les organitzacions al reduir les interrupcions provocades per la caiguda d'un sistema.

[2] *Conjunt de fases o passos per les que passa el desplegament d'una nova versió d'una aplicació*

A continuació, es mostra una imatge molt representativa sobre les fases que existeixen en el cicle de desenvolupament de programari i la relació existent entre cada una de les fases que la composen:



Il·lustració 1 – DevOps workflow
Font: https://www.programaenlinea.net/la-importancia-de-la-adopcion-de-devops-en-los-equipos/

- **Cloud Computing**: l'ús de plataformes de cloud públic és una tecnologia fonamental sobre la qual moltes companyies s'han recolzat per dissenyar el seu full de ruta cap a la digitalització, obtenint els avantatges següents:

  - Flexibilitat i escalabilitat en l'ús de recursos.
  - Inversió inicial molt reduïda.
  - Administració i manteniment delegat.
  - Model de seguretat i privacitat compartit.

  Cal indicar que existeixen diferents models de solucions en funció del tipus del servei ofert per les companyies, entre els quals tenim les següents opcions:

  - **IaaS:** les sigles corresponen al terme *Infrastructure as a Service* i es tracta d'un model on es té pràcticament el control total sobre la infraestructura desplegada a la plataforma. El proveïdors més importants són Amazon Web Services, Microsoft Azure i Google Cloud.

  - **PaaS:** les sigles corresponen al terme *Platform as a Service* i el model ofert permet gestionar una part del servei contractat, principalment enfocat en la part purament de l'aplicació. Exemples de solucions PaaS tenim Heroku, AWS Beanstsalk, Google App Engine o OpenShift.

  - **SaaS:** les sigles corresponen al terme *Software as a Service* i es tracta d'un model completament gestionat per l'empresa proveïdora on tenim la possibilitat d'utilitzar una aplicació habitualment a través del navegador. Aquests tipus de serveis són els més estesos i com a exemple tenim, entre d'altres: Microsoft Office 365, Adobe Creative Cloud o Google Workspace.

  Tanmateix, existeixen altres models de computació al núvol en funció de la seva ubicació o la propietat que es classifiquen en: cloud privat, cloud públic, cloud híbrid i multicloud.

- **Arquitectura de microserveis:** aquest patró de disseny d'aplicacions va ser descrit per primera vegada l'any 2005 per Peter Rodgers, concretament, fent menció al terme

"*micro-web-services*" tot i que no va ser fins l'any 2011 quan va disposar d'una major visibilitat gràcies a les aportacions de les companyies Netflix i Amazon.

Aquests tipus d'arquitectures intenten solucionar els problemes identificats en aplicacions antigues, anomenades monolítiques, on un mateix repositori de codi inclou multituds de components i funcionalitats. Aquest fet, provoca que amb el pas del temps el codi font creixi i sigui cada vegada més difícil implementar noves funcionalitats perquè el risc d'introduir un *bug*[3] augmenta al ser més complicat entendre tota la lògica de l'aplicació. A més a més, un petit canvi en el codi implica un desplegament total i, en cas d'error, generaria una caiguda general del sistema.

Pels motius anteriors, les aplicacions basades en arquitectura de microserveis, intenten solucionar les mancances identificades posant en pràctica els següents conceptes:

- **Una única responsabilitat:** aquesta funció, copiada dels "*SOLID principles*", estableix que una aplicació ha de realitzar una única funció però fer-la perfectament.

- **Cicles curts de desenvolupament:** el codi generat ha d'incorporar l'automatització de tests unitaris i d'integració per a desplegar-ho immediatament al mínim canvi introduït.

- **Desacoblament amb la resta de serveis:** un problema en el servei no ha d'impactar a l'operativa general de la plataforma i viceversa.

- **Desenvolupat per un equip:** com la filosofia és generar serveis petits amb codi font reduït ha de poder ser desenvolupat, desplegat i mantingut per un únic equip.

- **Desplegament independent:** les noves versions generades s'han de poder desplegar a cada un dels entorns sense necessitat de coordinar-se amb altres equips de desenvolupament.

A banda de les millores indiscutibles que introdueixen les arquitectures d'aplicacions basada en microserveis respecte a les arquitectures monolítiques, cal dir, que la seva popularitat s'ha vist accelerada recentment per la irrupció dels contenidors. Els contenidors són una tecnologia que permet empaquetar i distribuir una aplicació amb els elements mínims necessaris (versió minimalista del sistema operatiu, llibreries, configuracions, etc...) per a que pugui ser executada en qualsevol plataforma compatible. Aquestes plataformes, reben el nom d'orquestradors de contenidors on la més estesa amb diferència és Kubernetes.

Un cop descrites les metodologies i tecnologies que han contribuït majoritàriament en les millores dels processos de desenvolupament de programari i administració de sistemes, se'ns

[3] *Defecte trobat en una part del codi d'una aplicació*

planteja el següent dubte: hi ha algun estàndard definit per aplicar d'una manera coordinada totes les bones pràctiques anteriors? La resposta es si i s'anomena **GitOps**.

GitOps és un conjunt d'eines i estàndards per implementar la cultura DevOps sobre aplicacions *cloud-native* basades en arquitectura de microserveis, tenint com a única font de dades, un repositori de control de versions de codi Git.

Serà, durant el desenvolupament d'aquest treball de final de grau, on s'estudiarà i es posarà en pràctica aquest nou model d'administració de sistemes i desplegament d'aplicacions.

1.1. Justificació del treball

La motivació per a realitzar aquest projecte radica en el fet d'aprendre tecnologies i metodologies utilitzades actualment, o que ho seran en un futur, per la major part d'organitzacions que es dediquen al sector de la informàtica. D'aquesta manera, s'aprofitarà la investigació realitzada durant el treball de final de grau per a complementar i ampliar coneixements tècnics que poden ser de gran utilitat durant la trajectòria professional.

A més a més, l'elecció de la temàtica fa possible que aquesta investigació i implementació sigui aplicable a qualsevol empresa de software o departament TIC independentment del llenguatge de programació que utilitzin o plataforma de cloud, ja que, en totes les eleccions tecnològiques que s'hagin de realitzar, s'optarà per aquelles que ofereixin major flexibilitat i adaptabilitat.

Finalment, i no menys important, es vol trobar els límits i/o les deficiències de totes les eines que s'explorin durant en el desenvolupament del treball per a proposar millores futures o bé tenir un criteri sòlid per a poder opinar amb dades objectives sobre la seva idoneïtat o no depenent de cada cas d'ús.

1.1.1. Objectius del treball

El treball de final de grau té com a objectius estudiar la metodologia de processos GitOps i l'ecosistema que l'envolta, dissenyar l'arquitectura de la infraestructura necessària i finalment implementar la solució dissenyada incloent el desplegament d'una aplicació de prova per validar-ne el funcionament. Tanmateix, es pretén establir un plataforma que serveixi com a base per a agilitzar la incorporació de noves aplicacions. Respecte als objectius, es classificaran en dos tipus: personals i funcionals. Primerament, passarem a enumerar els objectiu personals:

- Estudiar i aplicar els conceptes de la metodologia GitOps.
- Estudiar el funcionament i l'ús d'una plataforma de cloud pública.
- Estudiar i aplicar tests previs al desplegament d'infraestructura.
- Estudiar els components i l'ús de l'orquestrador de contenidors Kubernetes.
- Estudiar i utilitzar les eines que formen l'ecosistema CI/CD en metodologies GitOps.
- Aprendre nous llenguatges de programació.

A banda dels objectius personals, més basats en l'aprenentatge, també es definiran com a objectius uns requisits funcionals per garantir que la solució implementada compleix amb els principis indicats en l'apartat Introducció d'aquest document:

- Els recursos que s'afegeixin a la plataforma han d'estar prèviament definits en un repositori de control de versions Git, aplicant conceptes *Infrastructure as Code*[4] i *Pipeline as Code*[5].
- Tant la infraestructura com les aplicacions han de tenir tests definits seguint els principis d'automatització i CI/CD.
- Qualsevol recurs afegit a la plataforma ha de ser reproduïble, re-generable i fàcil de destruir. Aquest aspecte és clau per a reduir la despesa del proveïdor de cloud públic.
- La infraestructura i les aplicacions han de complir amb les propietats d'escalabilitat, alta disponibilitat, seguretat i privacitat de les dades.
- En cas de dubte entre productes de prestacions similars, s'optarà per solucions *cloud-native*[6] abans que les *on-premise*[7].

A la finalització de cada una de les quatre PACs planificades es revisaran els objectius per comprovar que s'estan complint. En cas que durant el desenvolupament del treball aparegui algun risc inesperat que posi en perill algun punt, es proposarà modificar mínimament els objectius marcats inicialment.

1.2. Planificació del treball

Aprofitarem els objectius marcats en l'apartat anterior per a definir, a alt nivell, les fites més importants que s'hauran d'assolir per a complir amb les dates d'entrega establertes en el pla docent del treball de final de grau. A banda de les fites generals, també llistarem dins de cada una d'aquestes, les tasques individuals que s'hauran de realitzar per donar-les per completades. Així, dividirem el projecte en les fases següents:

1.2.1. Pla de Treball

En aquesta primera fase, de descobriment i lectura inicial sobre els nous conceptes, es prepararà el pla de treball que es seguirà durant el desenvolupament del treball:

- Lectura i visualització de vídeos conceptuals sobre GitOps.
- Presentació de la proposta al tutor.
- Preparació del format de la plantilla de la memòria.
- Redacció de la introducció del TFG.
- Redacció de la justificació, objectius i planificació del TFG.
- Redacció de la planificació i diagrama de Gantt.

[4] *Tècnica que consisteix en definir la infraestructura IT en codi utilitzant un llenguatge de programació específic.*
[5] *Tècnica que consisteix en definir les pipelines de desplegament d'aplicacions en codi utilitzant un llenguatge de programació específic.*
[6] *Tipus d'aplicació dissenyada per a ser executada exclusivament sobre plataformes de cloud privades, públiques o híbrides.*
[7] *Es refereix a les aplicacions o serveis que s'instal·len sobre servidor ubicats en centre de dades propietaris.*

1.2.2. Investigació sobre l'ecosistema GitOps

Una vegada presentat el pla de treball, es dedicarà el temps en investigar i estudiar detalladament com es pot posar en pràctica la metodologia GitOps. Durant aquesta fase, s'avaluaran les diferents opcions d'eines disponibles i les plataformes de cloud, seleccionant-ne les més adients. Les tasques que s'hauran de realitzar en aquesta fase són les següents:

- Introducció al concepte GitOps
- Identificació i estudi de les tècniques requerides: Infrastructure as Code (IaC) i CI/CD.
- Identificació i estudi de les tecnologies requerides: Git i Kubernetes.
- Estudi i avaluació dels tipus de desplegaments que ofereix GitOps.
- Avantatges que proporciona les metodologies GitOps.
- Avaluació i selecció de les eines a utilitzar per la part pràctica: Git, IaC, CI/CD, Kubernetes i plataforma cloud.

1.2.3. Preparació de l'entorn local

Després d'haver adquirit coneixement sobre els conceptes d'aquesta metodologia i haver seleccionat les tecnologies que s'utilitzaran, es passarà a donar d'alta els serveis necessaris i a preparar l'estació de treball executant les següents tasques:

- Alta del servei a la plataforma cloud escollida.
- Alta del servei a la plataforma Git escollida.
- Instal·lació de les eines requerides en l'estació de treball per a implementar la solució.

Probablement, a partir d'aquesta fase i posteriors serà quan es requereixi més dedicació i en la què pot aparèixer algun risc que faci modificar els objectius marcats inicialment.

1.2.4. Disseny, implementació i desplegament de la infraestructura

En aquesta fase es dissenyarà, s'implementarà i desplegarà la infraestructura base sobre la qual s'executarà l'aplicació que es desenvoluparà per a demostrar l'eficiència dels desplegaments utilitzant els principis GitOps. Tanmateix, la pròpia infraestructura seguirà la mateix filosofia que l'aplicació en temes de desplegament. Entre les activitat que es preveuen durant aquesta etapa tenim:

- Definició dels repositori de codi Git per emmagatzemar el arxius de configuració de la infraestructura.
- Definició dels entorns a desplegar.
- Disseny, implementació i desplegament de la topologia de xarxa en la plataforma cloud.
- Disseny, implementació i desplegament de l'arquitectura del clúster de Kubernetes.
- Definició i implementació dels processos d'integració i desplegament continu a seguir per la infraestructura.
- Desplegament de l'eina per a gestionar el procés de desplegament continu de les aplicacions.

1.2.5. Cas pràctic: desenvolupament i desplegament d'una aplicació

Un cop completada la fase anterior, arribarà el moment d'utilitzar la infraestructura creada anteriorment. Durant aquesta etapa, es desenvoluparà una aplicació web de proves que servirà per validar el correcte funcionament de tots els elements creats anteriorment. Les tasques identificades dins d'aquesta fase són les que es detallen a continuació:

- Creació dels repositoris Git per l'aplicació.
- Aprendre el llenguatge de programació escollit per desenvolupar l'aplicació.
- Dissenyar l'arquitectura de l'aplicació.
- Implementar l'aplicació dissenyada.
- Afegir els tests unitaris per garantir el funcionament de l'aplicació en els processos de CI/CD.
- Generar el diagrama de flux de desplegament de codi.
- Definició i implementació dels processos d'integració i desplegament continu a seguir per l'aplicació.
- Definició i creació dels entorns de l'aplicació.
- Creació dels entorns de desplegament.
- Creació dels objectes de Kubernetes per desplegar l'aplicació.
- Registre domini i crear registres DNS per l'aplicació.
- Desplegament de l'aplicació de prova aplicant GitOps.

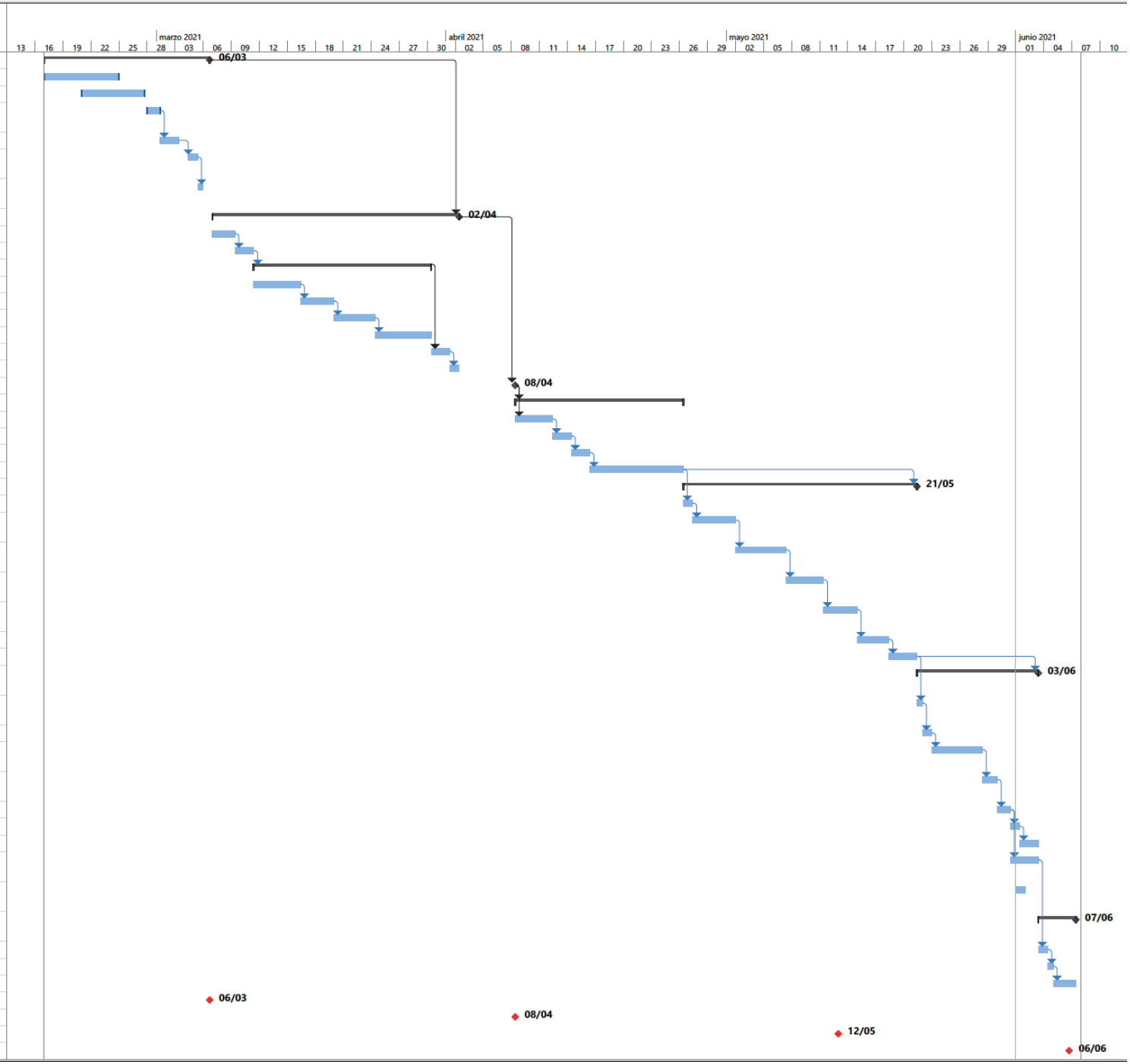
1.2.6. Conclusions i preparació lliurament final

Finalment, després d'haver implementat la solució, serà el moment de fer balanç i identificar les millores que introduiríem a la solució que no han pogut ser implementades per manca de temps. També es reservarà espai dins d'aquesta fase per a finalitzar la memòria del treball de final de grau i per preparar la presentació virtual. Les tasques individuals que s'engloben dins d'aquesta fase són:

- Redacció de conclusions i propostes de millores.
- Finalitzar el document de la memòria del TFG.
- Crear i preparar la presentació per exposar el TFG.
- Gravar el vídeo per exposar el TFG.
- Preparar la defensa virtual.

Per acabar, representarem la planificació de les tasques anteriors utilitzant un diagrama de Gantt, tenint present les dates definides en el pla docent per l'entrega de cada una de les quatre PACs.

Id	Nombre de tarea	Duración	Comienzo	Fin
1	**1 Pla de treball**	**18 días**	**17/02/21**	**06/03/21**
2	1.1 Lectura de continguts sobre GitOps	8 días	17/02/21	24/02/21
3	1.2 Presentació de la proposta al tutor	7 días	21/02/21	27/02/21
4	1.3 Preparació del format de la plantilla de la memòria	1 día	28/02/21	01/03/21
5	1.4 Redacció de la introducció del TFG	2 días	01/03/21	03/03/21
6	1.5 Redacció de la justificació i objectius del TFG	1 día	04/03/21	05/03/21
7	1.6 Redacció de la planificació i diagrama de Gantt	1 día	05/03/21	06/03/21
8	**2 Investigació sobre l'ecosistema GitOps**	**26 días**	**07/03/21**	**02/04/21**
9	2.1 Què es GitOps?	2 días	07/03/21	09/03/21
10	2.2 Principis a seguir	2 días	09/03/21	11/03/21
11	2.3 Tècniques i tecnologies requerides	19 días	11/03/21	30/03/21
12	2.3.1 Git	5 días	11/03/21	16/03/21
13	2.3.2 Infrastructure as Code	4 días	16/03/21	20/03/21
14	2.3.3 CI/CD	4 días	20/03/21	24/03/21
15	2.3.4 Kubernetes	6 días	24/03/21	30/03/21
16	2.4 Mètodes de desplegament	2 días	30/03/21	01/04/21
17	2.5 Beneficis d'implantació	1 día	01/04/21	02/04/21
18	**3 Selecció de tecnologies**	**5 días**	**03/04/21**	**08/04/21**
19	**4 Preparació de l'entorn de treball**	**18 días**	**08/04/21**	**26/04/21**
20	4.1 Alta a la platforma cloud	4 días	08/04/21	12/04/21
21	4.2 Creació compte al servei de Git	2 días	12/04/21	14/04/21
22	4.3 Bucket per gestionar infraestructura	2 días	14/04/21	16/04/21
23	4.4 Instal·lació eines a l'equip local	10 días	16/04/21	26/04/21
24	**5 Disseny i implementació de la plataforma**	**25 días**	**26/04/21**	**21/05/21**
25	5.1 Creació del repositori Git	1 día	26/04/21	27/04/21
26	5.2 Disseny de l'arquitectura en la plataforma de cloud pública	5 días	27/04/21	02/05/21
27	5.3 Implementació i desplegament de la infrastructura en la plataforma cloud	5 días	02/05/21	07/05/21
28	5.4 Disseny de l'arquitectura del clúster de Kubernetes	4 días	07/05/21	11/05/21
29	5.5 Implementació i desplegament del clúster de Kubernetes	4 días	11/05/21	15/05/21
30	5.6 Disseny de les eines per CI/CD	3 días	15/05/21	18/05/21
31	5.7 Implementació de les eines per CI/CD	3 días	18/05/21	21/05/21
32	**6 Cas pràctic: desenvolupament i desplegament d'aplicacions**	**13 días**	**21/05/21**	**03/06/21**
33	6.1 Anàlisi del llenguatge de programació a utitlizar	1 día	21/05/21	22/05/21
34	6.2 Disseny de l'arquitectura de l'aplicació	1 día	22/05/21	23/05/21
35	6.3 Implementació d'una aplicació de prova	5 días	23/05/21	28/05/21
36	6.4 Implementació dels processos d'integració continúa	2 días	28/05/21	30/05/21
37	6.5 Disseny del diagrama de desplegament	1 día	30/05/21	31/05/21
38	6.6 Creació dels entorns de desplegament	1 día	31/05/21	01/06/21
39	6.7 Creació dels objectes de Kubernetes	2 días	01/06/21	03/06/21
40	6.8 Desplegament de l'aplicació de prova aplicant GitOps	3 días	31/05/21	03/06/21
41	6.9 Alta dels domini DNS i creació de registres	1 día	01/06/21	01/06/21
42	**7 Conclusions i preparament lliurament final**	**4 días**	**03/06/21**	**07/06/21**
43	7.1 Redacció conclusions	1 día	03/06/21	04/06/21
44	7.2 Revisar la memòria del TFG	1 día	04/06/21	05/06/21
45	7.3 Preparar la presentació powerpoint	2 días	05/06/21	07/06/21
46	**8 PAC1**	**18 días**	**17/02/21**	**06/03/21**
47	**9 PAC2**	**32 días**	**07/03/21**	**08/04/21**
48	**10 PAC3**	**34 días**	**09/04/21**	**12/05/21**
49	**11 Entrega Final**	**25 días**	**13/05/21**	**06/06/21**



2. Introducció a GitOps

Per tal de tenir una visió més detallada sobre què significa el terme GitOps es realitzarà un estudi profund sobre aquest concepte per obtenir el coneixement necessari abans de dissenyar i implementar la solució que es durà a terme en aquest treball.

2.1. Què és GitOps?

GitOps és un model operacional, introduït l'any 2017 per la companyia Weaveworks, que defineix un conjunt de pràctiques a seguir sobre com desplegar, configurar, monitoritzar i administrar una aplicació i la seva infraestructura a partir d'una font de dades única emmagatzemada en un o més repositoris de control de versions Git.

Tot i que aquesta metodologia de treball podria ser aplicable a qualsevol projecte de desenvolupament de programari, amb algunes adaptacions, el terme GitOps està indicat principalment a aplicacions *cloud-native* basades en arquitectura de microserveis executant-se sobre l'orquestrador de contenidors Kubernetes.

El conjunt de principis definits dins de la metodologia GitOps ajuden a estandarditzar els processos dins dels departaments de tecnologies d'informació ja que el *workflow*[8] seguit pel desplegament d'infraestructura és idèntic al de les aplicacions. D'una banda, els equips d'operacions disposen d'un model unificat de bones pràctiques aplicables a la gestió i desplegament de la infraestructura i, de l'altra, els equips de desenvolupament gaudeixen d'uns processos automatitzats pel desplegament de noves versions, basades en tècniques de CI/CD, que permeten disposar ràpidament en l'entorn de treball desitjat (integració, pre-producció, producció, etc...) el nou codi desenvolupat.

2.2. Principis a seguir

Per tal de posar en pràctica la metodologia GitOps és obligatori l'aplicació de les següents propietats a l'hora de dissenyar i implementar una solució tecnològica:

- **Descripció declarativa[9] del sistema:** la plataforma sencera ha d'estar programada fent servir pràctiques declaratives mitjançant l'ús d'eines específiques. Aquest concepte consisteix a definir, mitjançant codi, l'estat final que volem obtenir sense importar-nos el com, assegurant que qualsevol canvi manual sigui revertit a l'estat desitjat al tornar a ser aplicada la configuració emmagatzemada en el repositori.

- **L'estat desitjat ha d'estar versionat en Git:** la declaració dels components de la infraestructura i de l'aplicació mitjançant codi hauran d'estar emmagatzemats en un sistema de control de versions (Git) el qual es considerarà com l'únic punt de referència des del qual es gestionarà l'administració de la plataforma. El fet d'utilitzar Git com a SCM (*source code management*) ens facilita la restauració del sistema en un punt

[8] *Flux de treball automatitzat on cada pas es troba perfectament definit i està acceptat per tots els empleats d'una organització.*
[9] *Paradigma de programació que busca obtenir un resultat concret mitjançant l'abstracció de processos sense tenir en compte com es realitza ni els passos que es necessiten. El tipus contrari s'anomena imperatiu.*

concret en cas que aparegui un problema derivat d'un canvi introduït. Tanmateix, el fet d'utilitzar Git com a control de versions ens permet aprofitar els avantatges que aquesta eina ofereix a nivell de seguretat com, per exemple: traçabilitat sobre l'autoria dels canvis, gestió centralitzada de l'autentificació d'usuàries/usuaris o ús per defecte de protocols de comunicació segurs (SSH/HTTPS més MFA[10])

- **Els canvis aprovats s'han de desplegar automàticament:** la introducció de canvis dins de la plataforma s'ha de realitzar a través dels *workflows* existents en Git que permeten modificar una branca a partir dels mecanismes anomenats *Pull Request* o *Merge Request* depenent del producte utilitzat. Quan una persona de l'equip tècnic vulgui introduir un canvi crearà una *Pull Request* o *Merge Request* la qual haurà de ser validada i aprovada per un o més companys abans que es pugui afegir a la branca destí. Un cop els canvis s'hagin aprovat i es trobin a la branca destí aquests es desplegaran automàticament a l'entorn seleccionat. Amb aquesta pràctica, obtindrem més control sobre els canvis introduïts, millorarem la seguretat de la plataforma al no requerir la introducció de credencials i reduirem la possibilitat d'errors humans al disposar d'un procés totalment automàtic de desplegament.

- **Els agents han de garantir l'estat desitjat i alertar en cas de divergències:** tenir en tot moment l'estat desitjat declarat en el repositori Git és un element clau en l'ús de metodologies GitOps. Com s'ha explicat anteriorment, es delega tota la responsabilitat de tenir un sistema en un estat concret a la definició que hi hagi en el repositori Git que actua com a "*source of truth*"[11], és a dir, com únic punt sobre el que podem confiar completament. Per garantir que aquesta definició s'està complint s'han d'establir mecanismes, mitjançant l'ús d'agents de software, que puguin restaurar automàticament el sistema a l'estat desitjat i, en cas que no puguin realitzar, hauran de notificar a l'equip tècnic sobre la divergència trobada.

Un cop descrites les propietats anteriors podem comprovar que es tracten de conceptes purament conceptuals. Per aquesta raó, en l'apartat següent passarem a indicar quines són les tècniques específiques que s'hauran d'aplicar per a complir amb els requeriments indicats.

2.3. Tècniques i tecnologies requerides

A partir dels conceptes teòrics que s'han d'aplicar en les solucions tecnològiques per introduir l'ús de metodologies GitOps tenim a disposició tècniques i aplicacions existents que seran imprescindibles per fer possible la seva implantació, concretament: Git, Infrastructure as Code, CI/CD i Kubernetes. A continuació, s'analitzarà detalladament cada un d'aquests components.

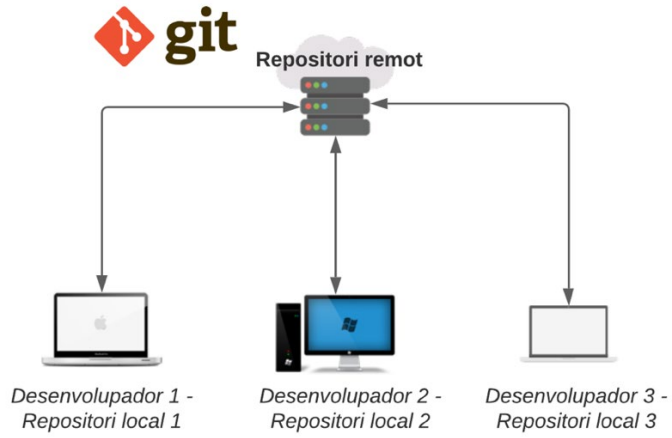
2.3.1. Git

Git és un sistema de control de versions distribuït de codi obert, dissenyat per Linus Torvalds l'any 2005, utilitzat principalment en projectes de desenvolupament de programari que ofereix

[10] *Sigles de multi-factor authentication, també conegut com a 2FA (two-factor authentication). És una capa addicional de seguretat per l'autenticació d'usuaris/usuàries que valida un codi temporal i aleatori generat per una aplicació tercera vinculada al compte del propi.*
[11] *Font de la veritat. És un punt centralitzat de consulta que garanteix l'autenticitat de la informació que emmagatzema.*

traçabilitat i un registre històric sobre les modificacions introduïdes en el codi font durant tot el seu cicle de vida.

Els sistemes de control de versions distribuïts (DVCS), com Git o Mercurial, estan basats en una arquitectura distribuïda on cada persona disposa d'una còpia completa en local de la història del projecte que també es troba en un servidor remot. Aquest fet, redueix el risc de pèrdua d'informació ja que, implícitament, estem realitzant còpies de seguretat en múltiples ubicacions i permet continuar desenvolupant encara que es perdi la connexió amb el servidor remot. En canvi, els sistemes de control de versions centralitzats (CVCS), com Subversion, tenen una arquitectura client-servidor on tota la informació històrica del projecte només quedarà registrada en el servidor.



Il·lustració 2 - Arquitectura git. Font: pròpia

2.3.1.1. Cicle de vida

El cicle de vida de git es pot dividir en 4 fases, explicades a continuació:

1- **Directori de treball o *working directory***: es tracta del directori local on es troba el projecte en les estacions de treball. Si el directori s'ha inicialitzat o bé és el resultat de la clonació d'un repositori remot, trobarem un directori ocult anomenat `.git` que contindrà la meta informació del repositori.

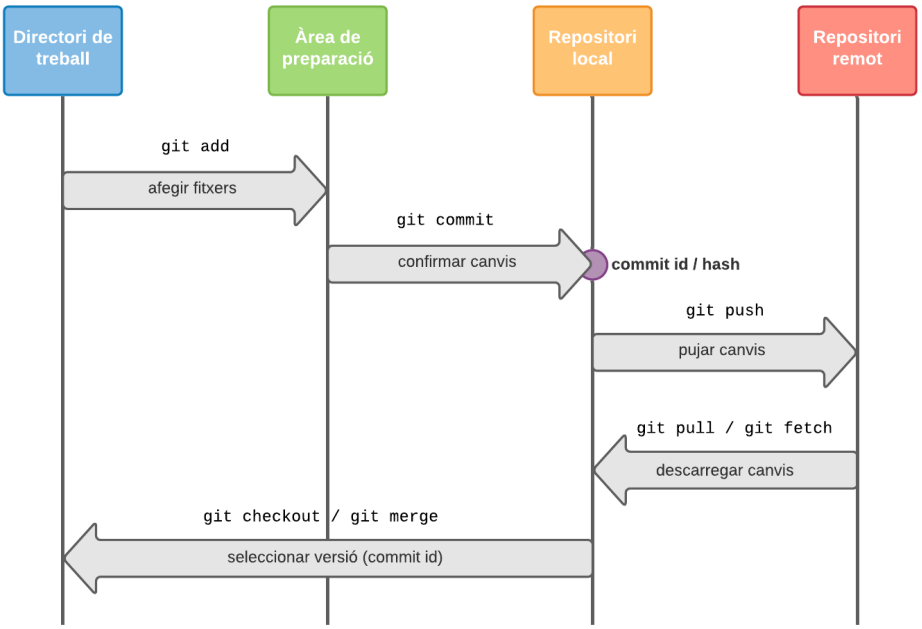
2- **Àrea de proves o *stage area*:** després de modificar o crear nous fitxers en l'àrea de treball, seleccionarem aquells candidats que considerem que es troben preparats per a ser inclosos en l'històric del control de versions del repositori.

3- **Repositori local o *local repository*:** els fitxers candidats que es trobin en l'àrea de proves seran emmagatzemat en el repositori local, generant un nou *snapshot*[12], assignant-los un identificador anomenat "*commit id*". Aquest identificador únic, de 40 caràcters SHA-1[13], també es coneix com a "*commit hash"* i serveix per identificar de manera inequívoca cada una de les versions emmagatzemades en el repositori git.

[12] *Còpia de la totalitat dels arxius existents en un moment puntual.*
[13] *Secure Hash Algorithm versió 1. És una funció hash criptográfica de 160bits que s'utilitza per a calcular un valor de comprovació única.*

4- **Repositori remot o *remote repository*:** aquest últim pas serveix per a sincronitzar l'estat del repositori local amb el repositori remot, clau per aconseguir un dels objectius de git: fomentar la col·laboració entre la comunitat de desenvolupament.

El diagrama següent mostra les fases que segueix un fitxer des de que es crea fins que s'emmagatzema en l'històric de versions del repositori remot:



Il·lustració 3 - Fases git. Font: pròpia

2.3.1.2. Branques

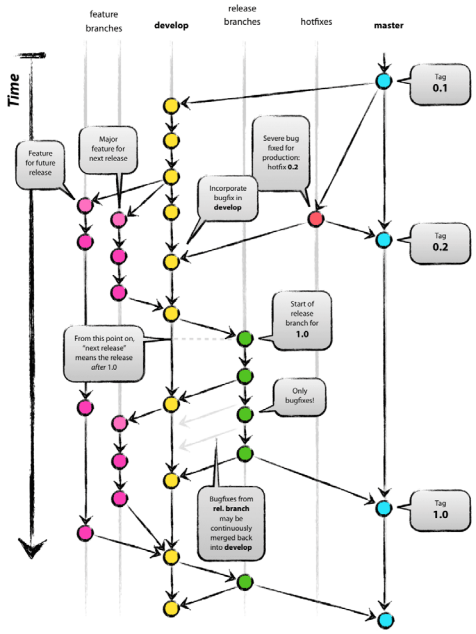
Una de les característiques més potents que disposa git és el concepte de **branca** (*branch*). Com s'ha explicat en el punt anterior, cada vegada que git emmagatzema un nou *snapshot* se li assigna automàticament un identificador SHA-1 al qual se li afegeix meta informació utilitzada per temes de traçabilitat (hora de confirmació del canvi, autor, missatge explicatiu, etc...). Doncs bé, una branca és simplement un apuntador mòbil a un dels identificadors existents en el repositori, que es va actualitzant automàticament a mesura que s'afegeixen nous *commits*[14]. Un repositori ha de tenir com a mínim una branca, que es crea automàticament al inicialitzar un repositori amb el nom ***main,*** tot i que pot ser substituïda posteriorment per una altra. La creació de noves branques dona lloc a un procés anomenat ***merge*** que consisteix en integrar el contingut de dues branques independents en una única.

Els avantatges que ens ofereixen les branques en els projectes de desenvolupament són els següents:

- Augmentar la velocitat d'entrega de valor a les persones usuàries al poder treballar simultàniament en el desenvolupament de funcionalitats.
- Reduir la complexitat de la gestió de pegats per solucionar *bugs*.
- Potenciar l'experimentació.

[14] *És l'acció de guardar un o més arxius en l'històric del repositori generant un snapshot i, conseqüentment, un identificador SHA-1.*

- Reduir l'impacte que suposa el canvi de context pel personal tècnic.
- Crear estratègies de desplegaments basades en branques com *git-flow*[15] *o* el propi GitOps.



Il·lustració 4 - Diagrama estratègia git-flow
Font: https://nvie.com/posts/a-successful-git-branching-model/

2.3.1.3. Operativa

Totes les funcionalitats descrites anteriorment es poden utilitzar instal·lant l'eina de línia de comandes **git** en el sistema operatiu, descarregant-la en aquest pàgina https://git-scm.com/ o bé utilitzant un gestor de paquets com *yum*, *dnf*, *brew* o *apt*. Evidentment, a banda de la instal·lació d'aquesta utilitat, també necessitarem configurar l'autenticació necessària per a poder dur a terme certes operacions en servidors remots.

A continuació, descriurem algunes de les operacions més habituals que podem realitzar amb la utilitat git:

- **Configuració d'un repositori**
  - **git init:** crea un nou projecte de git a partir del directori actual. Al executar aquesta instrucció, es crea un directori ocult `.git` amb la meta informació del repositori.
  - **git clone:** a partir d'un repositori remot ja existent, es clona el seu contingut en un directori local generant l'estructura necessària (subdirectori ocult *.git* i connexió remota amb nom *origin*)
  - **git config:** s'utilitza per a configurar les propietats del repositori modificant l'arxiu *.gitconfig*.

- **Gestió de continguts**
  - **git add:** afegeix un fitxer nou o modificat del directori de treball a l'àrea de preparació.
  - **git rm:** elimina arxius de l'àmbit de seguiment que realitza git.

[15] *Metodologia de desplegament de codi basat en les següents branques de git: master, develop, feature, hotfix i release.*

  - **git commit:** confirma els canvis afegits a l'àrea de preparació generant un nou *snapshot* i *commit id*.
  - **git checkout:** carrega en el directori de treball el contingut d'un *snapshot* del repositori podent ser una branca o un identificador.
  - **git revert:** permet desfer canvis realitzats en el repositori, generant un nou *snapshot*, per tal de preservar la integritat en l'historial de revisions.

- **Eines informatives:**
  - **git log:** mostra l'historial del projecte a partir dels *snapshots* confirmats.
  - **git status:** mostra per pantalla un resum de l'estat del directori de treball i de l'àrea de preparació.
  - **git diff:** mostra per pantalla les diferències que presenta un mateix arxiu en dues ubicacions diferents (branca, repositori remot, tag, etc..)

- **Sincronització**
  - **git fetch:** descarrega *commits*, *snapshots*, *tags*, etc... d'un repositori remot al repositori local.
  - **git push:** puja el contingut del repositori local al repositori remot.
  - **git pull:** és la combinació de dues operacions: *git fetch* i *git merge*. La seva acció uneix els canvis presents en la branca del repositori remot amb els de la branca del repositori local.
  - **git remote:** permet gestionar la configuració d'un repositori remot com pot ser la URL o l'alies que té assignat.

- **Gestió de branques**
  - **git branch:** aquesta instrucció permet crear, llistar, canviar el nom o eliminar branques.
  - **git merge:** uneix el contingut de dues branques generant un nou *commit* amb els canvis fusionats.

Respecte als repositoris remots de git, que actuen com a punt d'unió entre els col·laboradors, existeixen diverses solucions comercials en versió *on-premise* i *SaaS*. L'elecció d'aquest component es tractarà en apartats posteriors on s'avaluaran les diferents propostes que hi ha disponibles.

2.3.2. Infrastructure as Code

El concepte **Infrastructure as Code** (IaC) consisteix en gestionar i aprovisionar la infraestructura IT (servidors, xarxes, base de dades, emmagatzematge, etc...) mitjançant de la definició dels recursos en codi a través d'arxius de configuració. Aquesta pràctica és l'evolució de l'administració de sistemes tradicional que contribueix en reduir les intervencions manuals i l'execució de tasques repetitives.

2.3.2.1. Tipus d'IaC

Com en el cas dels llenguatges de programació, el concepte Infrastructure as Code es pot aplicar utilitzant eines de dos tipus:

- **Declaratives:** les eines declaratives estan dissenyades per a desenvolupar el codi, definint quin resultat volem obtenir sense importar-nos el com, és a dir, és responsabilitat de l'eina deixar el sistema en l'estat desitjat.

- **Imperatives:** a diferència del cas anterior, les eines imperatives necessiten que es defineixin les instruccions exactes per obtenir l'estat desitjat i que aquestes es trobin en l'ordre adequat. En aquest cas, s'ha de tenir present que un pas pot tenir dependències d'un altre així que s'ha de vigilar aquestes situacions.

Sembla doncs que resulta més senzill i pràctic l'ús d'eines declaratives en comptes d'imperatives perquè redueixen complexitat a la gestió de la infraestructura.

2.3.2.2. Avantatges

Aquesta pràctica intenta solucionar problemes històrics identificats en la gestió d'infraestructura IT i, alhora, aprofitar al màxim els beneficis que aporta la computació al núvol respecte a l'escalabilitat, gestió de costos i disponibilitat. Així, podem identificar els següents beneficis que obtenim si utilitzem IaC com a mètode preferit per a administrar infraestructura:

- **Velocitat de desplegament:** un cop estigui codificada la infraestructura serà molt ràpid replicar aquesta configuració per a crear un entorn nou o, fins i tot, re-generar la plataforma sencera en cas de desastre.

- **Reducció de costos:** d'una banda, es redueix la càrrega de treball en administració de la plataforma, i, de l'altra, es té un major control sobre els recursos que es troben desplegats permetent obtenir una estimació acurada sobre el cost en infraestructura.

- **Reducció d'errors humans:** el codi generat, ja sigui per crear infraestructura o modificar-ne l'existent, ha de passar pels controls interns de l'organització que proporcionen les eines de control de versions (*Pull Request* o *Merge Request*). Així, la persona que proposi un canvi haurà d'esperar a rebre l'aprovació d'un o més companys abans de poder afegir els canvis en la branca utilitzada per la gestió de l'entorn. A més a més, un cop validat, els canvis s'aplicaran automàticament sense la necessitat de realitzar intervencions manuals.

- **Traçabilitat dels canvis:** una de les característiques d'IaC és que el codi generat ha d'estar emmagatzemat en un sistema de control de versions, per tant, qualsevol canvi introduït quedarà registrat i es podrà identificar l'autor.

- **Major consistència:** al disposar de la definició de la infraestructura en codi, garantim que qualsevol canvi manual introduït serà revertit en la propera execució de l'eina IaC utilitzada.

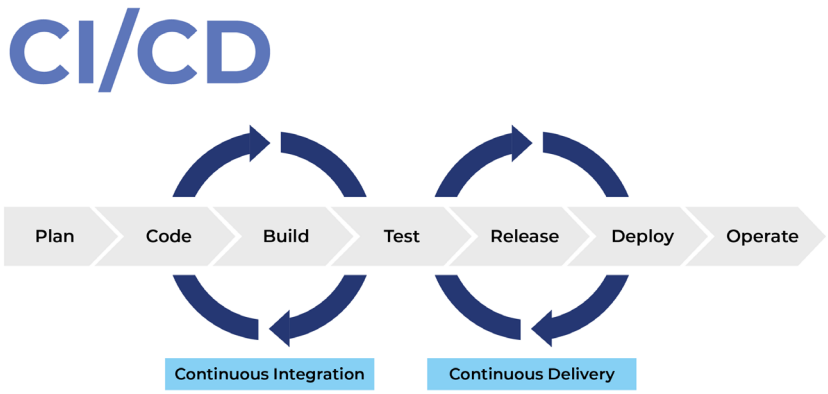
- **Documentació auto-generada:** el codi generat pot ser utilitzat com documentació de la plataforma ja que, qualsevol amb els coneixements necessaris, pot fer-se una idea dels recursos que es troben desplegats simplement llegint el codi.

- **Tests automatitzats:** al tractar-se de codi, es poden aplicar les mateixes pràctiques de tests utilitzades en el desenvolupament d'aplicacions (test unitaris, tests d'integració, etc...) reduint així els possibles riscos a l'hora d'aplicar canvis en els entorns productius.

Tot i que la gestió de la infraestructura utilitzant codi no és exclusiu de plataformes cloud, el cert és que la seva extensió s'ha fet més popular gràcies a les característiques específiques que aquestes proporcionen. El fet de no haver de configurar hardware i la quantitat de serveis que les plataformes cloud ofereixen fan que l'IaC esdevingui un element pràcticament imprescindible en qualsevol arquitectura moderna.

2.3.3. CI/CD

Els termes *Continuous Integration* i *Continuous Delivery*, que conformen les sigles CI/CD, són un conjunt de pràctiques molt esteses en la filosofia DevOps que tenen com a objectiu principal la distribució freqüent de noves versions d'aplicacions a partir d'aplicar tècniques d'automatització en totes les àrees que participen en els cicles de desenvolupament de programari: el propi desenvolupament, el tests sobre el codi, les *pipelines* de desplegament, la configuració de l'aplicació i la gestió de la infraestructura. La imatge següent mostra clarament quines accions tenen lloc a cada un dels termes CI/CD:



Il·lustració 5 - Etapes de CI/CD
Font: https://edge.siriuscom.com/_wss/clients/509/assets/503/CI-CD-graphic.PNG

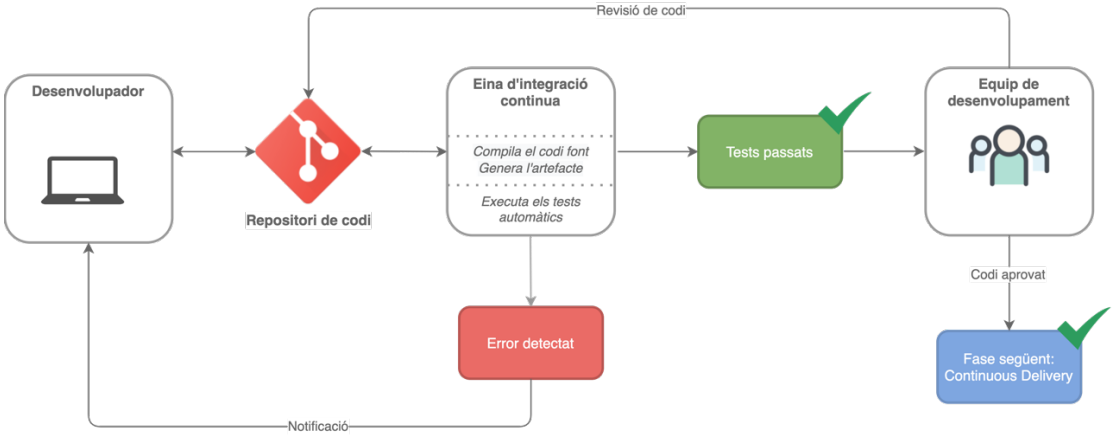
2.3.3.1. Continuous Integration (CI)

En els projectes de desenvolupament actuals són molts els col·laboradors que poden estar contribuint simultàniament en un mateix repositori de codi per afegir noves funcionalitats. Per assegurar i validar la idoneïtat de cada canvi introduït per un desenvolupador o desenvolupadora en el codi font abans de poder ser afegit en la branca principal, primerament haurà de passar per un seguit de fases que garanteixin la qualitat del codi. Aquest conjunt de fases, on la majoria són tasques automàtiques, l'operació d'afegir els canvis en la branca principal s'anomena **integració continua** (CI).

A continuació, detallarem quins són els passos pels que ha de passar el codi nou abans que sigui integrat en la branca principal del repositori:

1- El desenvolupador o desenvolupadora, un cop té preparat el codi nou, obre una sol·licitud de canvi en el sistema de control de versions. Depenent la plataforma utilitzada, aquesta sol·licitud rep el nom de *Pull Request* o *Merge Request*.
2- L'eina d'integració continua té configurat el repositori i, quan detecta un canvi en aquest, descarrega el codi que ha de verificar i executa el conjunt de tests definits. Si cal, també compila el codi font o l'empaqueta per a construir un *artefacte*[16] per les fases posteriors.
3- Si algun dels passos definits falla, s'envia una notificació a la persona que ha proposat el canvi per a que corregeixi els problemes detectats. En cas contrari, s'espera una revisió manual del codi (*Code Review*) per part de l'equip de desenvolupament on es poden suggerir millores o identificar problemes potencials.
4- Un cop s'han passat tots el mecanismes de validació, la sol·licitud de canvi està preparada per a ser desplegada en un entorn.

El següent diagrama representa gràficament la seqüència descrita anteriorment:



Il·lustració 6 – Workflow d'integració continua. Font: pròpia

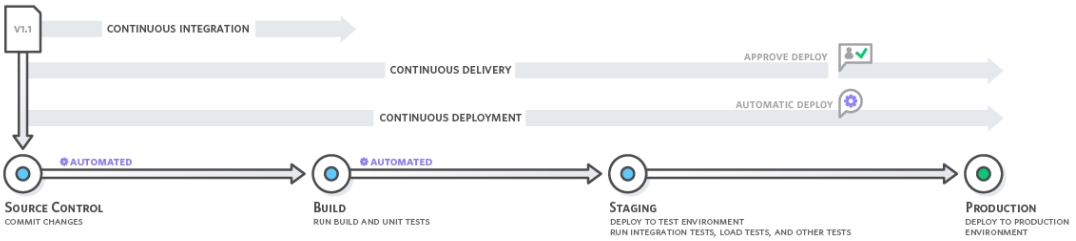
Els avantatges que s'obtenen al posar en pràctica la integració continua en el cicle de desenvolupament tenen repercussió en tota l'organització ja que, com el codi es va afegint amb més freqüència, fa possible que es redueixi el temps de lliurament de noves funcionalitats adaptant-se així a les necessitats del mercat amb major agilitat. Tanmateix, l'experiència de desenvolupament és veu beneficiada al obtenir un feedback instantani del treball realitzat mitjançant les notificacions automàtiques de l'eina d'integració continua. A més a més, com es necessita la revisió del codi d'algun company es potencia la transferència de coneixement passiva i es millora la comunicació

[16] *És el resultat del desenvolupament d'un producte, en forma de codi empaquetat o compilat, preparat per a ser executat en un entorn. El format pot ser de molts tipus: imatge de Docker, rpm, jar, tar.gz, zip, etc...*

2.3.3.2. Continuous Delivery / Continuos Deployment (CD)

Utilitzant la descripció del terme *Continuous Delivery*[17], que defineix Martin Fowler[18], podem dir que és una disciplina de desenvolupament software on la mentalitat ha de ser que qualsevol codi introduït ha de ser potencialment candidat a ser desplegat a l'entorn de producció en qualsevol moment. Aquesta afirmació implica que cada fase del cicle de desenvolupament ha de garantir el compliment d'uns requeriments de qualitats, evidentment automatitzats, per a que l'artefacte generat tingui la fiabilitat necessària per a ser desplegat a un entorn productiu sense generar incidències.

En l'espai temporal, podem considerar que la distribució continua (CD) és la fase justament posterior a la integració contínua (CI) i, com es pot sobreentendre, estan estretament relacionats. El fet de disposar d'una versió de codi preparada per ser lliurada no implica que aquesta hagi de ser desplegada automàticament ni de forma continuada als entorns productius ja que, en moltes ocasions, aquesta decisió es prendrà en funció de les necessitats del negoci o requeriments de clients. El concepte que posa en pràctica l'entrega de codi continuada en els entorns productiu es coneix com *Continuous Deployment* (CD). La següent imatge representa d'una manera molt gràfica fins a on arriba cada terme:



Il·lustració 7 - Fases de CI/CD.
Font: https://aws.amazon.com/es/devops/continuous-delivery/

El concepte de *pipeline*, esmentat en apartats anteriors, és un component essencial per a posar en pràctica tant la distribució continua com el desplegament continu en les organitzacions, on l'eina més estesa per implementar-ho s'anomena *Jenkins*[19]. Les *pipelines* permeten automatitzar els processos de desplegament de noves versions de les aplicacions en els entorns, evitant la intervenció manual i assegurant la repetibilitat del procés en cada execució. Recentment, com en el cas de la infraestructura, s'ha introduït el concepte *Pipeline as Code*[20] que comparteix les mateixes característiques que *Infrastructure as Code* on s'aposta per definir els passos de desplegament d'una aplicació en un fitxer de configuració fent servir el llenguatge de programació definit per l'eina utilitzada (en el cas de *Jenkins* s'utilitza *Groovy*).

[17] *Descripció obtinguda de la web https://martinfowler.com/bliki/ContinuousDelivery.html*
[18] *Martin Fowler és un prestigiós enginyer de software britànic que ha publicat diversos llibres sobre el desenvolupament àgil*
[19] *Jenkins és un projecte Java open source que permet implementar pràctiques de CI/CD en projectes de desenvolupament de software*
[20] *En aquest enllaç es troba explicat detalladament com generar pipelines declaratives en Jenkins*

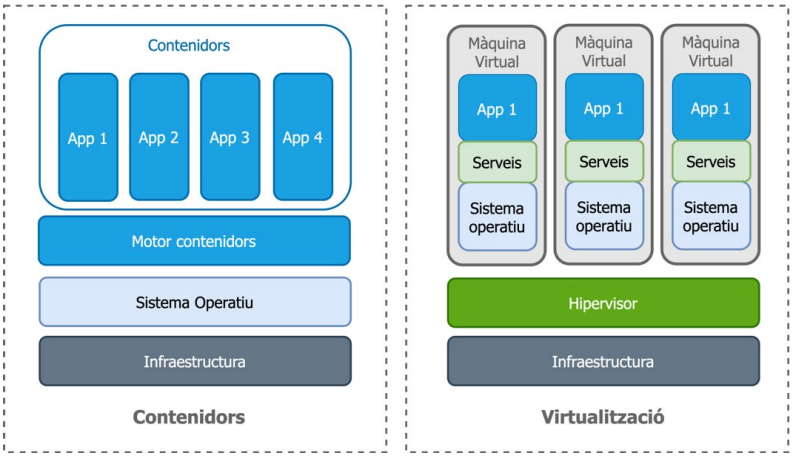
2.3.4. Kubernetes

Kubernetes és un orquestrador de contenidors de codi obert, alliberat per Google l'any 2014 després d'haver-ho utilitzat internament durant 15 anys, per a l'execució de càrregues de treball d'aplicacions pròpies. En aquesta definició inicial, apareixen dos conceptes que cal explicar primer per entendre el significat d'aquest producte: contenidors i orquestradors.

2.3.4.1. Contenidors

Un contenidor és una tecnologia que permet executar una aplicació i les seves dependències com a procés independent sobre qualsevol sistema operatiu que disposi del motor d'execució de contenidors. Per a que això sigui possible, cal que existeixi una imatge emmagatzemada en un repositori, conegut com a *registry*[21], a partir de la qual es podran instanciar un o més contenidors prèvia descarrega. Aquestes imatges de contenidors tenen el mínim imprescindible per a poder ser executades: instal·lació minimalista d'un sistema operatiu, llibreries de sistema, configuracions, el codi de l'aplicació i les seves dependències.

Tot i que sovint es descriuen els contenidors com un tipus de virtualització el cert és que són tecnologies diferents. Mentre que els contenidors són una abstracció de la capa d'aplicació que comparteixen el *kernel*[22] del sistema operatiu però com processos independents, la virtualització és una abstracció dels servidors físics on és necessària una capa addicional anomenada *hipervisor*[23] i la instal·lació completa del sistema operatiu. A més a més, les característiques dels contenidors fan que siguin elements més portables i lleugers que les màquines virtuals convertint-los en candidats ideals per arquitectures modernes de microserveis.



Il·lustració 8 - Contenidors vs Virtualització. Font: pròpia

La popularitat que han adquirit els contenidors és, en bona part, gràcies a Docker. Aquest projecte de codi obert va ser presentat l'any 2013 i va provocar una revolució en la manera com es van començar a desplegar les aplicacions gràcies a les propietats de portabilitat i

[21] *Repositori on es troben emmagatzemades les imatges dels contenidors amb format nom:etiqueta. El més utilitzat a l'actualitat és Docker Hub.*
[22] *Component fonamental d'un sistema operatiu que s'encarrega de la comunicació entre els programes i el maquinari.*
[23] *Software que s'instal·la en els servidors per habilitar la virtualització de sistemes operatius. Permet executar múltiples màquines virtuals sobre una única màquina física.*

immutabilitat que ofereix. Tot i això, cal dir, que existeixen alternatives a Docker com LXC (Linux Containers), rkt (CoreOS Rocket), Podman, containerd o Windows Hyper-V.

2.3.4.2. Orquestradors

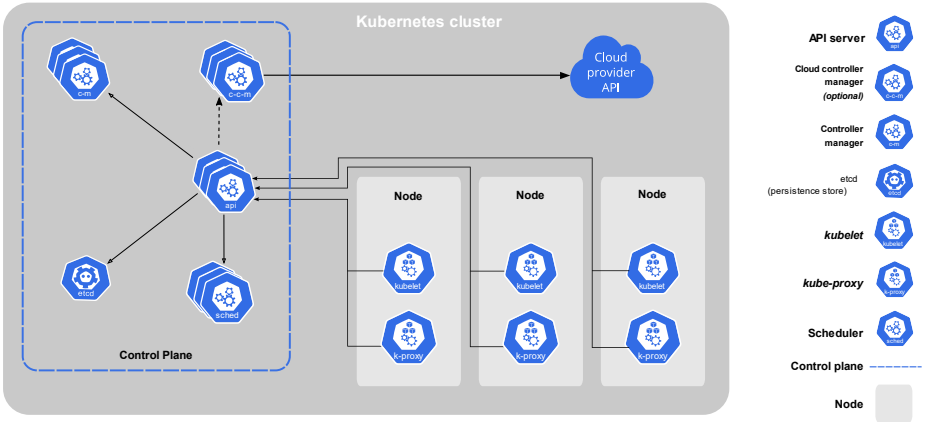
Els orquestradors són sistemes, habitualment clusteritzats, que s'encarreguen de gestionar el cicle de vida dels contenidors indicats especialment per aquelles plataformes dinàmiques amb càrregues de treball exigents. Les seves funcions són, entre d'altres activitats, les següents:

- Assegurar que estan en execució el número de contenidors configurats.
- Descarregar les imatges dels contenidors i gestionar el seu desplegament.
- Ampliar o reduir el número de contenidors en funció de la càrrega de les aplicacions.
- Balancejar la distribució dels contenidors entre els nodes en funció de l'ús dels recursos.
- Moure els contenidors, entre hosts que formen el clúster, en cas de fallida d'algun node.
- Exposar el ports dels contenidors, si estan configurats, per habilitar l'accés exterior.
- Monitoritzar l'estat dels contenidors i dels nodes.
- Gestió de volums per l'emmagatzematge de dades persistents dels contenidors.
- Automatització dels processos de desplegaments posant a disposició estratègies com *canary*[24] *release* o *blue-green*[25] *deployment*.

Les solucions d'orquestradors que podem trobar, a banda de Kubernetes, són principalment Apache Mesos i Docker Swarm.

2.3.4.3. Arquitectura

Un clúster de Kubernetes està format per múltiples servidors que poden assumir únicament 2 rols: *nodes* i *control plane*. Els *nodes*, únicament s'encarreguen d'executar aplicacions dins de contenidors coneguts amb el nom de *Pods* que veurem més endavant què significa i quina funció tenen. En canvi, el *control plane* es considera l'element que gestiona totes les operacions del clúster i dels propis *nodes*, així que, és l'element més crític i sobre el qual s'executen més serveis.



Il·lustració 9 - Arquitectura de Kubernetes.
Font: https://kubernetes.io/docs/concepts/overview/components/

[24] *Desplegament progressiu d'una aplicació per veure el comportament de la nova versió únicament amb volum reduït de tràfic..*
[25] *Disposar de dues versions diferents d'una aplicació en producció per redirigir el tràfic quan s'està segur que la versió nova és completament funcional. També es pot utilitzar pel procés invers: fer rollback a la versió anterior tornant a redirigir el tràfic.*

Considerant aquest últim punt, es recomana que el *control plane* en entorns productius estigui format per múltiples servidors amb els components distribuïts i/o replicats per garantir tolerància a fallides i alta disponibilitat de la plataforma.

2.3.4.3.1. Components control plane

Com s'ha explicat anteriorment, el *control plane* es l'element que incorpora més components de Kubernetes, els quals, passarem a enumerar tot seguit:

- **API Server:** aquest servei exposa la REST API de Kubernetes a l'exterior actuant com l'únic *endpoint* accessible del *control plane* que permet la creació, modificació o eliminació d'objectes per part dels clients si es tenen els permisos necessaris. Aquest component permet l'escalat horitzontal ja que l'emmagatzemat de dades està gestionat pel clúster *etcd*.

- **Etcd:** s'encarrega d'emmagatzemar tota la informació referent als objectes que existeixen en el clúster de Kubernetes així que es considera com el cor de la plataforma. *Etcd* és una aplicació de codi obert que consisteix en un magatzem distribuït i replicat de registres clau-valor que implementa l'algorisme de consens *Raft* per garantir la disponibilitat de les dades en cas de fallida d'alguna rèplica.

- **Scheduler:** el component *scheduler* és el responsable de trobar el node més adient per aquells *pods* que encara no hagin estan assignats a cap node. Alguns dels factors que té present per prendre la decisió són: requeriment específics de recursos, polítiques de restricció d'execució, ubicació de dades persistents o configuració específica d'afinitat.

- **Control manager:** responsable de vigilar que totes les definicions dels objectes compleixen amb l'estat desitjat, és a dir, es podria considerar com una eina de monitoratge però amb capacitat per executar operacions per a restaurar el servei.

- **Cloud control manager:** aquest component tan sols té utilitat en casos en que el clúster està desplegat en un proveïdor de cloud públic. La seva funció és interactuar amb la API del proveïdor per realitzar les mateixes funcions que el *control manager* però en aquest cas monitoritzant l'estat dels recursos desplegats en la plataforma cloud.

2.3.4.3.2. Components node

Els servidors amb rol *node* tenen una funcionalitat més limitada respecte el *control plane* i només tenen com a missió suportar la càrrega de treball assignada al clúster. Els components que podem trobar són:

- **Kubelet:** és l'agent que s'executa en cada node del clúster de Kubernetes per garantir que els *pods* que li han sigut assignats s'estan executant, amb capacitat de prendre accions per regenerar-los en cas que sigui necessari. A més a més, es comunica amb l'API Server per intercanviar-se informació sobre els *pods* que té en execució i quins més se li han assignat

- **Kube-proxy:** aquest component actua com a proxy per gestionar la xarxa dels nodes. Bàsicament, interactua amb els objectes tipus *service* que es on es defineixen les regles de xarxa per a permetre l'accés als *endpoints* dels *pods* que estan en execució, ja sigui des del propi clúster o des de l'exterior.

- **Container runtime:** és el motor que possibilita l'execució dels contenidors en els *nodes*. Actualment, estan suportats els següent motors: Docker, containerd i CRI-O.

2.3.4.4. Tipus de clústers

A l'hora de posar en marxa un clúster de Kubernetes tenim multitud d'opcions disponibles: instal·lació local, auto-gestionat o gestionat per un proveïdor. A continuació, enumerarem quines opcions podem escollir entre les categories anteriors:

- **Instal·lació local:** es tracta d'instal·lar un clúster de Kubernetes en una màquina local bàsicament per temes de desenvolupament. Les solucions que ofereixen aquest servei són: minikube o kind.

- **Instal·lació auto-gestionada:** es tracta d'instal·lar un clúster de Kubernetes en servidors *on-premise* o en el cloud però realitzant la instal·lació de cada rol de forma manual i assumint posteriorment el manteniment. Per dur a terme la instal·lació, existeixen eines que faciliten moltíssim el procés com kubespray, kubeadm i kops.

- **Gestionada per un proveïdor:** en aquesta categoria es troben inclosos la majoria de proveïdors de cloud públic com Amazon Web Services , Google Cloud i Microsoft Azure amb els serveis EKS (Elastic Kubernetes Service), GKE (Google Kubernetes Engine) i AKS (Azure Kubernetes Service) respectivament. El que ofereixen és un servei on el *control plane* està completament gestionat per ells on, l'equip d'operacions, únicament s'ha d'encarregar de desplegar els *nodes* necessaris (utilitzant una imatge del propi proveïdor)

Per a la realització d'aquest treball, optarem per l'última categoria ja que un dels objectius és dissenyar i implementar una solució purament *cloud-native*.

2.3.4.5. Operativa

Un cop tenim un clúster operatiu i coneixem els seus components el que ens interessa es poder administrar-lo i crear objectes. Com ja s'ha explicat anteriorment, l'únic element exposat que ens permet gestionar el clúster és l'API Server i, per a fer-ho, necessitarem una eina de línia de comandes específica anomenada ***kubectl.***

La configuració d'aquesta utilitat, on es poden definir connexions a més d'un clúster mitjançant la creació d'un context, es troba habitualment en la ruta `$HOME/.kube/config`. En aquest fitxer es troben les propietats de connexió com la URL del clúster, usuari, certificat, *namespace*, etc...

Val la pena dir, que tant la gestió com la creació d'objectes en el clúster es pot realitzar mitjançant la comanda *kubectl* passant-li els conjunt de paràmetres adients (forma imperativa)

o únicament executant la comanda passant-li com a paràmetre un fitxer de configuració en format YAML[26] on es trobin definides les accions a realitzar (forma declarativa). De fet, i com s'ha anat remarcant en repetides ocasions en aquest document, l'última opció és la recomanada ja que ens interessa disposar de tota la configuració en un repositori de control de versions.

No detallarem en aquesta secció la totalitat de les comandes que ofereix l'eina *kubectl* a causa de l'elevat número de paràmetres que té disponibles. En la fase d'implementació del projecte s'aniran explicant aquelles que, segons les necessitats, s'hagin d'utilitzar tot i que es poden consultar en aquest enllaç.

2.3.4.6. Objectes

Després d'investigar sobre els components que formen un clúster de Kubernetes i com administrar-los, passarem a veure com podem treure partit a les funcionalitats d'aquest producte mitjançant la creació de recursos. Els objectes són entitats persistents en el clúster que ofereixen serveis per a desplegar, mantenir i escalar aplicacions.
A continuació, s'enumeraran els objectes de Kubernetes més utilitzats, dividits en categories segons l'àrea a la que corresponen.

2.3.4.6.1. Computació

- **Pods:** un *Pod* és un o més contenidors que actua com una sola entitat, considerat com l'objecte més petit i bàsic que es pot implementar en Kubernetes. Aquesta característica d'atomicitat fa que en cas que un *Pod* tingui més d'un contenidor, aquests, estiguin ubicats en un mateix node compartint recursos de xarxa i d'emmagatzemament. Cal dir que el cicle de vida d'un *Pod* és curt i, en cas de caiguda del node o error d'execució, aquest no es tornarà a executar al no trobar-se persistit en el sistema.

- **ReplicaSets:** aquest objectes tenen com a objectiu garantir l'execució d'un número concret de rèpliques de *Pods* i, per tant, tenen la capacitat d'aixecar nous *Pods* si detecta que algun no està funcionant adequadament. A més a més, permet l'escalat horitzontal de *Pods* en escenaris de càrrega elevada en la capa d'aplicació.

- **Deployments:** són una implementació dels objectes *Pod* i ReplicaSets amb l'especialització particular de poder gestionar el desplegament de noves versions de les aplicacions. Entre les estratègies de desplegament que incorpora trobem:

  - **Recreate:** bàsicament elimina tots els pods existents del *ReplicaSet* i, posteriorment, en genera de nous utilitzant la nova versió de la imatge del contenidor. Aquest procés implica caiguda temporal del servei.

  - **RollingUpdate:** aquesta estratègia, tot i ser més lenta que l'anterior, és més robusta ja que el servei es troba actiu durant tot el procés. El funcionament consisteix en anar actualitzant progressivament la versió dels *Pods* en execució amb la nova versió

[26] *Acrònim de "YAML Ain't Markup Language" que correspon a un tipus de fitxer que té com objectiu principal serialitzar dades a partir d'un format de fàcil lectura basat en la indentació.*

de la imatge del contenidor i, un cop verificat que s'han actualitzat correctament, s'eliminen els de la versió antiga.

- **StatefulSets:** es tracta d'objectes amb les mateixes característiques que *ReplicaSets* però indicat per aplicacions que necessiten conservar l'estat pel seu funcionament (identificador, emmagatzematge, desplegament ordenat, etc...).

- **DaemonSets:** els objectes *Daemonset* garanteixen que tots els nodes del clúster de Kubernetes tindran sempre una còpia d'un pod en execució. Aquests objectes estan indicats per aplicacions d'agregació de logs o monitorització.

- **Jobs:** els *jobs* creen *pods* que executen tasques concretes de durada determinada i es destrueixen automàticament quan aquestes finalitzen correctament.

- **CronJob:** són objectes idèntics als *jobs* però amb la capacitat de ser programats per executar-se en una data i hora concreta. La programació es realitza utilitzant el format *cron* dels sistemes Unix.

2.3.4.6.2. Xarxa

- **Services:** aquest objecte és una capa d'abstracció que agrupa un o més pods per a exposar-los externament com un únic punt d'entrada (URL única). En funció del nivell d'accessibilitat que necessitem podem triar quatre tipus:

    - ***ClusterIP*:** exposa el servei en una IP interna del clúster.
    - ***NodePort***: exposa el servei en un port en tots els nodes del clúster.
    - ***LoadBalancer*:** exposa el servei externament utilitzant un balancejador de càrrega creat en la plataforma del proveïdor de cloud.
    - ***ExternalName*:** enllaça el servei amb un registre DNS extern.

- **Ingress:** és un objecte amb capacitats de balancejador de xarxa HTTP/HTTPS, permetent la terminació SSL en aquest capa d'accés. En combinació amb l'objecte *Service*, es podria considerar una alternativa més econòmica a l'opció *LoadBalancer*.

- **NetworkPolicy:** actua com un firewall dins del clúster al permetre establir regles de seguretat per a gestionar el tràfic intern, operant en les capes 3 i 4 del nivell OSI.

2.3.4.6.3. Configuració

- **ConfigMaps:** s'utilitza per emmagatzemar dades no sensibles en format clau/valor per ser utilitzades com paràmetres de configuració dels *Pods*. Aquest objecte, evita que s'injectin configuracions específiques en la fase de desenvolupament d'una aplicació, afavorint la immutabilitat i portabilitat d'aquesta.

- **Secrets:** a diferència de l'objecte anterior, aquest s'encarrega d'emmagatzemar dades sensibles com contrasenyes o claus SSH privades.

2.3.4.6.4. Emmagatzematge

- **Volume:** aquest objecte s'utilitza quan necessitem emmagatzemar les dades generades per un *Pod* en un sistema de fitxers fins que aquest existeixi.

- **PersistentVolume:** a diferència del cas anterior, aquest objecte s'utilitza quan necessitem emmagatzemar les dades generades per un *Pod* però que continuïn disponibles encara que aquest s'elimini. El volum generat és gestionat pel clúster de Kubernetes utilitzant algunes de les solucions d'emmagatzematge disponibles com NFS, iSCSI, glusterFS, Amazon EBS, Google GCE, etc..

2.3.4.6.5. Organització del clúster

- **Namespace:** es considera com una espècie de directori dins del clúster de Kubernetes per emmagatzemar objectes, amb la particularitat que permet establir una capa de separació i aïllament entre objectes de diferents *namespaces*. És molt útil per a crear entorns dedicats per evitar que les accions d'un equip puguin interferir en les d'una altre.

- **Label:** són etiquetes de tipus clau/valor que es poden afegir als objectes per a permetre identificar-los fàcilment a l'hora de fer consultes a la API.

- **Annotations:** són també anotacions de tipus clau/valor que són utilitzades per aplicacions terceres, per funcions experimentals de Kubernetes o per assignar meta informació als objectes però que no són utilitzades per processos internes del clústers.

2.4. Mètodes de desplegament

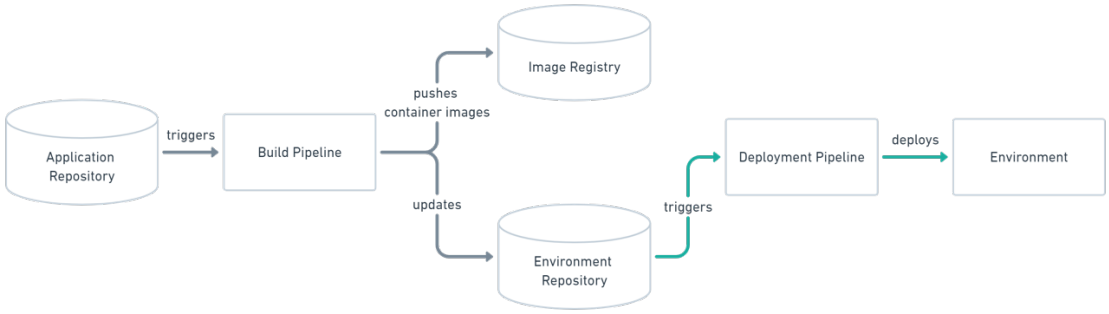
A l'hora d'implementar l'estratègia de desplegament utilitzant la filosofia GitOps podem escollir entre aquestes dues opcions: *Push-based* o *Pull-based*.

2.4.1. Push-based deployment

Aquest mètode, també conegut com "*agentless GitOps*", està basat en eines dedicades a CI/CD que són les encarregades de realitzar el desplegament sobre la plataforma destí, habitualment, Kubernetes. Amb aquest sistema tenim, d'una banda, el procés que s'encarrega de construir la imatge del contenidor i publicar-la al *registry* a partir de processos automatitzats d'integració continua i, de l'altra, el repositori de codi on es troba definida la infraestructura que al detectar qualsevol canvi, llança automàticament la *pipeline* de desplegament a l'entorn afectat.

Els inconvenients que presenta aquest tipus de desplegament és que l'eina de CI/CD ha de tenir credencials amb privilegis elevats suposant això un risc de seguretat. A més a més, com el repositori de codi ha de notificar dels esdeveniments que succeeixen a l'eina de CI/CD per automatitzar els desplegaments, serà indispensable exposar la API externament afegint un altre element de risc per la seguretat de l'entorn. Finalment, aquest mètode no disposa de

mecanismes de monitorització que garanteixin el compliment de l'estat desitjat ja que només té capacitat d' aplicar la configuració desitjada quan es registra un canvi en el repositori de codi.

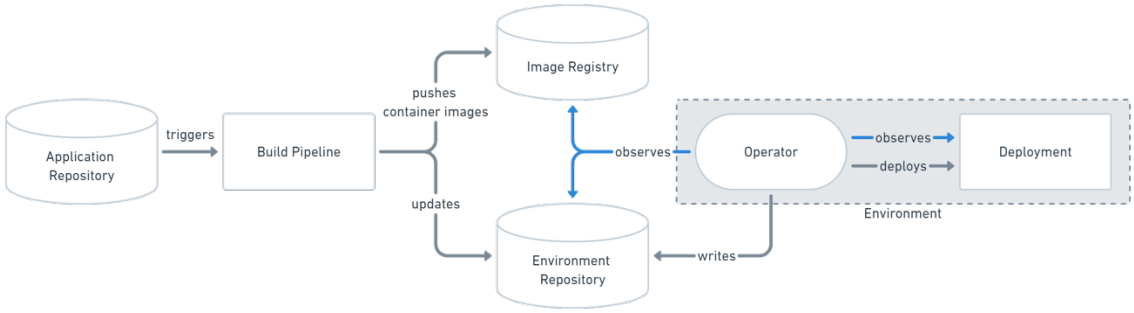


Il·lustració 10 - Diagrama de desplegament push-based
Font: https://www.gitops.tech/

2.4.2. Pull-based deployment

L'alternativa al mètode anterior té el nom de "*Pull-based deployment*" o "*Agent based GitOps*". El seu objectiu és el mateix, desplegar canvis d'infraestructura i d'aplicació automàticament quan es detecti un canvi en el repositori. La diferència la trobem en l'aparició d'un component nou anomenat operador que s'encarrega de monitoritzar el repositori constantment per detectar si l'estat actual de la plataforma i el desitjat coincideixin. En cas que s'identifiquin inconsistències, l'operador, intentarà modificar l'estat de l'entorn per a deixar-lo configurat tal com està especificat en el repositori.

Com es pot deduir, aquest tipus de desplegament soluciona les mancances de seguretat identificades en el cas anterior. D'una banda, no cal utilitzar credencials per accessos externs ja que l'eina encarregada de monitoritzar l'estat i realitzar els desplegaments resideix en el mateix clúster de Kubernetes amb els permisos necessaris per realitzar les operacions necessàries (mitjançant l'aplicació de polítiques RBAC[27] o comptes de servei) i, de l'altra, l'eina de desplegaments no s'ha d'exposar públicament perquè és aquesta la que consulta periòdicament el repositori de codi per detectar canvis.



Il·lustració 11 - Diagrama de desplegament pull-based
Font: https://www.gitops.tech/

[27] *Sigles de Role Based Access Control. Component de Kubernetes que gestiona usuàries/usuaris, grups i polítiques de permisos.*

2.5. Beneficis

Després d'haver investigat l'ecosistema que envolta la metodologia GitOps, aprofitarem aquest últim apartat per identificar els avantatges que la implantació d'aquest tipus de pràctiques ofereix a les organitzacions que es dediquen al sector del software:

- **Velocitat d'entrega de valor:** l'acceleració dels cicles de desenvolupament, permet desplegar funcionalitats noves pels clients de forma continuada i en un període de temps més curt, millorant l'estratègia *time-to-market*[28] corporativa.

- **Ambient col·laboratiu:** el fet que qualsevol canvi hagi de passar pel procés de revisió de codi, a través del mecanismes *Pull Request* o *Merge Request*, potencia la interacció entre els enginyers facilitant, d'una banda, la comunicació en la organització i, de l'altra, la transferència de coneixement passiva. A més a més, permet involucrar als equips de desenvolupament en tasques d'infraestructura i orientar als equips d'operacions tradicionals cap a pràctiques pròpies de desenvolupament de programari.

- **Consistència entre entorns:** la definició declarativa de la infraestructura i les eines de desplegament, que asseguren que la plataforma es troba en l'estat desitjat, redueixen les divergències entre els entorns de desenvolupament/test i l'entorn productiu.

- **Millora de la seguretat:** com la gestió de la plataforma es realitza mitjançant eines d'automatització i a través d'operadors de Kubernetes, és a dir, accessos interns, reduïm la necessitat d'utilitzar credencials personals per a realitzar operacions sobre la infraestructura.

- **Traçabilitat dels canvis:** gràcies al sistema de control de versions, podem consultar el registre històric sobre els canvis realitzats en la plataforma i l'autoria d'aquests.

- **Reducció de riscos:** l'eliminació d'intervencions manuals per operar amb la infraestructura suposa una reducció dels errors humans. A més a més, a mesura que anem repetint el procés automàtic, millorem la confiança i fiabilitat del mateix.

- **Reducció de costos:** un sistema on tots els processos estan tan automatitzats ens ajuda reduir el temps a invertir en tasques manuals de manteniment. A més a més, la implantació d'*Infrastructure as Code* com a mètode únic per aprovisionar infraestructura evita que es creïn recursos en les plataformes cloud de manera manual. En moltes ocasions, els recursos creats manualment queden oblidats suposant un cost innecessari per les empreses.

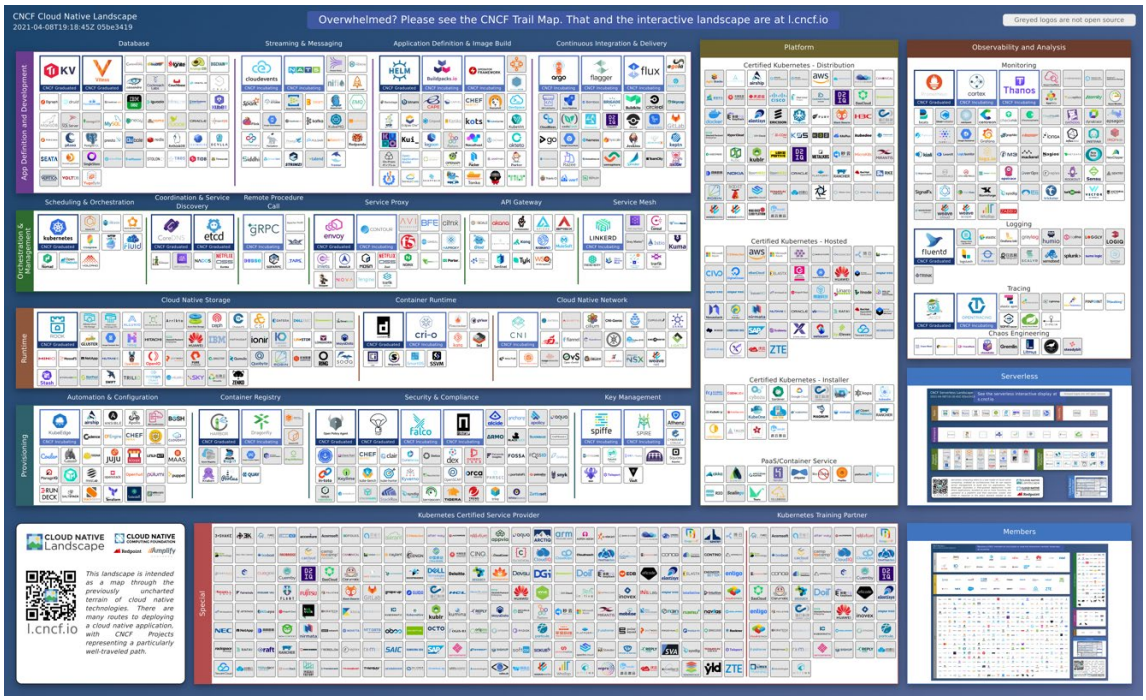
- **Pla de recuperació ràpid:** l'automatització de processos, la definició de la infraestructura com a codi i l'emmagatzematge de les aplicacions en format immutable permet regenerar la plataforma sencera en un període curt de temps. D'aquesta manera, amb (còpies de seguretat), disposaríem d'un pla de contingències complet, provat i preparat per ser aplicat en cas de desastre.

[28] *En les empreses, període de temps que transcorre des de que apareix una idea i aquesta està disponible pels clients.*

3. Selecció de tecnologies

Després d'haver estudiat profundament el concepte GitOps durant els apartats anteriors, a partir d'ara, el treball de final de grau passarà a tenir un enfocament més pràctic en el qual s'aplicaran els coneixements adquirits per dissenyar i implementar una solució completa de desplegament d'aplicacions i gestió d'infraestructura.

Primerament, identificarem els productes principals que utilitzarem pel projecte, tot i que, puntualment, es podria utilitzar alguna eina addicional per a realitzar tasques específiques. Per a realitzar la selecció, a banda de revisar publicacions en Internet sobre comparatives entre diferents solucions, ens recolzarem també en la classificació d'aplicacions que ofereix la Cloud Native Computing Foundation (CNCF[29]). Aquesta organització, que forma part de la Linux Foundation, suporta multitud de projectes de codi obert per a impulsar l'adopció de tecnologies *cloud-native* preferiblement d'aquelles que no estableixin dependències amb un únic proveïdor.



Il·lustració 12 – Tecnologies CNCF
Font: https://landscape.cncf.io/images/landscape.png

Els criteris que s'han tingut en compte per a realitzar la selecció són:

- La maduresa i la popularitat del producte.
- Preferència per solucions de codi obert.
- Preferència per solucions agnòstiques.

3.1. GitHub

GitHub és la plataforma més estesa per emmagatzemar projectes de desenvolupament sota el sistema de control de versions Git, propietat de la companyia Microsoft des de l'any 2018.

[29] https://www.cncf.io/

Aquest portal, facilita la gestió de les operacions que es poden realitzar amb git i disposa d'una interfície web molt intuïtiva que promou la col·laboració entre la comunitat de desenvolupament. La URL per accedir al servei és https://github.com/.

Il·lustració 13 - Logo de GitHub
Font: https://github-media-downloads.s3.amazonaws.com/GitHub-Mark.zip

El plans que ofereix GitHub per utilitzar la plataforma són:

- **Free:** és la capa gratuïta d'ús de la plataforma SaaS que permet utilitzar les funcions bàsiques, ideal per a un ús particular.

- **Team:** aquest pla, també de tipus SaaS, està indicat per a organitzacions petites dedicades al desenvolupament de software. Afegeix funcionalitats més avançades que la capa gratuïta, especialment, aquelles relacionades amb gestió de grups, seguretat i permisos

- **Enterprise:** aquest model incorpora totes les funcionalitats que té disponibles la plataforma i, per tant, el que té el cost més elevat. Està indicat per a organitzacions grans i, a banda de la solució SaaS, també permet la instal·lació de la plataforma en servidors privats.

- **Student:** aquest pla, ofert únicament als estudiants, ofereix les mateixes funcionalitats que el model de subscripció "*Team*". Serà l'opció escollida per la realització d'aquesta treball.

Com s'ha explicat en apartats anteriors, el mètode per a sol·licitar canvis en un repositori de GitHub s'anomena *Pull Request*. Aquest terme, s'utilitzarà amb freqüència ja que serà una acció molt freqüent en la operativa GitOps.

Una alternativa a GitHub que també s'ha avaluat ha sigut GitLab però s'ha considerat que ofereix prestacions inferiors especialment en la usabilitat de la interfície web i en la popularitat en la comunitat.

3.2. ArgoCD

ArgoCD és una eina de desplegament continu indicada especialment per l'aplicació de metodologies GitOps. La descripció que apareix en la pàgina web del producte, disponible en la URL https://argoproj.github.io/argo-cd/, explica que segueix exclusivament els principis GitOps que s'han detallat anteriorment en aquest document: ús exclusiu de definicions declaratives, desplegament automatitzat, traçabilitat dels canvis i fàcil d'entendre.

Il·lustració 14 - ArgoCD logo
Font: https://cncf-branding.netlify.app/projects/argo/

Es tracta d'un projecte de codi obert, escrit majoritàriament amb el llenguatge Golang, i que la fundació CNCF considera ArgoCD com una de les eines de *continuous delivery* més madures i evolucionades, categoritzada amb el nivell "*Incubating*".

A més a més, utilitza el mètode de desplegament *Pull-based* que es la opció que es considera més adient per temes de seguretat, justificat en l'apartat "Mètodes de desplegament". La instal·lació d'ArgoCD es realitza sobre Kubernetes actuant com a operador intern que li permet realitzar operacions de desplegament.

A banda d'ArgoCD també s'han avaluat les opcions FluxCD i Jenkins X. La primera s'ha descartat perquè només suporta un únic repositori i la segona perquè el projecte es troba en un nivell inferior de maduresa segons el catàleg de la CNCF.

3.3. Google Cloud

La plataforma cloud serà un component fonamental ja que és on residirà tota la infraestructura que necessitarem per a implementar la solució tècnica. Bàsicament, els recursos que necessitarem seran: computació, emmagatzematge, gestió de zona DNS i Kubernetes gestionat. La plataforma escollida és Google Cloud perquè és la que té més experiència en el servei de clúster de Kubernetes gestionat, anomenat GKE (Google Kubernetes Engine), comparat amb Amazon Web Services i Microsoft Azure que són els competidors més directes. A més a més, es valora el fet que el número de serveis que ofereix Google Cloud és més reduït però més que suficient pels requeriments del projecte.

Il·lustració 15 - Google Cloud logo
Font: https://cloud.google.com/docs

Google Cloud disposa de 24 regions distribuïdes geogràficament amb 73 zones de disponibilitat. A continuació, explicarem breument aquells serveis que ofereix i utilitzarem pel desenvolupament d'aquesta pràctica:

- **Virtual Private Cloud:** és l'encarregat de gestionar la topologia de xarxa per garantir la comunicacions entre els recursos del cloud i també exposar serveis públicament.

- **Compute Engine:** és el servei per a crear i executar màquines virtuals en el cloud de Google. Aquests recursos són elements bàsics en aquestes plataformes i permet desplegar servidors amb sistemes operatius compatibles (Linux i Windows). El preu, va en funció de la capacitat escollida (memòria RAM i vCPUs).

- **IAM:** element crític en la plataforma ja que és el responsable d'autoritzar o denegar els permisos d'accés al recursos.

- **Cloud Load Balancing:** és un dispositiu de xarxa que permet exposar un punt d'accés únic, ja sigui per accessos interns o exposició pública (Internet), darrera del qual es troben recursos (habitualment màquines virtuals) que reben el tràfic que aquest distribueix.

- **Cloud DNS:** aquest servei permet hostatjar i gestionar una zona DNS.

- **Cloud Storage Engine:** és el servei especialitzat per l'emmagatzematge i recuperació de dades, indicat per oferir contingut estàtic dels llocs web, còpies de seguretat o arxius.

- **Cloud SQL:** és un servei de base de dades relacionals autogestionades per Google Cloud amb suport pels següents motors: MySQL, PostgreSQL o Microsoft SQL Server.

- **Google Kubernetes Engine (GKE):** aquest servei proporciona un clúster de Kubernetes, basat en recursos de tipus Compute Engine, administrat completament per Google.

- **Container Registry:** és un servei per emmagatzemar imatges de contenidors de forma privada.

- **Google Cloud Build**[30]**:** és una eina *serverless* que permet executar càrregues de treball relacionades amb processos d'integració i desplegament continu.

3.4. Terraform

Terraform és una aplicació de codi obert, desenvolupada per la companyia Hashicorp, dissenyada per a codificar infraestructura de forma declarativa, és a dir, és l'eina que s'utilitzarà per implementar el concepte "*Infrastructure as Code*". El llenguatge utilitzat per a crear els arxius de configuració per a gestionar la infraestructura s'anomena HCL (Hashicorp Configuration Language).

El repositori del projecte està emmagatzemat en un repositori de la plataforma GitHub, en la URL https://github.com/hashicorp/terraform, i com es pot comprovar està escrit en el llenguatge Golang.

Il·lustració 16 - Terraform logo
Font: https://www.terraform.io/

La principal característica és la seva extensibilitat mitjançant mòduls (alguns d'ells públics en el registry de terraform) i la capacitat agnòstica, és a dir, és compatible amb multitud de proveïdors

[30] *Serverless és una tecnologia de computació que permet executar tasques, en diversos llenguatges de programació, sense necessitat de desplegar servidors dedicats.*

com es pot verificar a partir d'aquest enllaç: https://registry.terraform.io/browse/providers. Aquest fet, fa especialment potent aquesta utilitat ja que permetria canviar de proveïdor de cloud o tenir una arquitectura multi-cloud sense haver d'aprendre a utilitzar des de zero un nou producte per codificar la infraestructura. Per a realitzar operacions en els proveïdors, terraform interactua amb les APIs que aquests exposen sempre i quan en la sessió d'execució es trobin credencials vàlides i amb privilegis per realitzar la petició.

Per a utilitzar aquesta eina necessitarem instal·lar un client, anomenat *terraform*, en la màquina que s'encarregui d'aprovisionar o modificar la infraestructura. Aquest arxiu únic, de format binari, està disponible per sistemes operatius Windows, Linux o OSX i es pot instal·lar descarregant-lo directament des d'aquesta pàgina https://www.terraform.io/downloads.html o bé utilitzant qualsevol dels gestor de paquets d'aplicacions (brew, yum, dnf, apt-get, etc...). Un cop el tinguem instal·lat, el podrem executar utilitzant la línia de comandes.

Després de la instal·lació, podríem crear el primer fitxer de configuració que permetria desplegar infraestructura en el proveïdor que haguem definit en l'arxiu si es disposa de credencials vàlides. En la fase d'implementació, es tractarà amb més detall les operacions que es poden realitzar amb terraform.

4. Preparació de l'entorn de treball

Després d'haver exposat la part conceptual de la metodologia GitOps, a partir d'ara, passarem a posar en pràctica el coneixement adquirit dissenyant una solució tècnica que segueixi la filosofia que s'ha descrit en els apartats anteriors. Separarem els propers apartats en tres seccions que es desenvoluparan en l'ordre següent:

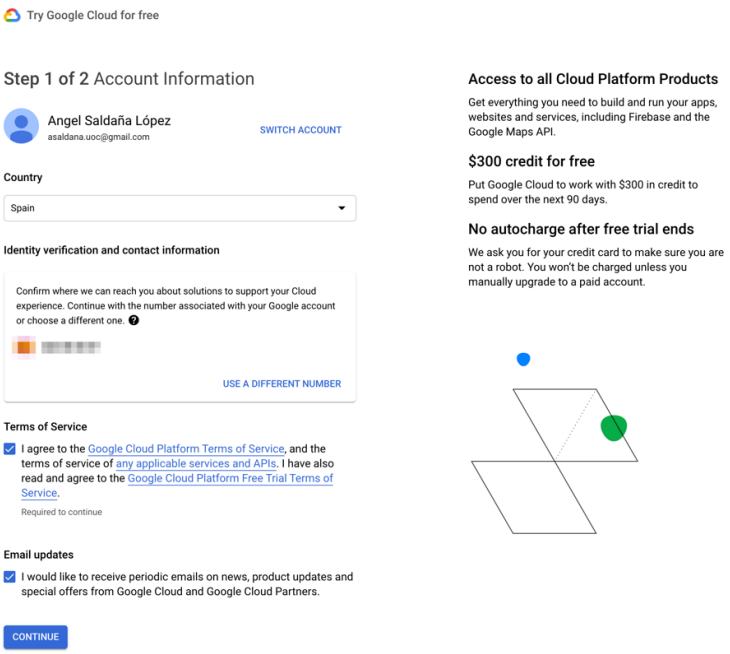
- Preparació de l'entorn local de treball i alta als serveis (Google Cloud i GitHub).
- Disseny i implementació de la infraestructura base que es necessita per a desplegar una aplicació en un clúster de Kubernetes .
- Creació d'una aplicació de prova que ens permeti validar el funcionament global de tot l'ecosistema GitOps.

4.1. Alta a Google Cloud Platform

Primerament, per tenir la capacitat de desplegar infraestructura haurem de crear un compte en el proveïdor de cloud públic escollit que, per aquest projecte, serà Google Cloud Platform (a partir d'ara, GCP). Aprofitarem la promoció disponible, a data de realització d'aquest treball, que consisteix en un crèdit de $300 per ser consumit en 90 dies, a banda de la capa gratuïta. Per a realitzar el registre, anirem a la següent URL:

- https://console.cloud.google.com/freetrial

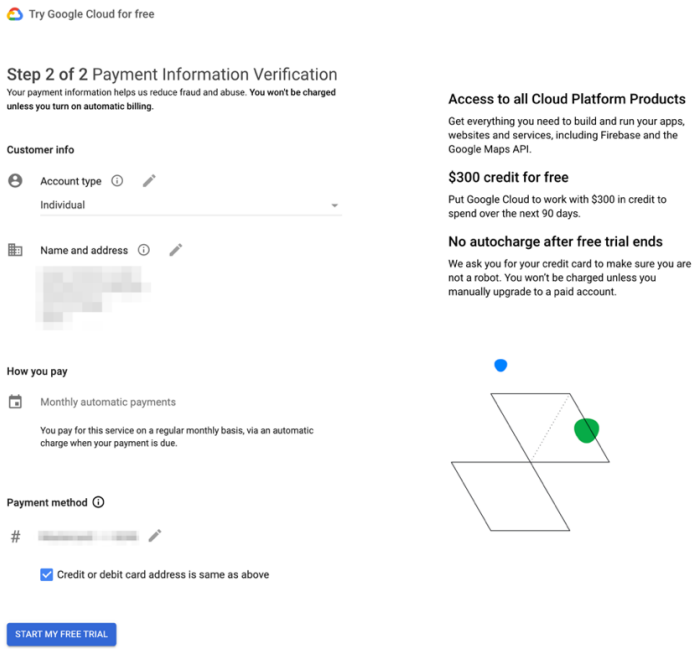
Com es pot veure en la captura següent, utilitzarem un compte de gmail per a donar-nos d'alta a GCP, acceptant les condicions d'ús i clicant en "*Continue*":



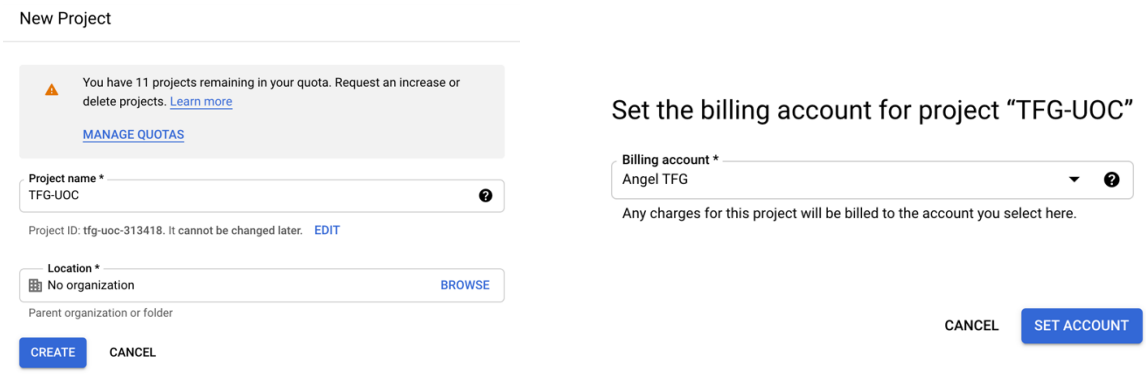
Tot seguit, haurem d'introduir les dades següents:

Camp a complimentar	Valors possibles
Tipus de compte	Individual o Business
Nom	Nom complet de la persona administradora
Adreça	Adreça de la persona administradora
Mètode de pagament	Dades de la targeta bancària

Complimentem el formulari amb les dades corresponents, seleccionant compte de tipus individual, i finalitzem el procés d'alta:



Un cop ja tenim el compte disponible crearem un nou projecte, que anomenarem TFG-UOC, sobre el qual desplegarem la infraestructura que necessitem per aquest cas pràctic i l'enllaçarem al compte de facturació:



Després de realitzar els passos anterior, podem donar per finalitzada la creació del compte de Google Cloud així que ja el tindrem disponible per desplegar infraestructura quan arribi el moment.

4.1.1. Client de Google Cloud Platform: *gcloud*

Tot i que es poden gestionar els recursos que es creïn a GCP fent servir un navegador i accedint consola a través de la URL https://console.cloud.google.com, instal·larem també l'eina de línia de comandes (*gcloud*) que proporciona Google ja que, en algunes ocasions, serà més ràpid consultar o crear recursos de forma programàtica en comptes de fer-ho a través de la consola web. Seguirem el procés descrit en aquest pàgina web.

1. Verificarem que tenim Python (versions 3.5 fins a 3.8 o 2.7.9 o superior) instal·lat:

```
~ python -V
Python 3.8.7
```

2. Descarregarem l'última versió del paquet *google-cloud-sdk* que correspongui amb el sistema operatiu i model de processador utilitzant *wget* i, posteriorment, descomprimim l'arxiu:

```
$ wget -q https://dl.google.com/dl/cloudsdk/channels/rapid/downloads/google-cloud-sdk-341.0.0-darwin-x86_64.tar.gz
$ tar xzf google-cloud-sdk-341.0.0-darwin-x86_64.tar.gz
$ cd google-cloud-sdk
```

3. Executem l'script d'instal·lació:

```
$ ./install.sh
```

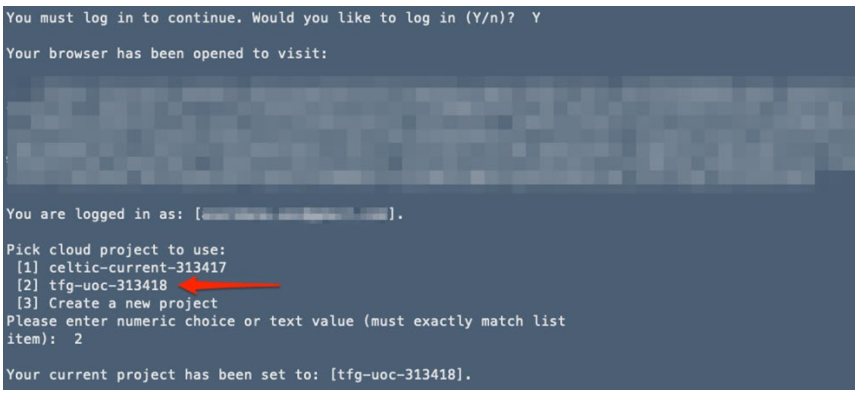
4. L'script anterior afegeix a l'arxiu d'inicialització de l'intèrpret de comandes, en el cas actual *zsh*, la ruta als binaris del SDK (Software Development Kit) de GCP. Finalment, comprovem que podem executar la utilitat *gcloud*:

```
~ gcloud version
Google Cloud SDK 341.0.0
bq 2.0.68
core 2021.05.14
gsutil 4.62
```

Un cop tenim l'executable *gcloud* instal·lat localment, passarem a inicialitzar l'eina per a que pugui accedir al projecte de GCP que hem creat anteriorment. Per a poder realitzar la configuració, executarem la comanda següent i anirem introduint les dades que pregunten:

```
$ gcloud init
```

Ens autenticarem a la plataforma accedint a l'enllaç que ens apareix en la consola i, per acabar, seleccionarem el projecte que volem establir per defecte (*tfg-uoc-313418).*

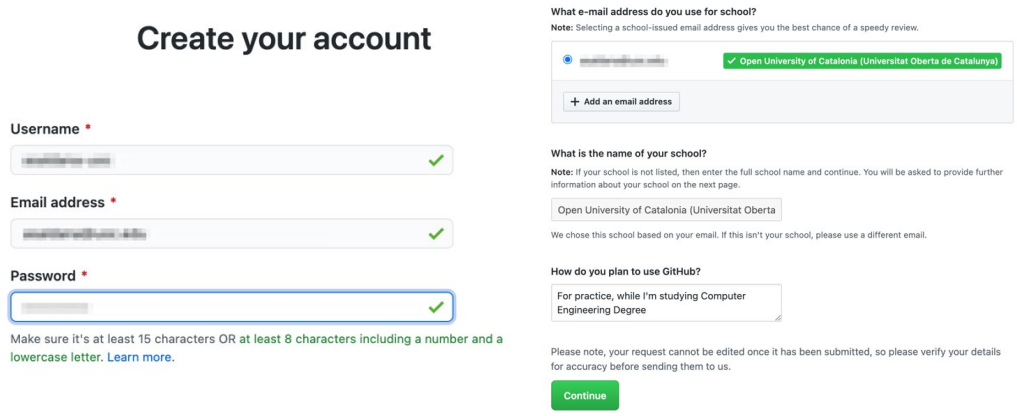


4.2. Creació de compte a GitHub

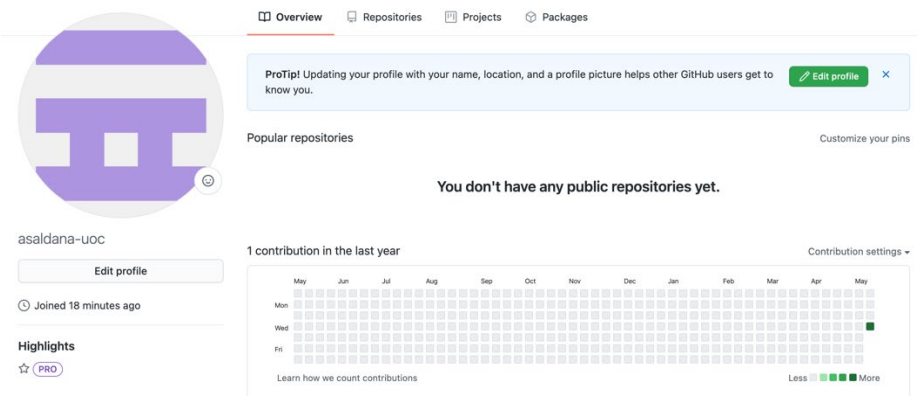
En aquest punt, crearem un compte a la plataforma GitHub que ens permetrà crear repositoris de control de versions basats en Git, essencials per aplicar els conceptes de la metodologia GitOps. S'ha investigat el producte i s'ha trobat que hi ha disponible una versió per estudiants que dona accés total a la plataforma de forma gratuïta, a banda d'altres avantatges en serveis oferts per socis col·laboradors de GitHub. La URL on es pot veure en detall totes les promocions és la següent:

- https://education.github.com/pack

Començarem el procés d'alta introduint el nom de la usuària o usuari, correu electrònic de la universitat i password d'accés. Posteriorment, haurem de confirmar que som propietaris del compte de correu i especificar el nom de la universitat (Open University of Catalonia):



Finalment, comprovem que ja tenim el perfil preparat per a crear repositoris i treballar amb ells:



4.3. Creació bucket Google Cloud Storage per Terraform State

La gestió de la infraestructura, utilitzant terraform, es fonamenta principalment en dos conceptes: l'estat actual de la infraestructura i la configuració existent del codi. Quan s'aplica terraform, primerament consulta l'estat i el compara amb el que hi ha definit als arxius de configuració; en funció de les diferències que trobi, realitzarà unes operacions o unes altres per tal de sincronitzar totes dues parts. Per defecte, l'arxiu d'estat s'emmagatzema localment amb el nom de *terraform.tfstate* però aquesta solució presenta molts inconvenients com, per exemple:

- Impedeix el treball en equip ja que l'estat es troba distribuït i no sincronitzat.
- Risc de pèrdua o eliminació de l'arxiu d'estat.
- Inconsistència en els recursos existents.

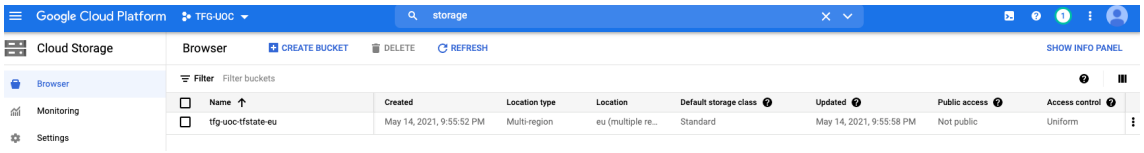
Per aquesta raó, és indispensable tenir una ubicació compartida i persistent on aquest estat quedi emmagatzemat. Un dels *backends* suportats per terraform és el servei d'emmagatzematge de GCP, conegut com a Google Cloud Storage (GCS), i la forma d'ús es pot consultar en aquest enllaç.

Per a crear un bucket en GCS utilitzarem la comanda ***gsutil***, que s'ha instal·lat conjuntament amb el client ***gcloud***, i s'especificarà la regió destí, el tipus d'emmagatzematge, el nom i la política de seguretat. A més a més, després de crear-lo, activarem el versionat de fitxers per tenir traçabilitat dels canvis:

```
$ gsutil mb -p tfg-uoc-313418 -b on -c standard -l eu gs://tfg-uoc-tfstate-eu
$ gsutil versioning set on gs://tfg-uoc-tfstate-eu
```

```
~ gsutil mb -p tfg-uoc-313418 -b on -c standard -l eu gs://tfg-uoc-tfstate-eu
Creating gs://tfg-uoc-tfstate-eu/...
~ gsutil versioning set on gs://tfg-uoc-tfstate-eu
Enabling versioning for gs://tfg-uoc-tfstate-eu/...
```

La sortida anterior ens mostra que el bucket amb nom *tfg-uoc-tfstate-eu* s'ha creat correctament amb versionat inclòs i amb política d'accés uniforme (paràmetre *-b on*) per tots els objectes, gestionat per IAM. tot i que anirem a la consola web per a verificar-ho:



4.4. Instal·lació d'eines a l'equip local

El següent que haurem de realitzar serà instal·lar localment, en el dispositiu de treball que utilitzem, les eines que ens permetran implementar, desplegar i administrar la plataforma. Tot i que es detallarà el procés d'instal·lació sobre sistema operatiu macOS, cal dir que la totalitat de les eines són compatibles també amb Windows i Linux.

4.4.1. Client de Git: *git*

El client de git també és una eina multi-plataforma que podem trobar les instruccions per instal·lar-ho en la secció de descàrregues de la pàgina web oficial:

- https://git-scm.com/downloads

Per a macOS, utilitzarem el gestor de paquets brew per dur a terme la instal·lació executant la instrucció següent:

```
$ brew install git
```

Un cop finalitzi el procés anterior, comprovarem que tenim git present en el sistema mostrant la versió instal·lada:

```
~ git version
git version 2.31.1
```

Per a que el client de git es pugui autentificar amb la plataforma GitHub, per a realitzar operacions de pujada/descarrega de codi, podem utilitzar dos protocols: https i ssh. Escollirem l'opció ssh i, per tant, haurem de generar un parell de claus (pública/privada) que s'utilitzaran per a configurar l'autentificació.

1. Creem el parell de claus (pública/privada) amb la utilitat ssh-keygen:

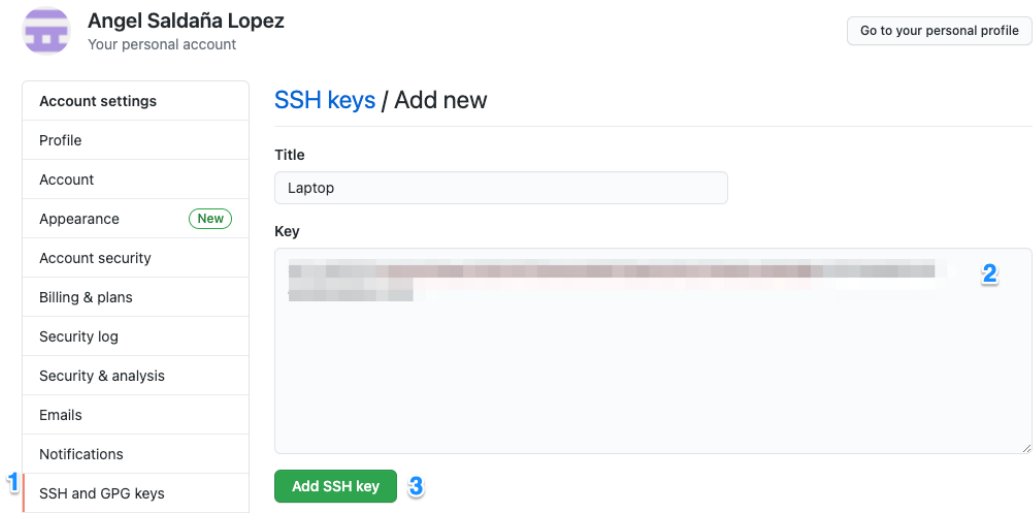
```
$ ssh-keygen -t ed25519 -C "XXXXXX@XXX.XXX" -f ~/.ssh/id_rsa_uoc
```

   Els modificadors passats en la instrucció següents ens donen la possibilitat de modificar els paràmetres per defecte de *ssh-keygen*.
   - **-t:** permet especificar el tipus de clau a crear: RSA, DSA, ECDSA o ECD25519. Escollim l´última ja que, de tots els disponibles, es tracta de l'algorisme de xifrat més segur.
   - **-C:** afegim com a comentari l'adreça de correu associada al compte de GitHub.
   - **-f:** especifiquem la ubicació on es desaran les clau pública i privada.

La sortida que obtenim es pot veure en la captura de pantalla següent:

```
~ ssh-keygen -t ed25519 -C "[redacted]" -f ~/.ssh/id_rsa_uoc
Generating public/private ed25519 key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/angel/.ssh/id_rsa_uoc.
Your public key has been saved in /Users/angel/.ssh/id_rsa_uoc.pub.
The key fingerprint is:
[redacted]
The key's randomart image is:
+--[ED25519 256]--+
[redacted]
+----[SHA256]-----+
```

Com que ja disposem del parell de claus, ara haurem de pujar la clau pública (*id_rsa_uoc.pub*) al perfil de GitHub. Per a fer-ho, anirem a la configuració del compte, seleccionarem "*SSH and GPG keys*" i afegirem el contingut de la clau pública en el quadre de text:



Tot seguit, afegirem a l'arxiu de configuració ***.gitconfig*** les dades personals per identificar l'autoria de cada commit realitzat:

```
# This is Git's per-user configuration file.
[user]
# Please adapt and uncomment the following lines:
       name = Angel Saldaña Lopez
       email = COMPTE_DE_CORREU_REGISTRAT_A_GITHUB
```

I, per acabar, afegirem una entrada a l'arxiu de configuració del client ssh, ubicat a ***~/.ssh/config***, per especificar la ruta a la clau privada que utilitzarem per autentificar-nos per ssh amb el host github.com:

```
Host github.com
        User git
        Hostname github.com
        PreferredAuthentications publickey
        IdentityFile ~/.ssh/id_rsa_uoc
```

4.4.2. Client de Terraform: *tfenv/terraform*

L'aplicació terraform, com en els dos casos anteriors, també està disponible per múltiples sistemes operatius a través d'aquest l'enllaç.

El tema de la versió de terraform és un factor a tenir molt present ja que, cada vegada que es realitza un canvi en la infraestructura, s'actualitza el fitxer d'estat incloent les modificacions realitzades i la versió del binari amb què s'ha executat. L'arxiu d'estat, que habitualment està emmagatzemat en ubicacions remotes i compartides (S3, Google Cloud Storage, Consul, etc..) per permetre el treball col·laboratiu, serveix per identificar les diferències existents entre la infraestructura desplegada i la declaració del codi. En cas que l'estat es modifiqui amb una versió concreta de terraform, les properes execucions mai podran utilitzar una versió inferior. Per aquesta raó, utilitzarem una utilitat de codi obert anomenada ***tfenv*** que ens facilitarà enormement la gestió de versions. Seguirem els passos d'instal·lació que trobem al *README* d'aquest repositori:

1. Utilitzarem el gestor de paquets brew per instal·lar la utilitat:

```
$ brew install tfenv
```

2. Instal·larem l'ultima versió de terraform que serà la que utilitzarem per aquest projecte:

```
$ tfenv install latest
```

```
Installing Terraform v0.15.3
Downloading release tarball from https://releases.hashicorp.com/terraform/0.15.3/terraform_0.15.3_darwi
n_amd64.zip
######################################################################################### 100,0%
Downloading SHA hash file from https://releases.hashicorp.com/terraform/0.15.3/terraform_0.15.3_SHA256S
UMS
No keybase install found, skipping OpenPGP signature verification
Archive:  tfenv_download.jHRjZ4/terraform_0.15.3_darwin_amd64.zip
  inflating: /usr/local/Cellar/tfenv/2.2.2/versions/0.15.3/terraform
Installation of terraform v0.15.3 successful. To make this your default version, run 'tfenv use 0.15.3'
```

3. Seleccionarem la versió 0.15.3 per defecte:

```
$ tfenv use 0.15.3
```

```
Switching default version to v0.15.3
Switching completed
```

4. I, per posar fi a la configuració de terraform, executarem el binari per consultar la versió:

```
$ terraform version
```

```
~ terraform version                                                    ✔ 3s
Terraform v0.15.3
on darwin_amd64
```

4.4.2.1. Credencials de GCP per Terraform

Tenir el client és un pas necessari per desplegar infraestructura però falta un punt molt important: la autentificació amb el proveïdor que utilitzem, en el cas d'aquest projecte GCP. Per aquesta raó, crearem un compte de servei a GCP aplicant l'estratègia d'assignar únicament els

permisos indispensables per a realitzar les operacions que necessita terraform, coneguda com POLP[31] (*Principle of Least Privilege*). Els rols en GCP es divideixen en tres tipus:

- **Basic roles:** són rols existents preestablerts en la plataforma que simplifiquen la gestió dels permisos sobre els recursos al tenir un abast global, és a dir, apliquen a tots els serveis. Els rols d'aquesta categoria que es poden assignar són *Owner* (permisos d'administració total), *Editor* () i *Viewer* (accés en mode lectura a tots els recursos).

- **Predefined roles:** són rols més avançats que tenen una granularitat superior als anteriors al poder establir permisos sobre serveis específics. Per facilitar l'administració, la plataforma GCP gestiona aquestes polítiques d'accés, les quals, podem seleccionar en funció de les necessitats particulars d'ús, documentades en aquesta pàgina.

- **Custom roles:** és idèntic al rol anterior però en comptes d'assignar polítiques d'accés preestablertes per GCP, aquestes hauran de ser creades pel client.

Pel cas d'ús d'aquesta pràctica, crearem un rol de tipus *predefined* sobre el qual afegirem les polítiques d'accés i administració dels serveis següents:

- Google Cloud Compute (`roles/compute.admin`)
- Google Kubernetes Engine (`roles/container.admin`)
- Google Cloud Storage (`roles/storage.admin`)
- Google Cloud Build (`roles/cloudbuild.builds.builder`)
- Google Cloud DNS (`roles/dns.admin`)

1. Creem un compte de servei amb el nom `tf-user`:

```
$ gcloud iam service-accounts create tf-user \
    --display-name="tf-user" \
    --description="Compte de servei utilitzat per terraform al projecte TFG-UOC"
```

```
gcloud iam service-accounts create tf-user \
    --display-name="tf-user" \
    --description="Compte de servei utilitzat per terraform al projecte TFG-UOC"

Created service account [tf-user].
```

2. Generem un arxiu, en format JSON[32], que emmagatzemarà localment les credencials del compte de servei que utilitzarà terraform:

```
$ gcloud iam service-accounts keys create ~/.config/gcloud/tf-user.json \
    --iam-account tf-user@tfg-uoc-313418.iam.gserviceaccount.com
```

```
gcloud iam service-accounts keys create ~/.config/gcloud/tf-user.json \
    --iam-account tf-user@tfg-uoc-313418.iam.gserviceaccount.com

created key [                                        ] of type [json] as [/Users/angel/.config/gclou
d/tf-user.json] for [tf-user@tfg-uoc-313418.iam.gserviceaccount.com]
```

[31] https://www.cyberark.com/what-is/least-privilege/
[32] *Acrònim de JavaScript Object Notation que correspon a un format senzill de fitxer per l'intercanvi de dades entre RESTful APIs*

3. Assignem els rols gestionats per GCP al compte de servei *tf-user*:

```
$ gcloud projects add-iam-policy-binding tfg-uoc-313418 \
      --member='serviceAccount:tf-user@tfg-uoc-313418.iam.gserviceaccount.com' \
      --role='roles/compute.admin'
$ gcloud projects add-iam-policy-binding tfg-uoc-313418 \
      --member='serviceAccount:tf-user@tfg-uoc-313418.iam.gserviceaccount.com' \
      --role='roles/container.admin'
$ gcloud projects add-iam-policy-binding tfg-uoc-313418 \
      --member='serviceAccount:tf-user@tfg-uoc-313418.iam.gserviceaccount.com' \
      --role='roles/storage.admin'
$ gcloud projects add-iam-policy-binding tfg-uoc-313418 \
      --member='serviceAccount:tf-user@tfg-uoc-313418.iam.gserviceaccount.com' \
      --role='roles/cloudbuild.builds.builder'
$ gcloud projects add-iam-policy-binding tfg-uoc-313418 \
      --member='serviceAccount:tf-user@tfg-uoc-313418.iam.gserviceaccount.com' \
      --role='roles/dns.admin'
```

4. També haurem d'activar les APIs dels serveis que utilitzarem:

```
$ gcloud services list --available # Llistem les APIs disponibles en GCP
$ gcloud services enable container.googleapis.com
$ gcloud services enable compute.googleapis.com
$ gcloud services enable storage.googleapis.com
$ gcloud services enable cloudbuild.googleapis.com
$ gcloud services enable dns.googleapis.com
```

5. Finalment, haurem de configurar la variable d'entorn *GOOGLE_APPLICATION_CREDENTIALS* amb la ruta al fitxer JSON on es troben les credencials del compte de servei. Per evitar realitzar-ho cada vegada que obrim un terminal, afegirem la configuració d'aquesta variable d'entorn en l'script d'inici de la *shell*:

```
$ echo "export GOOGLE_APPLICATION_CREDENTIALS=~/.config/gcloud/tf-user.json" >>
~/.zshrc
```

4.4.3. Client de Kubernetes: *kubectl*

Per interactuar amb la API de Kubernetes també necessitarem un client, anomenat ***kubectl***. La documentació per a dur a terme la instal·lació està disponible en aquest enllaç:

1. Descarregarem la versió 1.21 de *kubectl*:

```
$ curl -LO "https://dl.k8s.io/release/v1.21.0/bin/darwin/amd64/kubectl"
```

2. Movem l'arxiu descarregat en el pas anterior a una ruta que es trobi dins la variable d'entorn *PATH* del sistema:

```
$ sudo mv kubectl /usr/local/bin/kubectl
```

3. Establirem permisos d'execució sobre el binari del client de Kubernetes:

```
$ chmod +x /usr/local/bin/kubectl
```

4. Finalment, comprovem que es pot executar el binari i ens assegurem que disposem de la versió 1.21:

Efectivament, la sortida anterior ens indica que la versió instal·lada és la que esperàvem. Cal dir que els passos anteriors únicament han instal·lat el client kubectl però manca afegir la configuració per a connectar al clúster de Kubernetes que, la tindrem disponible, després d'haver-lo desplegat en GCP.

5. Disseny, implementació i desplegament de la infraestructura

Després d'haver configurar l'equip de treball amb les utilitats que es necessitaran pel desenvolupament d'aquest projecte, ara és el torn de dissenyar i desplegar la infraestructura comuna sobre la qual s'executaran les aplicacions de la organització. En mode resum, podem dir que es treballarà en les següents àrees:

- Topologia de xarxa (adreçament IP, polítiques de seguretat, accés a Internet)
- Clúster de Kubernetes (*control plane* i nodes)
- Aplicacions CI/CD per automatitzar els desplegaments (repositoris de GitHub, Google Cloud Build i ArgoCD)

5.1. Topologia de xarxa

La plataforma Google Cloud Platform està distribuïda mundialment en diferents països per a donar resposta a les necessitats dels clients en temes de latència, tolerància a fallides i compliment de les legislacions de cada país o zona. Per les raons anteriors, GCP té organitzada la seva plataforma en regions i zones:

- **Regió:** una regió és un conjunt de zones que es troben dins d'una àrea geogràfica delimitada oferint un alt rendiment de xarxa i latència baixa entre aquestes. A data de la realització d'aquest treball, hi ha 24 regions disponibles.

- **Zona:** una zona és una àrea d'implementació dins d'una regió. Per a aplicacions amb tolerància a fallides es recomana utilitzar més d'una zona en la regió. La majoria de regions tenen 3 zones tot i que alguna en pot arribar a tenir fins 4.

La representació de les regions en el mapa es pot consultar en la captura de pantalla següent, extreta de la pàgina web https://cloud.google.com/about/locations.

Il·lustració 17 - Regions i zones de Google Cloud. Font: https://cloud.google.com/about/locations

Per a seleccionar la regió, comprovarem quina és la que ens dona un temps de resposta menor, és a dir, aquella regió que ofereixi menys latència amb Barcelona. Per a la realització del treball potser aquest detall no té importància però s'ha de tenir molt present a l'hora de dissenyar l'arquitectura d'una aplicació amb ús real. Utilitzarem la pàgina web https://gcping.com/ per obtenir la informació de les latències i, el resultat obtingut és:

REGION	MEDIAN LATENCY
Belgium europe-west1	47 ms
Zurich, Switzerland europe-west6	48 ms
Global HTTPS Load Balancer →europe-west1	49 ms
London, UK europe-west2	49 ms
Frankfurt, Germany europe-west3	49 ms
Netherlands europe-west4	55 ms
Warsaw, Poland europe-central2	62 ms

Així doncs, la regió que ens ofereix millor temps de resposta és ***europe-west-1***, ubicada a Bèlgica. Per a validar-la definitivament, ens haurem d'assegurar primer que tots els serveis que necessitarem es troben disponibles en aquesta regió, consultant aquesta URL: https://cloud.google.com/about/locations#europe. Després de revisar-ho, veiem que compleix amb els requeriments marcats així que passarà a ser la regió escollida pel projecte.

En GCP, l'element principal sobre el qual es creen el conjunt d'objectes de xarxa es coneix amb el nom de **VPC** (Virtual Private Cloud). Els objectes de xarxa que farem servir per la realització de la part pràctica, i que explicarem tot seguit, seran:

- **Projects:** els projectes, que hem creat anteriorment, associa els objectes i serveis utilitzats amb la facturació. A més a més, un projecte pot tenir com a molt 5 networks.

- **Networks:** són elements que agrupen els objectes de tipus subnetwork però no tenen associat directament cap adreçament de xarxa. Una network és global i es pot distribuir

entre regions, fent possibles que recursos de diferents regions es puguin comunicar entre si. Existeixen 3 tipus de networks:

- **Default:** és el tipus per defecte i crea un subnetwork per regió amb les firewall rule (polítiques d'accés) per defecte també.

- **Auto:** les network de tipus auto, creen automàticament una subxarxa preestablerta per regió, dins del rang 10.128.0.0/9, amb una màscara de xarxa 255.255.240.0 (/20), extensible fins a 255.255.0.0 (/16). És a dir, tenim una capacitat d'entre 4094 i 65534 d'adreces IPs, aproximadament.

- **Custom:** aquest tipus és el més flexible i el que permet configurar l'adreçament de xarxa de forma personalitzada en les subnetworks, podent escollir els rangs privats que desitgem.

- **Subnetworks:** és el bloc per definir l'adreçament de xarxa que serà utilitzat per distribuir adreces IPs als recursos, el qual, s'associarà a una network. Ha de complir amb l'estàndard RFC 1918 respecte a l'assignació de xarxes privades i té com a particularitat que es pot estendre però no reduir. També cal dir que GCP reserva un total de 4 adreces IPs en cada subnetwork, les dues primeres i les dues últimes del rang.

- **IP addresses:** les adreces IP en Google Cloud poden ser internes o externes. Les primeres, s'assignaran a partir de les subnetworks definides i, les segones, són adreces IPs públiques que assignarà automàticament Google llevat que realitzem una reserva per algun cas d'ús concret.

- **Routes:** aquest objecte és l'encarregat d'enrutar el tràfic de les subnetworks cap a altres xarxes, inclòs Internet.

- **Firewall rules:** les polítiques de seguretat d'accés es defineixen a partir de les firewall rules i afecten a tots els recursos d'una mateixa xarxa tot i que s'aplica a nivell d'instància destí. Per defecte, les regles deneguen qualsevol petició entrant i autoritzen el tràfic sortint. A més a més, tenen la propietat de ser *stateful*, és a dir, que si un dispositiu estableix la connexió amb un host remot, automàticament es permet el tràfic del host remot de manera bidireccional.

- **Cloud NAT:** aquest servei gestionat per GCP, ofereix connexió a Internet als recursos d'una subnetwork sense que aquests tinguin adreçament IP públic. Aquest element està indicat per a protegir i tenir una plataforma més segura.

5.1.1. Disseny de la topologia de xarxa

Un cop hem explicat els components de xarxa, passarem a la fase de disseny on definirem els següents:

- **Zones:** per temes d'alta disponibilitat i tolerància a fallides, utilitzarem les 3 zones disponibles a la regió europe-west-1.

- **Network:** crearem una network de tipus custom ja que volem tenir el control total sobre l'adreçament de xarxa:

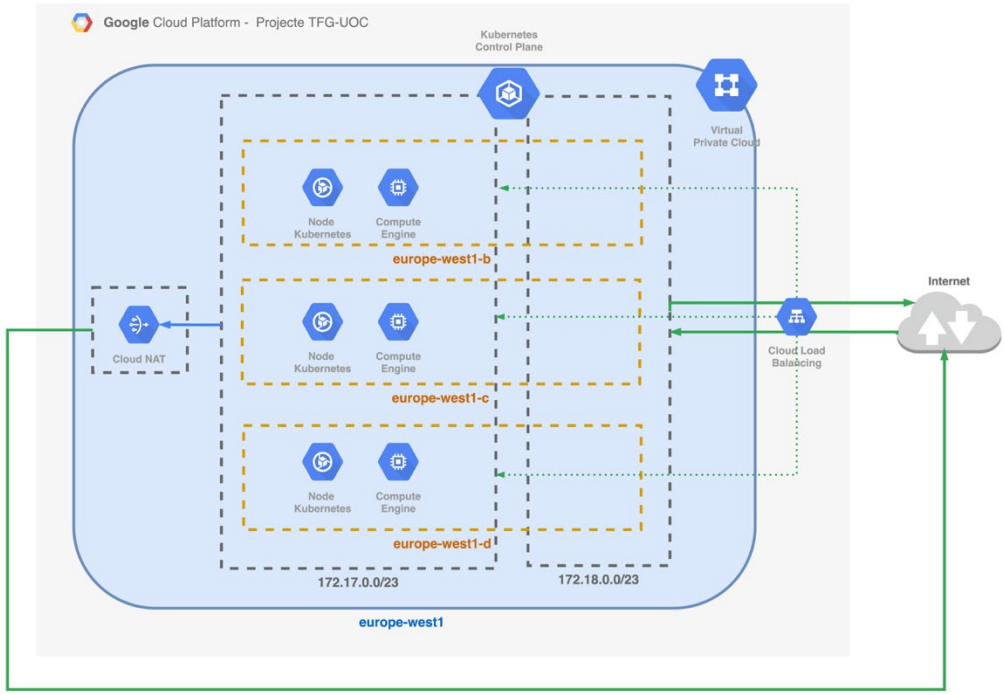
- **Subnetworks:** crearem una subnetwork 172.17.0.0/23 amb capacitat fins a 508 instàncies (512 menys les 4 que reserva GCP) d'adreçament privat i, una altra subnetwork, 172.18.0.0/23 pels elements amb adreçament públic. Aquesta separació la realitzem per tenir la possibilitat d'establir firewall rules per filtrar el tràfic i reforçar la seguretat. La taula següent mostra la informació descrita anteriorment d'una forma més visual:

Nom	CIDR block	Rang d'adreces IP	Total IPs
tfg-uoc-subnet-public	172.17.0.0/23	172.17.0.0 - 172.17.1.254	508
tfg-uoc-subnet-private	172.18.0.0/23	172.18.0.0 - 172.18.1.254	508

Tot i que no són segments de xarxa molt amplis, podríem estendre-la en un futur si fos necessari aprofitant la flexibilitat en aquest aspecte que ofereix GCP.

- **Cloud NAT:** desplegarem el servei Cloud NAT per habilitar l'accés a Internet a les instàncies que es trobin en la subnetwork privada.

El diagrama de l'arquitectura de la xarxa, segons s'ha explicat anteriorment, queda representat de la següent manera:



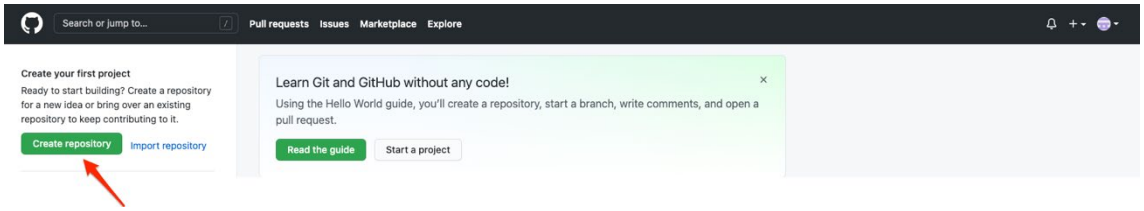
Il·lustració 18 - Arquitectura de xarxa GCP. Font: pròpia.

5.1.2. Implementació de la topologia de xarxa

Un cop tenim dissenyada la xarxa en GCP, passarem a implementar-ho generant el codi de terraform per a desplegar els recursos. Per seguir el principis de GitOps, haurem de crear un repositori de control de versions a GitHub per emmagatzemar tot el codi generat, seguint els passos següents:

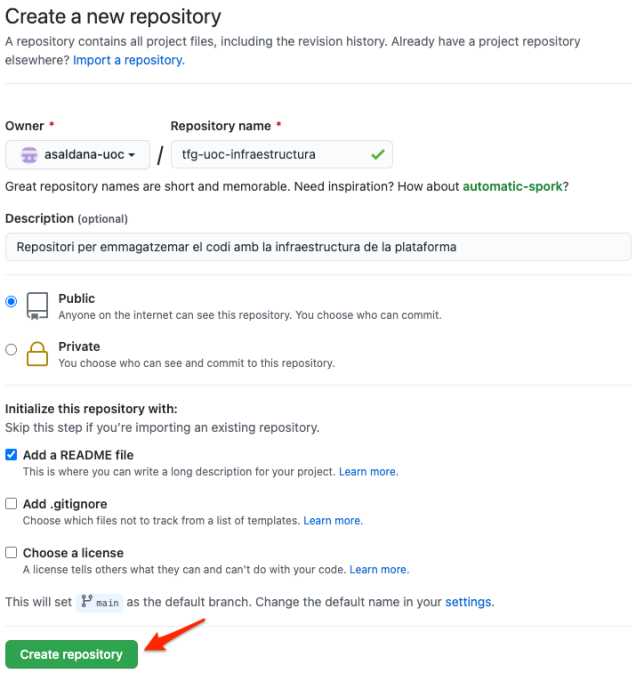
1. Obrir al navegador la URL https://github.com/ i autenticar-nos.

2. Seleccionarem "*Create repository*":

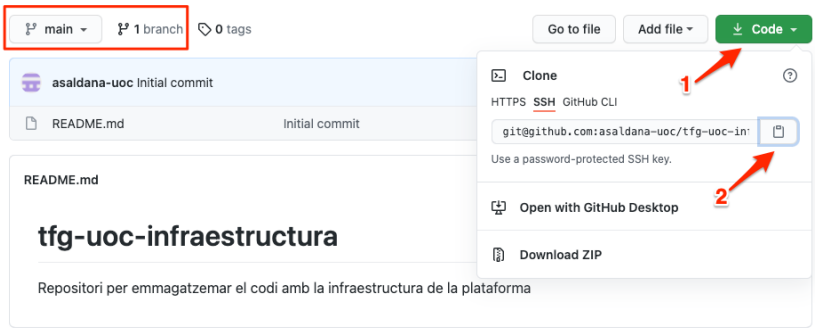


3. Introduïm les dades que ens sol·licita el formulari:

    a. **Nom de repositori:** *tfg-uoc-infraestructura*
    b. **Descripció:** Afegim un comentari descriptiu sobre el contingut del repositori.
    c. **Tipus:** el creem com a privat ja que es un projecte de proves que s'haurà d'exposar públicament. En el cas d'una organització privada, s'hauria de marcar com a privat.
    d. **Fitxers addicionals:** marquem que s'inicialitzi el repositori amb un arxiu README.



4. Veiem com el repositori s'ha creat amb la branca ***main*** per defecte.



5. Ara, clonarem el repositori en l'equip local per a començar a desenvolupar el codi copiant la URL que es veu en el pas 2 de la captura anterior:

```
$ git clone git@github.com:asaldana-uoc/tfg-uoc-infraestructura.git
```

```
~/repos git clone git@github.com:asaldana-uoc/tfg-uoc-infraestructura.git
Cloning into 'tfg-uoc-infraestructura'...
remote: Enumerating objects: 67, done.
remote: Counting objects: 100% (67/67), done.
remote: Compressing objects: 100% (37/37), done.
remote: Total 67 (delta 21), reused 57 (delta 14), pack-reused 0
Receiving objects: 100% (67/67), 11.09 KiB | 3.70 MiB/s, done.
Resolving deltas: 100% (21/21), done.
```

6. Per acabar, crearem un nou projecte en el IDE, GoLand en el nostre cas, i verificarem que disposem del *plugin* HCL per facilitar el desenvolupament del codi de terraform.

El codi generat per a implementar la topologia estarà disponible en aquest repositori de GitHub. Per a realitzar la implementació, s'han utilitzat mòduls públics pel proveïdor GCP que es troben al terraform registry https://registry.terraform.io/namespaces/terraform-google-modules. L'estructura de directoris i fitxers s'ha pensat tenint en compte la futura escalabilitat de la plataforma, fins i tot, en diversos proveïdors de cloud. Per tant el patró que es seguirà per a estructurar el codi serà *PROVEÏDOR/ENTORN/REGIO/SERVEI* tal com es pot veure en la captura següent:

```
~/r/tfg-uoc-infraestructura / main tree -d
.
└── gcp
    └── devel
        └── europe-west1
            ├── gke
            └── vpc
```

El fet de només tenir un entorn (*devel*) és perquè al tractar-se d'un projecte de proves no es vol desplegar més infraestructura per reduir els costos. En un entorn real, s'haurien de crear més directoris corresponents al número d'entorns que necessiti l'organització (*stage*, *pre-production*, *production*, etc..)

Pel que respecta als arxius de configuració de terraform, s'ha optat per dividir el codi segons el tipus de recurs que es generi. El codi complet, amb comentaris explicatius, està disponible en aquest enllaç però, a continuació, es detallarà la funció de cada fitxer:

- **locals.tf:** en aquest fitxer només trobem un recurs, anomenat ***locals***, que s'utilitza per a definir constants que utilitzarem per tenir consistència en els noms dels recursos.

- **provider.tf:** en aquest fitxer és defineix un recurs de tipus ***provider*** que és necessari per a que terraform conegui la plataforma (API) amb la que haurà d'interactuar. A més a més, es defineix un altre recurs anomenat ***backend*** per especificar on s'emmagatzemarà l'estat de terraform, en aquest cas a un bucket GCS.

- **main.tf:** és l'arxiu principal on es defineixen els recursos que es crearan en la plataforma. En la implementació, s'utilitzen dos mòduls del *registry* als quals se'ls passen els paràmetres amb els valors decidits en la fase de disseny. És important comentar que tenim especificada la versió del mòdul que volem utilitzar per evitar problemes futurs amb retro-compatibilitats. A més a més, en aquest codi, aprofitem les constants definides en l'arxiu `locals.tf` per establir els noms dels recursos utilitzant la funció format que ajuda a concatenar *strings* i definir una convenció.

5.1.3. Desplegament de la topologia de xarxa

Després d'haver acabat amb la implementació del codi comprovarem que no es genera cap error i que la topologia de xarxa està preparada per ser desplegada. Executarem els següents passos amb terraform:

1. Ubicats en el directori *vpc*, inicialitzarem terraform i descarregarem els mòduls:

```
$ terraform init
```

```
~/r/tfg-uoc-infraestructura/g/d/e/vpc / main terraform init
Initializing modules...
Downloading terraform-google-modules/cloud-router/google 1.0.0 for cloud-router...
- cloud-router in .terraform/modules/cloud-router
Downloading terraform-google-modules/network/google 3.2.2 for vpc...
- vpc in .terraform/modules/vpc
- vpc.firewall_rules in .terraform/modules/vpc/modules/firewall-rules
- vpc.routes in .terraform/modules/vpc/modules/routes
- vpc.subnets in .terraform/modules/vpc/modules/subnets
- vpc.vpc in .terraform/modules/vpc/modules/vpc

Initializing the backend...

Successfully configured the backend "gcs"! Terraform will automatically
use this backend unless the backend configuration changes.
```

2. La inicialització ha funcionat correctament, això significa que s'ha pogut descarregar l'arxiu d'estat que es troba al bucket GCS. Tot seguit, verificarem quins canvis estan pendent d'aplicar:

```
$ terraform plan
```

```
  # module.vpc.module.vpc.google_compute_network.network will be created
  + resource "google_compute_network" "network" {
      + auto_create_subnetworks         = false
      + delete_default_routes_on_create = false
      + gateway_ipv4                    = (known after apply)
      + id                              = (known after apply)
      + mtu                             = 0
      + name                            = "tfg-uoc-vpc"
      + project                         = "tfg-uoc-313418"
      + routing_mode                    = "GLOBAL"
      + self_link                       = (known after apply)
    }

Plan: 5 to add, 0 to change, 0 to destroy.

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take
exactly these actions if you run "terraform apply" now.
~/r/tfg-uoc-infraestructura/g/d/e/vpc / main / 4s
```

En la captura anterior, s'ha truncat la sortida perquè és molt extensa però podem veure un petit resum dels canvis que terraform té pendent d'aplicar. Evidentment, coincideix amb les previsions ja que hem creat 5 recursos: 1 network, 2 subnetworks, 1 router i 1 nat gateway.

3. Finalment, apliquem els canvis confirmant que estem d'acord com es podrà veure en la imatge següent:

```
$ terraform apply
```

```
Plan: 5 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

Per verificar que els recursos existeixen en GCP utilitzarem el client gcloud:

```
$ gcloud compute networks subnets list --network=tfg-uoc-vpc
```

```
~ gcloud compute networks subnets list --network=tfg-uoc-vpc                    TFG-UOC
NAME                    REGION        NETWORK      RANGE
tfg-uoc-subnet-private  europe-west1  tfg-uoc-vpc  172.17.0.0/23
tfg-uoc-subnet-public   europe-west1  tfg-uoc-vpc  172.18.0.0/23
```

Per tant, donem per finalitzada els desplegament de la topologia de xarxa. Posteriorment, quan estigui disponible l'eina CI/CD, automatitzarem els passos anteriors.

5.2. Clúster de Kubernetes

Un cop tenim la topologia de xarxa desplegada en la regió europe-west1, podrem desplegar el clúster de Kubernetes sobre el VPC. Com s'ha explicat anteriorment, el servei de GCP que ofereix un clúster de Kubernetes gestionat s'anomena Google Kubernetes Engine (GKE, a partir d'ara). Està disponible en dues modalitats operacionals:

- **Standard:** la modalitat estàndard proporciona un *control plane* gestionat per GCP però deleguen l'administració dels *nodes* de Kubernetes en el client. Aquesta opció ofereix més flexibilitat a l'hora de configurar el clúster de Kubernetes i, al mateix temps, ens permet tenir un control més estricte sobre els costos generats ja que es paga segons el número de nodes actius.

- **Autopilot:** aquest mode està totalment gestionat per GCP, tant el *control plane* com els nodes. L'avantatge principal és que redueix la càrrega de treball d'administració però, pel contrari, ens obliga a acceptar la configuració per defecte amb poca flexibilitat per a realitzar canvis. A més a més, la facturació és diferent que la versió estàndard ja que es paga pels recursos sol·licitats pels *pods* (CPU, memòria, disc).

Segons l'explicació anterior, pel desenvolupament d'aquesta pràctica optarem per l'opció *standard* perquè es busca total flexibilitat en el disseny al tractar-se d'un exercici pràctic amb el qual volem aprendre i explotar totes les funcionalitats que ofereix Kubernetes. A més a més, el

tema dels costos és important i, com en el cas anterior, es vol tenir control total en aquesta àrea. Les opcions que es poden personalitzar a l'hora de desplegar un clúster GKE, que s'explicaran amb major detall en l'apartat d'implementació, fan referència a les àrees següents:

- Versió i ubicació del clúster.
- Gestió dels nodes, conegut amb el nom de *node pool*.
- Aprovisionament de recursos.
- Tipus d'imatge pels nodes (bàsicament sistema operatiu i motor dels contenidors).
- Facturació.
- Gestió de la seguretat.
- Configuració de xarxa.
- Actualitzacions i manteniment.
- Autentificació i credencials.
- Autoescalat de càrregues de treball, tant de nodes com de pods (HPA[33]/VPA[34])
- Monitorització i gestió de logs.
- Instal·lació de funcionalitats addicionals al clúster.

5.2.1. Disseny de l'arquitectura de Kubernetes

Tot seguit, realitzarem el disseny i el diagrama de com encaixarà el clúster de Kubernetes i els nodes en el VPC configurat anteriorment. Primerament, s'ha de tenir present que s'haurà d'assignar dos segments de xarxa addicionals a la topologia existent ja que el propi clúster de Kubernetes té un adreçament IP intern propi: una xarxa dedicada als recursos de tipus *Service* i una altra xarxa específica pels recursos *Pods*. Tenint en compte que el número de pods serà amb total probabilitat molt més elevat que el número de serveis, assignarem un adreçament IPs més ampli en el primer cas, quedant de la següent manera:

Nom	CIDR block	Rang d'adreces IP	Total IPs
tfg-uoc-subnet-private-k8s-pods	10.10.0.0/20	10.10.0.1 - 10.10.15.254	4094
tfg-uoc-subnet-private-k8s-services	10.20.0.0/22	10.20.0.1 - 10.20.3.254	1022

Després d'haver assignat l'adreçament de xarxa continuarem definint el tipus d'instàncies que utilitzarem pels nodes. En les plataformes cloud, els tipus d'instàncies es classifiquen en famílies oferint millor rendiment en les càrregues de treball pel qual han estat dissenyades adaptant-se així a diferents casos d'ús. A més a més, dins de cada família, existeixen diferents tipus d'instàncies en funció dels recursos que ofereixen (vCPU, memòria, disc, GPU, etc...), evidentment, tenint un cost adaptat a aquests paràmetres. En GCP, la classificació de les famíles és la següent:

- **Ús general:** són les més balancejades en rendiment oferint una relació qualitat/preu òptima. Dins d'aquest tipus tenim les instàncies de tipus E2, N2, N2D i N1.

[33] *HPA: Horizontal Pod autoscaling.* https://cloud.google.com/kubernetes-engine/docs/how-to/horizontal-pod-autoscaling
[34] *VPA: Vertical Pod autoscaling.* https://cloud.google.com/kubernetes-engine/docs/how-to/vertical-pod-autoscaling

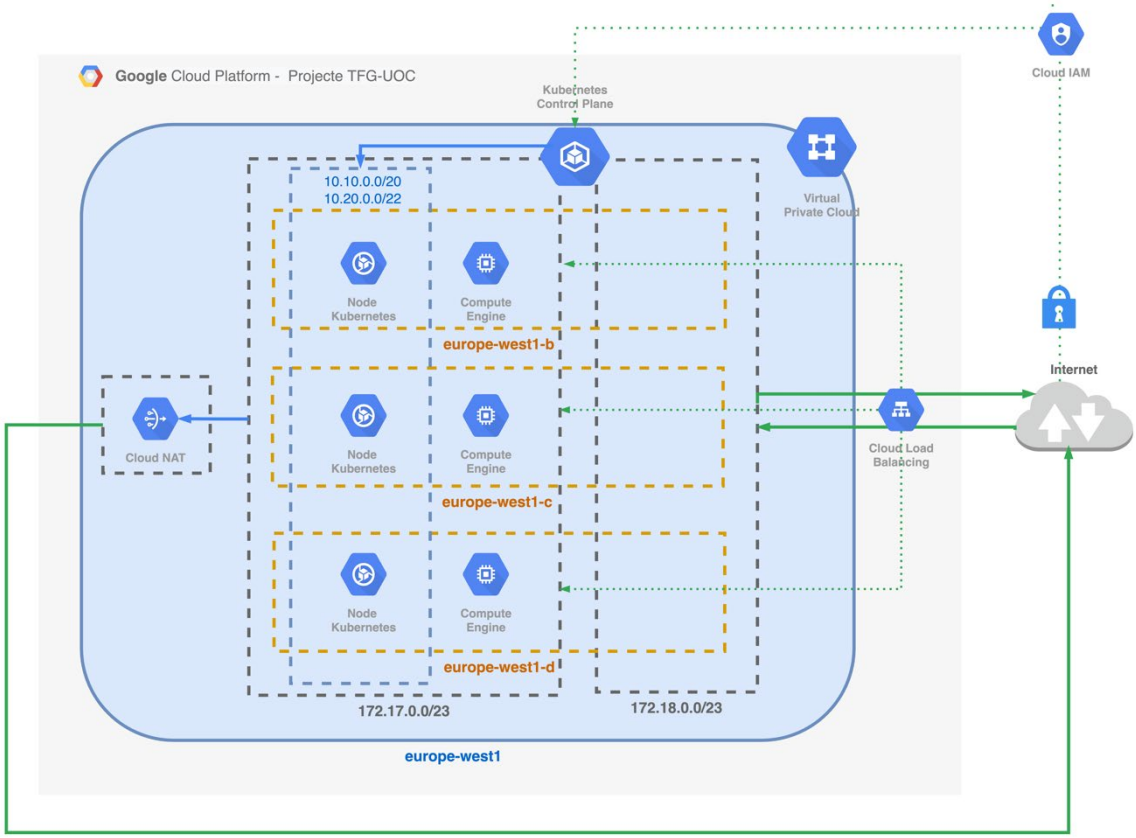
- **Optimitzades per computació:** instàncies específiques per càrregues de treball intensives d'operacions que requereixen recursos de computació. Formen part d'aquest grup els tipus d'instàncies C2.

- **Optimitzades per memòria:** instàncies específiques per càrregues de treball que necessiten molts recursos de memòria RAM. Formen part d'aquest grup els tipus d'instàncies M1 i M2.

- **Optimitzades per Machine Learning:** instàncies que disposen d'una targeta d'acceleració dedicada NVIDIA per dur a terme anàlisis de *Machine Learning*. Formen part d'aquest grup els tipus d'instàncies A2.

Per al cas d'ús d'aquest projecte optarem per les instàncies d'ús general de tipus ***e2-small*** que disposen de 2vCPU compartides i 2GB de memòria. En desplegarem 3 instàncies d'aquest tipus, una a cada zona de la regió europe-west1, per a provar l'alta disponibilitat i la tolerància a fallides. De totes maneres, en cas de detectar problemes de rendiment, el canvi a una instància amb majors prestacions seria un canvi senzill de realitzar ja que només caldria modificar el repositori on tenim el codi de la infraestructura i aplicar terraform.

Respecte al control d'accés i la seguretat, el *control plane* estarà protegit per les credencials pròpies que té per defecte però, a més a més, s'afegirà una capa addicional de seguretat a nivell de xarxa autoritzant únicament l'accés a l'endpoint a IPs públiques conegudes i de confiança. Per tant, el diagrama de l'arquitectura de Kubernetes queda representat en la següent imatge:



5.2.2. Implementació de l'arquitectura de Kubernetes

A continuació, passarem a traduir el disseny de l'arquitectura del diagrama a codi de terraform per a poder desplegar el clúster de Kubernetes segons les especificacions descrites. Utilitzarem el mateix repositori de GitHub creat anteriorment, https://github.com/asaldana-uoc/tfg-uoc-infraestructura, però afegirem una nova carpeta que anomenarem *gke*. Tanmateix, utilitzarem el mòdul oficial que es trobà al registry de terraform per facilitar la implementació, disponible en aquesta URL.

Primerament, haurem d'afegir les subnetworks secundàries, que utilitzaran els pods i els services de Kubernetes, en el codi de terraform del VPC que es troba al fitxer *main.tf*:

Un cop afegit, verifiquem els canvis que introduirà terraform a la topologia de xarxa:

```
$ terraform plan
```

```
  # module.vpc.module.subnets.google_compute_subnetwork.subnetwork["europe-west1/tfg-uoc-subnet-priv
ate"] will be updated in-place
  ~ resource "google_compute_subnetwork" "subnetwork" {
        id                       = "projects/tfg-uoc-313418/regions/europe-west1/subnetworks/tfg-u
oc-subnet-private"
        name                     = "tfg-uoc-subnet-private"
      ~ secondary_ip_range       = [
          + {
              + ip_cidr_range = "10.10.0.0/20"
              + range_name    = "tfg-uoc-subnet-private-k8s-pods"
            },
          + {
              + ip_cidr_range = "10.20.0.0/22"
              + range_name    = "tfg-uoc-subnet-private-k8s-services"
            },
        ]
        # (9 unchanged attributes hidden)
    }

Plan: 0 to add, 2 to change, 0 to destroy.


Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take
exactly these actions if you run "terraform apply" now.
```

Efectivament, veiem que els canvis que apareixen són els que hem afegit així que apliquem terraform i verifiquem que així apareix a GCP:

```
$ terraform apply
$ gcloud compute networks subnets describe tfg-uoc-subnet-private --region=europe-
west1
```

```
~ gcloud compute networks subnets describe tfg-uoc-subnet-private --region=europe-west1
creationTimestamp: '2021-05-25T15:22:44.898-07:00'
enableFlowLogs: false
fingerprint: K1CmDS_iwf0=
gatewayAddress: 172.17.0.1
id: '641133269227718235'
ipCidrRange: 172.17.0.0/23
kind: compute#subnetwork
logConfig:
  enable: false
name: tfg-uoc-subnet-private
network: https://www.googleapis.com/compute/v1/projects/tfg-uoc-313418/global/networks/tfg-uoc-vpc
privateIpGoogleAccess: true
privateIpv6GoogleAccess: DISABLE_GOOGLE_ACCESS
purpose: PRIVATE
region: https://www.googleapis.com/compute/v1/projects/tfg-uoc-313418/regions/europe-west1
secondaryIpRanges:
- ipCidrRange: 10.10.0.0/20
  rangeName: tfg-uoc-subnet-private-k8s-pods
- ipCidrRange: 10.20.0.0/22
  rangeName: tfg-uoc-subnet-private-k8s-services
selfLink: https://www.googleapis.com/compute/v1/projects/tfg-uoc-313418/regions/europe-west1/subnetw
orks/tfg-uoc-subnet-private
```

Tot seguit, llegirem la documentació del submòdul de terraform per a crear un clúster privat de Kubernetes que podem trobar en aquest enllaç. La secció sobre els requeriments veiem que es

necessiten permisos de IAM addicionals pel compte de servei que utilitza terraform. Per aquesta raó, afegirem els permisos que indica la documentació:

```
$ gcloud projects add-iam-policy-binding tfg-uoc-313418 \
      --member='serviceAccount:tf-user@tfg-uoc-313418.iam.gserviceaccount.com' \
      --role='roles/iam.serviceAccountAdmin'
$ gcloud projects add-iam-policy-binding tfg-uoc-313418 \
      --member='serviceAccount:tf-user@tfg-uoc-313418.iam.gserviceaccount.com' \
      --role='roles/iam.serviceAccountUser'
$ gcloud projects add-iam-policy-binding tfg-uoc-313418 \
      --member='serviceAccount:tf-user@tfg-uoc-313418.iam.gserviceaccount.com' \
      --role='roles/resourcemanager.projectIamAdmin'
```

A més a més, haurem d'habilitar l'API del servei Cloud Resource Manager ja que és una dependència pel funcionament del submòdul de terraform:

```
$ gcloud services enable cloudresourcemanager.googleapis.com
```

```
~ gcloud services enable cloudresourcemanager.googleapis.com ✔ 6s TFG-UOC G
Operation "operations/acf.p2-1054054397089-260e6afc-6650-443b-8590-2970b3fa3702" finished successful
ly.
```

L'estructura de directoris i fitxers per aquest servei segueix el mateix criteri emprat per desplegar la topologia de xarxa tot i que s'han generat més arxius de configuració com es pot veure en la captura següent:

```
~/r/tfg-uoc-infraestructura/g/d/e/gke main tree ✔
.
├── locals.tf
├── main.tf
├── provider.tf
├── variables.tf
└── variables.tfvars

0 directories, 5 files
```

Pel que respecta als arxius de configuració de terraform, s'ha seguit amb l'estratègia anterior dividint el codi segons el tipus de recurs que es generi però, en aquesta ocasió, s'ha requerit afegir variables d'entrada. Tot i que el codi es pot consultar en GitHub en aquest enllaç, es detallarà mínimament la funció de cada fitxer:

- **locals.tf:** en aquest fitxer només trobem un recurs, anomenat ***locals***, que s'utilitza per a definir constants que utilitzarem per tenir consistència en els noms dels recursos. Aquesta pràctica redueix el errors potencials a l'hora d'establir noms als recursos.

- **variables.tf:** aquest arxiu de configuració ens ofereix la possibilitat d'introduir l'adreça IP que volem autoritzar l'accés a l'endpoint públic del *control plane*. És útil ja que, al tenir adreçament dinàmic a casa, ens permet canviar fàcilment la política de seguretat interactivament.

- **variables.tfvars:** aquest arxiu no es pujarà a GitHub, es troba a l'arxiu ***.gitignore***, perquè conté dades sensibles però ens servirà per a definir el valor de la variable anterior per a que sigui llegit automàticament per terraform i així evitar, per comoditat, introduir l'adreça IP.

- **main.tf:** aquí tenim la implementació pròpia del clúster de Kubernetes, tant control plane com nodes, tal com s'havia dissenyat en l'apartat anterior. Cal comentar que hi ha dos recursos previs (*data* i el *provider kubernetes*) que requereix el submòdul emprat. Descriurem les opcions escollides més destacables:

  - Tant el clúster com el pool de nodes seran de tipus regionals, és a dir, estaran distribuïts en múltiples zones per temes de tolerància a fallades.
  - Es força l'ús de les xarxes específiques, creades anteriorment, per aquest servei.
  - Es crea un *service account* per a que els nodes puguin crear i modificar certs recursos a GCP com, per exemple, els balancejadors.
  - S'afegeix la política de seguretat per a permetre descarregar imatges de Docker des de Google Container Registry sobre la *service account* anterior.
  - S'ha establert un límit inicial de 100 pods per node.
  - Els nodes no seran accessibles públicament i el *control plane* únicament des d'una adreça IP (la que es passa com a paràmetre d'entrada)
  - Com a mínim hi haurà un node per zona i com a màxim 2.
  - El motor d'execució dels contenidors serà *ContainerD*, el nou futur estàndard de Kubernetes, a través del paràmetre ***image_type***.
  - Els nodes tindran una capacitat de 75GB d'emmagatzemament.
  - S'han activat les *network policies* de Kubernetes amb Calico.

- **provider.tf:** aquest arxiu és pràcticament idèntic al cas del VPC a excepció del fitxer de terraform estat que en aquest cas canvia (*prefix*).

5.2.3. Desplegament de l'arquitectura de Kubernetes

Una vegada tenim el codi generat, podrem executar terraform per a poder desplegar el clúster de Kubernetes tal com hem definit:

```
$ terraform apply
```

En aquesta ocasió comprovem com abans de començar a aplicar la configuració ens pregunta pel valor de la variable ***allowed_ip_access_control_plane*** així que la introduïm i confirmem:

```
~/r/tfg-uoc-infraestructura/g/d/e/gke / main !1 terraform apply
var.allowed_ip_access_control_plane
  Enter a value:
```

Es veu com el codi generat crearà 10 recursos nous en la plataforma GCP, acceptem i esperem que el procés finalitzi.

```
Plan: 10 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

```
l_resource.module_depends_on[0]: Creating...
module.gke-private-cluster.module.gcloud_delete_default_kube_dns_configmap.module.gcloud_kubectl.nul
l_resource.module_depends_on[0]: Creation complete after 0s [id=5269520578621943343]

Apply complete! Resources: 10 added, 0 changed, 0 destroyed.
~/r/tfg-uoc-infraestructura/g/d/e/gke main !1 ✔ 8m 25s
```

Finalment, després d'uns 8 minuts aproximadament, el desplegament ha finalitzat i tan sols quedarà comprovar que podem accedir al clúster. Per a configurar el client *kubectl* ho farem directament a través del client de GCP:

```
$ gcloud container clusters get-credentials tfg-uoc-k8s-cp --region europe-west1 --
project tfg-uoc-313418
```

```
~ gcloud container clusters get-credentials tfg-uoc-k8s-cp --region europe-west1 --project tfg-
uoc-313418
Fetching cluster endpoint and auth data.
kubeconfig entry generated for tfg-uoc-k8s-cp.
```

Comprovem que s'ha generat correctament l'arxiu de configuració a ***~/.kube/config*** i intentem obtenir els detalls del clúster creat en els passos anteriors:

```
$ kubectl cluster-info
```

```
~ kubectl cluster-info ✔ tfg-uoc-k8s-cp *
Kubernetes control plane is running at https://34.
calico-typha is running at https://34./api/v1/namespaces/kube-system/services/calico-typha:c
alico-typha/proxy
GLBCDefaultBackend is running at https://34./api/v1/namespaces/kube-system/services/default-
http-backend:http/proxy
KubeDNS is running at https://34./api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
Metrics-server is running at https://34./api/v1/namespaces/kube-system/services/https:metric
s-server:/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Efectivament, podem connectar a *l'endpoint* del *control plane* amb la qual cosa podem confirmar que tant el clúster com la configuració del client kubectl són correctes. Per finalitzar aquesta secció, ens assegurarem que tant els nodes com els *pods* i els *services* estan utilitzant l'adreçament privat que hem definit:

1. Verifiquem que tenim 3 nodes en execució.

```
$ kubectl get nodes
```

```
~ kubectl get nodes ✔ tfg-uoc-k8s-cp *
NAME                                                STATUS   ROLES    AGE   VERSION
gke-tfg-uoc-k8s-cp-tfg-uoc-k8s-nodes-3df7dfb1-rvsr  Ready    <none>   10m   v1.19.10-gke.1600
gke-tfg-uoc-k8s-cp-tfg-uoc-k8s-nodes-5ce10e08-96z5  Ready    <none>   10m   v1.19.10-gke.1600
gke-tfg-uoc-k8s-cp-tfg-uoc-k8s-nodes-8029ff7b-qr6w  Ready    <none>   17m   v1.19.10-gke.1600
```

2. Verifiquem que els nodes tenen una IP del rang 172.17.0.0/23.

```
$ kubectl describe node gke-tfg-uoc-k8s-cp-tfg-uoc-k8s-nodes-654a301b-vt8g |grep -i
internalip
```

```
~ kubectl describe node gke-tfg-uoc-k8s-cp-tfg-uoc-k8s-nodes-3df7dfb1-rvsr |grep -i internalip
  InternalIP:   172.17.0.5
```

3. Creem un pod per veure que l'adreça IP pertany al bloc 10.10.0.0/20 i que la adreça IP de sortida a Internet és la Cloud NAT:

```
$ kubectl run -it --rm --image=centos centos -- bash
```

```
kubectl run -it --rm --image=centos centos -- bash
If you don't see a command prompt, try pressing enter.
[root@centos /]# yum install -yq net-tools
Failed to set locale, defaulting to C.UTF-8
warning: /var/cache/dnf/baseos-f6a80ba95cf937f2/packages/net-tools-2.0-0.52.20160912git.el8.x86_64.r
pm: Header V3 RSA/SHA256 Signature, key ID 8483c65d: NOKEY
Importing GPG key 0x8483C65D:
 Userid     : "CentOS (CentOS Official Signing Key) <security@centos.org>"
 Fingerprint: 99DB 70FA E1D7 CE22 7FB6 4882 05B5 55B3 8483 C65D
 From       : /etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial
[root@centos /]# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1460
        inet 10.10.0.13  netmask 255.255.255.255  broadcast 10.10.0.13
        inet6 fe80::cc96:b8ff:fe3b:f522  prefixlen 64  scopeid 0x20<link>
        ether ce:96:b8:3b:f5:22  txqueuelen 0  (Ethernet)
        RX packets 1199  bytes 9418681 (8.9 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1066  bytes 83789 (81.8 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

[root@centos /]# curl curlmyip.net
192.158.30.248
```

Revisem els logs del servei Cloud NAT Gateway per comprovar que l'adreça IP coincideix:

```
labels: {
 nat.googleapis.com/instance_zone: "europe-west1-c"
 nat.googleapis.com/network_name: "tfg-uoc-vpc"
 nat.googleapis.com/subnetwork_name: "tfg-uoc-subnet-private"
 nat.googleapis.com/instance_name: "gke-tfg-uoc-k8s-cp-tfg-uoc-k8s-nodes-8029ff7b-qr6w"
 nat.googleapis.com/nat_ip: "192.158.30.248"
 nat.googleapis.com/router_name: "tfg-uoc-router"
}
logName: "projects/tfg-uoc-313418/logs/compute.googleapis.com%2Fnat_flows"
```

4. Verifiquem que els *services* de Kubernetes utilitzen un adreça IP del rang 10.20.0.0/22:

```
$ kubectl get services --all-namespaces
```

```
kubectl get services --all-namespaces
NAMESPACE     NAME                   TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
default       kubernetes             ClusterIP   10.20.0.1      <none>        443/TCP          27m
kube-system   calico-typha           ClusterIP   10.20.3.38     <none>        5473/TCP         26m
kube-system   default-http-backend   NodePort    10.20.2.115    <none>        80:32041/TCP     26m
kube-system   kube-dns               ClusterIP   10.20.0.10     <none>        53/UDP,53/TCP    26m
kube-system   metrics-server         ClusterIP   10.20.3.188    <none>        443/TCP          26m
```

Com s'ha pogut comprovar en les captures anteriors, totes les proves han sigut satisfactòries així que podem donar per validat el clúster de Kubernetes.

5.3. Estratègia d'integració i desplegament continu

En aquesta secció introduirem les eines que s'encarregaran d'executar els processos d'integració i desplegament continu, tant de la infraestructura que s'ha implementat anteriorment com de l'aplicació de prova que es desenvoluparà el cas pràctic en l'últim apartat.

5.3.1. Integració continua

En tots dos casos, utilitzarem Google Cloud Build per a realitzar tasques que pertanyen a la fase d'integració continua com són l'execució de tests o la construcció d'imatges de contenidors. En el cas de la infraestructura, també l'utilitzarem com a eina per a realitzar desplegaments a la plataforma GCP amb terraform perquè, com es necessiten permisos específics, es considera més segura aquesta solució ja que permet l'ús de comptes de serveis i evita haver de distribuir credencials en altres utilitats terceres que no siguin pròpies de GCP. Les característiques i funcionalitats que ens ofereix Google Cloud Build són les següents:

- Servei completament *serverless*, és a dir, no requereix desplegar instàncies dedicades.
- Suport per múltiples llenguatges de programació com Node.js, Python, Golang, Java, etc...
- Integració completa amb GitHub a partir de la subscripció a esdeveniments.
- Suport natiu per Docker.
- Cost baix, amb capa gratuïta de fins a 120 minuts d'execució al dia.

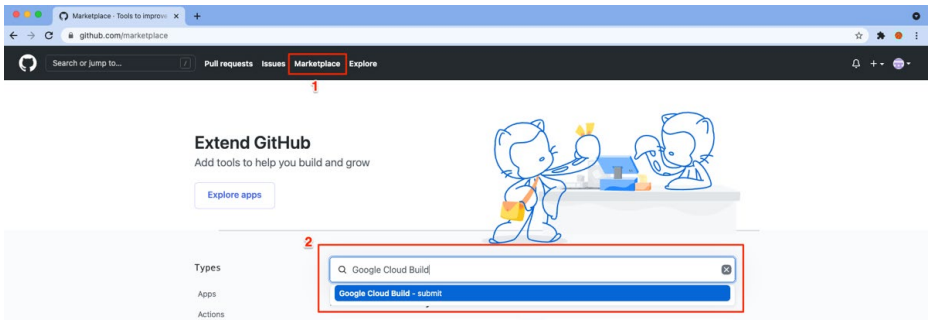
5.3.1.1. Integració continua: configuració de Google Cloud Build

Per començar a afegir tasques d'integració continua amb Google Cloud Build haurem d'activar el servei a GCP, en cas que no ho estigui:

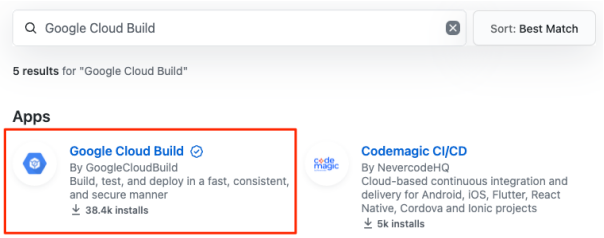
```
$ gcloud services enable cloudbuild.googleapis.com
```

Tot seguit, instal·larem l'aplicació Google Cloud Build a GitHub seguint els passos següents:

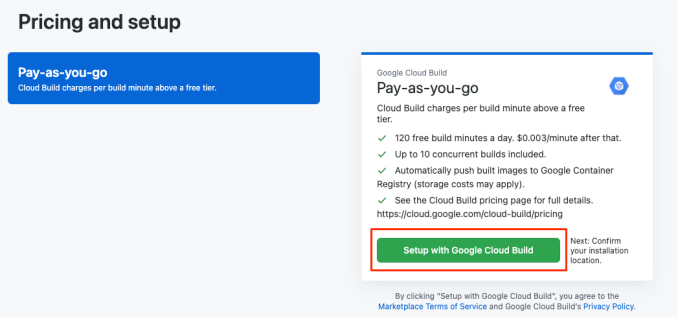
1. Anirem al menú "*Marketplace*" de GitHub i en el quadre de cerca introduirem "*Google Cloud Build*":



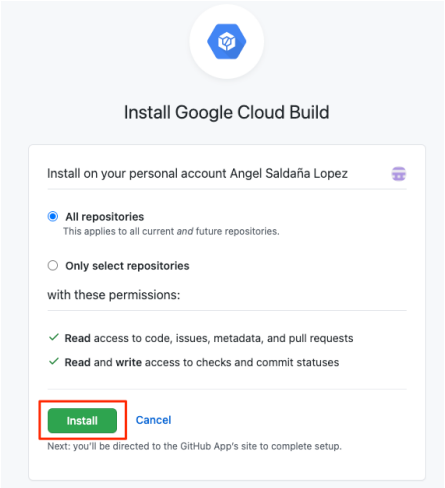
2. Seleccionarem l'aplicació "*Google Cloud Build*" que ha trobat el cercador:



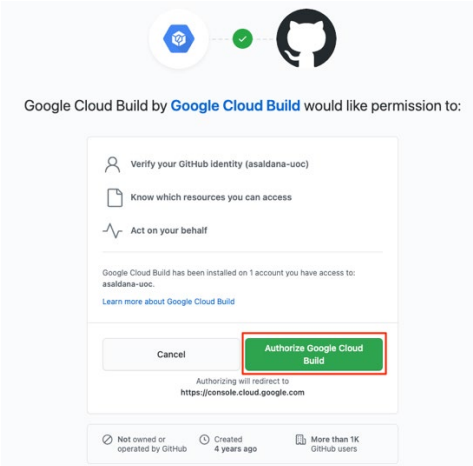
3. Ens desplaçarem a la part inferior de la pàgina i clicarem sobre el botó "*Setup with Google Cloud Build*":



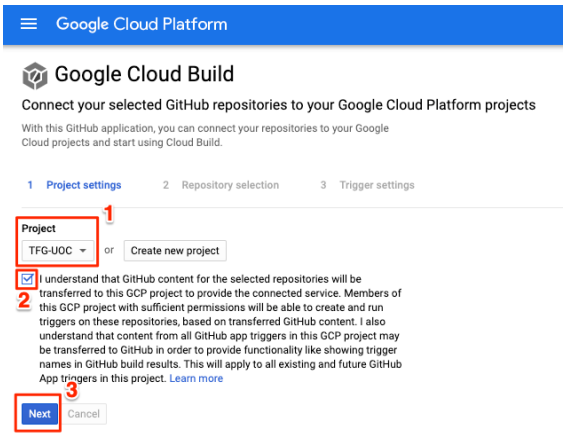
4. Escollirem l'opció de donar accés a tots els repositori que tinguem en el compte personal de GitHub i seleccionarem "*Install*":



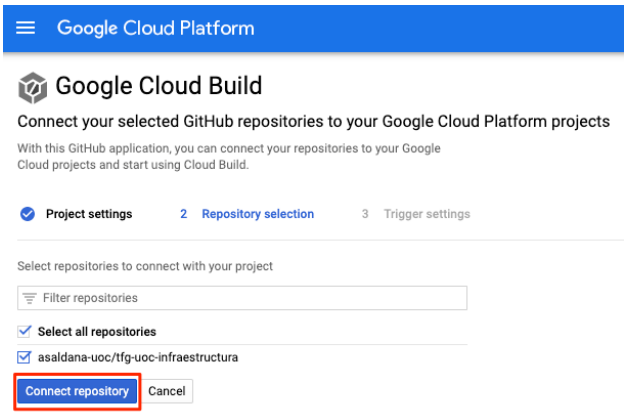
5. Confirmem les credencials de GitHub i, en la pantalla següent, haurem d'autoritzar l'accés a GitHub de Google Cloud Build:



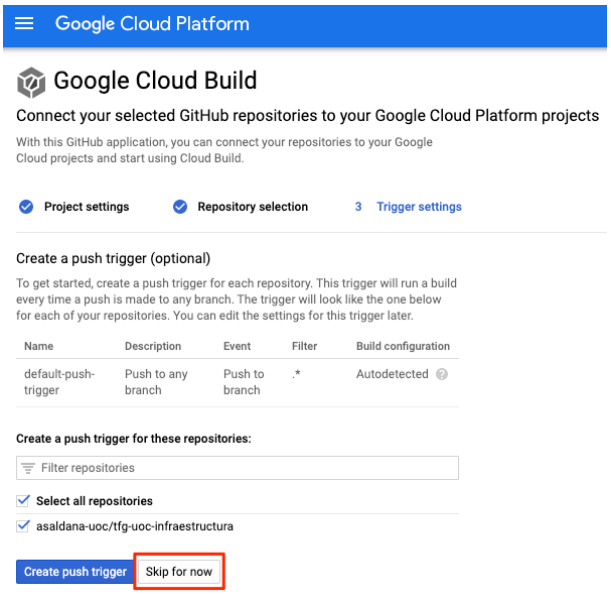
6. Ja situats en GCP, seleccionarem el projecte que vincularem a GitHub:



7. Tot seguit, escollirem entre els repositoris de GitHub que hi ha disponibles al perfil del compte personal, en aquest cas, únicament el que conté el codi de la infraestructura:



8. En l'últim pas no crearem de moment cap acció a executar quan es detecti un nou esdeveniment en el repositori de GitHub ja que aquesta part la realitzarem després:



5.3.1.2. Integració continua: configuració repositori d'infraestructura

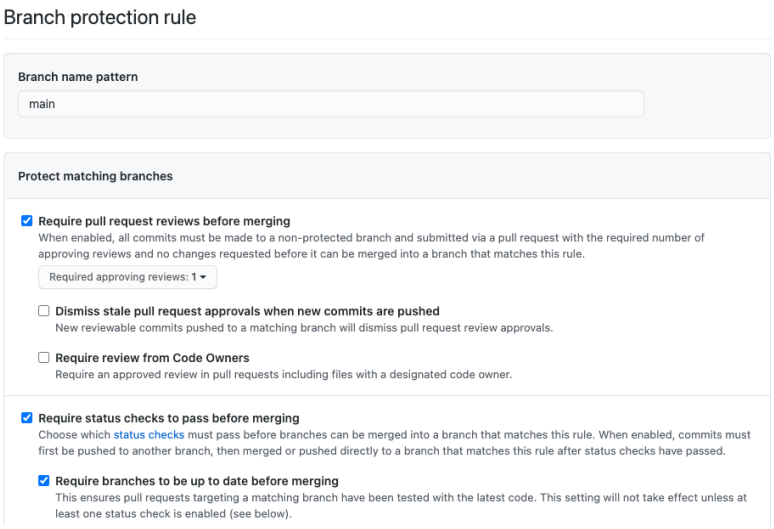
Per posar en marxa els tests d'integració continua en el repositori d'infraestructura, primerament haurem de realitzar modificacions en el repositori de GitHub per tal de protegir la branca principal, en el cas d'aquest projecte ***main***. Amb aquesta restricció, el que forçarem és que els canvis que es vulguin introduir en el repositori hauran de passar per unes comprovacions prèvies, mitjançant la creació d'una "*Pull Request*", abans de poder ser fusionats. Per aplicar aquesta configuració, haurem d'afegir una nova regla al repositori utilitzant la web de GitHub accedint a "*Settings*" > "*Branches*" > "*Add Rule*" seleccionant les següents opcions:

- ***Require pull request reviews before merging:*** aquesta característica fa referència a la revisió manual del codi per part d'una altra persona, la qual, haurà d'aprovar els canvis. Definim com a llindar una única persona perquè estem en un entorn de proves (en un entorn real aquest valor s'ha d'ajustar en funció del número de membres en l'equip).

- ***Require status checks to pass before merging:*** aquest opció és justament els tests que s'executaran a Google Cloud Build. Si els tests retornen algun error, s'impedirà que els canvis puguin ser afegits a la branca destinació.

- ***Require branch to be up to date before merging:*** el contingut de la branca que origina el canvi ha d'estar actualitzada amb la branca destí. En cas contrari, no es podrà integrar el codi d'ambdues branques.

La configuració en la plataforma, queda de la següent forma:



A continuació, haurem d'implementar les comprovacions que volem realitzar en cada "*Pull Request*" i el *trigger* de Google Cloud Build responsable d'executar automàticament aquests tests. La definició dels passos que ha de realitzar una funció de Google Cloud Build es realitza en un arxiu YAML i la documentació que s'ha seguit per a generar el codi d'aquest recurs es troba en aquesta pàgina web. Aquesta utilitat, es basa en molts dels conceptes que s'utilitzen en Docker per a especificar les propietats dels contenidors, de les quals, s'han utilitzat les següents:

- ***Steps:*** és el component inicial a partir del qual es defineix les accions que s'executaran.

- ***Name:*** el nom de la imatge de Docker sobre la qual s'executarà la tasca. En el cas d'aquest projecte, farem servir les imatges oficials de Hashicorp per la versió de terraform que estem utilitzant (0.15.3)
- ***Entrypoint:*** serveix per sobreescriure la comanda inicial que executa per defecte la imatge de Docker.
- ***Args:*** conté el conjunt de paràmetres que se li passa a la comanda inicial o al *entrypoint* si aquest està definit.
- ***Timeout:*** és el temps màxim que pot estar executant-se el *trigger*. S'ha definit un període de 60s per evitar superar la capa gratuïta diària de 120 minuts en cas d'error (bucle infinit per exemple)

El codi complet generat s'adjuntarà amb la memòria d'aquesta pràctica tot i que es pot consultar en aquest enllaç. De totes maneres, a manera de resum, la implementació consisteix en afegir els següents tests de validació en cada "*Pull Request*":

- ***Comprovació del format:*** el format del codi facilita la llegibilitat a l'hora de revisar-lo així que es fa ús de la funcionalitat de terraform que garanteixi que es segueixen els estàndards definits respecte la indentació, espais utilitzats, etc...

- ***Validació del codi i sintaxis:*** aprofitant la comanda "*terraform validate*", que valida la sintaxis correcta del codi generat, afegirem aquest test per a que verifiqui de manera recursiva en tots els directoris els fitxers de terraform per a que no s'introdueixi cap error en la branca principal *main*.

Un cop tenim definit els tests de les "*Pull Requests*" per la infraestructura queda pendent generar el codi de terraform per a crear el *trigger* anterior en GCP. Bàsicament, s'ha de crear un recurs de tipus `google_cloudbuild_trigger`, documentat en la web de terraform, amb les propietats que desitgem, que en cas actual serà:

- ***Name:*** el nom que volem pel recurs a crear, que serà `tfg-uoc-ci-infraestructura`.

- ***Filename:*** la ubicació del fitxer YAML creat anteriorment. Per defecte, Google Cloud Build busca un arxiu anomenat `cloudbuild.yaml` a l'arrel del repositori però en el nostre cas ho modificarem per a què utilitzi l'arxiu ubicat en la ruta següent: `gcp/common/ci/infraestructura/cloudbuild.yaml`.

- ***GitHub:*** la configuració pròpia de GitHub i quines operacions es realitzaran davant d'un esdeveniment. Pel cas d'ús actual, seleccionarem que s'executi automàticament sobre cada *commit* nou en cada una de les "*Pull Requests*" (noves i existents). A més a més, es podrà re-executar fàcilment afegint un comentari amb el contingut "`/gcbrun`".

El codi de terraform, com en el cas anterior, s'adjuntarà amb la memòria del treball però es pot consultar al repositori de GitHub accedint a aquest enllaç. Després d'haver-lo creat, desplegarem el *trigger* aplicant terraform:

```
$ cd gcp/common/ci/infraestructura && terraform apply
```

```
Terraform will perform the following actions:

  # google_cloudbuild_trigger.ci_trigger will be created
  + resource "google_cloudbuild_trigger" "ci_trigger" {
      + create_time = (known after apply)
      + description = "Trigger que s'executarà en cada Pull Request nova"
      + filename    = "gcp/ci/cloudbuild.yaml"
      + id          = (known after apply)
      + name        = "tfg-uoc-infraestructura-ci"
      + project     = (known after apply)
      + trigger_id  = (known after apply)

      + github {
          + name  = "tfg-uoc-infraestructura"
          + owner = "asaldana-uoc"

          + pull_request {
              + branch          = ".*"
              + comment_control = "COMMENTS_ENABLED_FOR_EXTERNAL_CONTRIBUTORS_ONLY"
            }
        }
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

Un cop desplegada crearem una branca nova amb un error de sintaxis i, a partir d'aquesta branca, crearem una "*Pull Request*" que hauria de prohibir que puguem fusionar els canvis amb la branca *main* perquè no ha passat amb els tests de verificació.

1. Situats en el repositori, crearem una branca de git que anomenarem *TFG-Test_CI_Infra*:

```
$ git checkout -b TFG-Test_CI_Infra
```

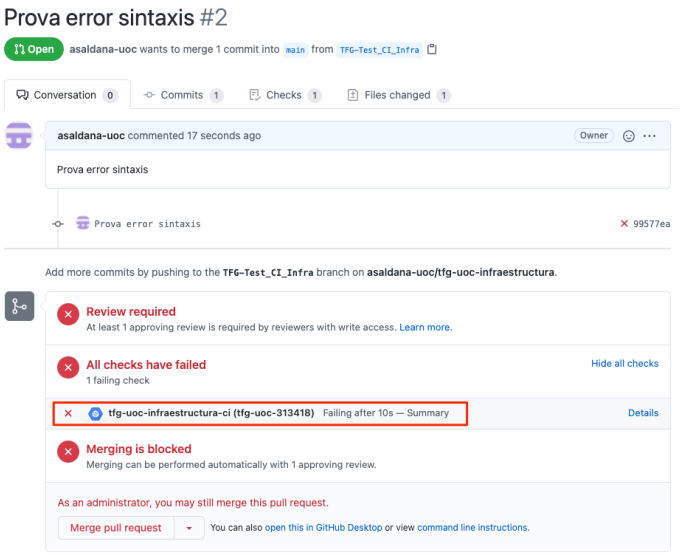
```
~/r/tfg-uoc-infraestructura/g/ci / main git checkout -b TFG-Test_CI_Infra ✔
Switched to a new branch 'TFG-Test_CI_Infra'
~/r/tfg-uoc-infraestructura/g/ci / TFG-Test_CI_Infra ✔
```

2. Generarem un error de sintaxis en qualsevol arxiu de terraform, per exemple, *gcp/devel/europe-west1/gke/variables.tf* i, posteriorment, afegirem els canvis en la branca de git creada anteriorment i pujarem els canvis al repositori remot.

```
$ git checkout -b TFG-Test_CI_Infra
```

```
~/r/tfg-uoc-infraestructura / TFG-Test_CI_Infra !1 git add gcp/devel/europe-west1/gke/var
iables.tf
~/r/tfg-uoc-infraestructura / TFG-Test_CI_Infra +1 git commit -m "Prova error sintaxis"
[TFG-Test_CI_Infra 99577ea] Prova error sintaxis
 1 file changed, 1 insertion(+), 1 deletion(-)
~/r/tfg-uoc-infraestructura / TFG-Test_CI_Infra git push origin TFG-Test_CI_Infra ✔
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 544 bytes | 544.00 KiB/s, done.
Total 7 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
remote:
remote: Create a pull request for 'TFG-Test_CI_Infra' on GitHub by visiting:
remote:      https://github.com/asaldana-uoc/tfg-uoc-infraestructura/pull/new/TFG-Test_CI_Infra
remote:
To github.com:asaldana-uoc/tfg-uoc-infraestructura.git
 * [new branch]      TFG-Test_CI_Infra -> TFG-Test_CI_Infra
```

3. Tot seguit, crearem la "*Pull Request*" en GitHub accedint a l'enllaç que es mostra en la captura anterior i, automàticament, hauríem de veure com els tests no han passat:



4. Efectivament, podem comprovar com el test d'integració continua està marcat en vermell i no permet fusionar els canvis a la branca *main* (els administradors o propietaris del repositori es poden saltar aquesta prohibició tot i que no és aconsellable ni es considera una bona pràctica). Finalment, accedim a la consola de GCP per verificar que l'error que apareix coincideix amb els canvis realitzats:

```
Step #0 - "terraform format": docker.io/hashicorp/terraform:0.15.3
Step #0 - "terraform format": |
Step #0 - "terraform format": | Error: Invalid expression
Step #0 - "terraform format": |
Step #0 - "terraform format": |   on gcp/devel/europe-west1/gke/variables.tf line 4, in variable "allowed_ip_access_control_plane":
Step #0 - "terraform format": |    4:   type =
Step #0 - "terraform format": |    5: }
Step #0 - "terraform format": |
Step #0 - "terraform format": | Expected the start of an expression, but found an invalid expression token.
Step #0 - "terraform format": |
Step #0 - "terraform format":
Finished Step #0 - "terraform format"
ERROR
ERROR: build step 0 "hashicorp/terraform:0.15.3" failed: step exited with non-zero status: 1
```

Fins aquí hem creat els components necessaris per introduir tests de validació en els canvis introduïts en el repositori d'infraestructura com a part de la integració continua. Aquesta mateixa configuració, la realitzarem posteriorment en el cas pràctic quan es creï l'aplicació de proves.

5.3.2. Desplegament continu

Després d'haver completat la fase d'integració continua, passarem a descriure com gestionarem el desplegament automàtic tant de la infraestructura com de les aplicacions. A causa del diferent nivell d'accessibilitat de cada component, s'utilitzaran dos estratègies diferents. D'una banda, per desplegar infraestructura són necessàries credencials amb permisos específics per a poder crear recursos a GCP i, de l'altra, l'aplicació únicament s'executarà en el clúster de Kubernetes. Per les raons anteriors i per raons de seguretat, s'utilitzarà Google Cloud Build (amb *service account* dedicat) per a desplegar infraestructura i ArgoCD per a implementar els processos GitOps d'entrega d'aplicacions ja que aquest últim estarà en el propi clúster de Kubernetes amb privilegis de gestió.

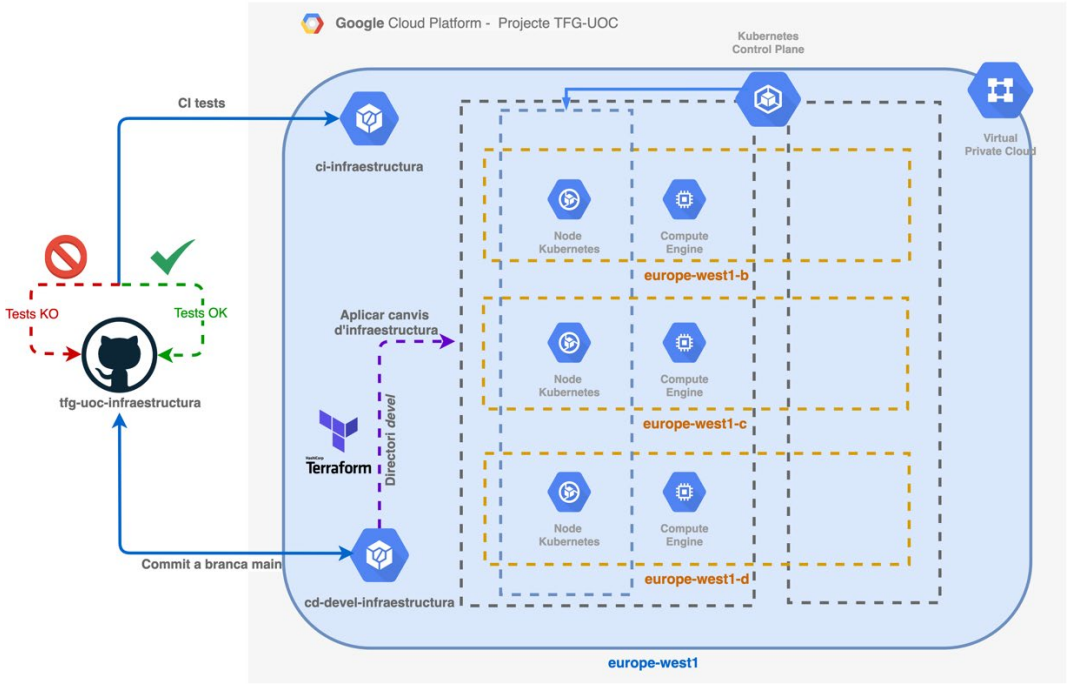
5.3.2.1. Desplegament d'infraestructura

Com s'ha indicat anteriorment, el desplegament d'infraestructura es realitzarà creant un altre *trigger* de Google Cloud Build com ja es va fer per a l'execució dels tests d'integració continua de terraform. Per a poder aïllar els desplegaments en funció dels entorns tenim dos opcions:

- **Branques:** aquesta opció consisteix en utilitzar una branca per a cada entorn i configurar el *trigger* de Google Cloud Build per a que s'executi automàticament cada vegada que detecti un *commit* nou sobre aquesta branca.

- **Directoris**: l'opció dels directoris es basa en disposar de la configuració actual de tots els entorn en la branca principal *main* però tenint un directori dedicat per entorn. La configuració del *trigger* es realitza de tal manera que només s'executi quan es detectin canvis en un directori concret.

Després d'avaluar els avantatges i inconvenients de les alternatives anteriors, s'ha apostat per l'estratègia dels directoris ja que es considera que a llarg termini les branques poden ser més complicades de gestionar perquè poden aparèixer conflictes en el codi. A més a més, utilitzant únicament la branca *main* ens permetrà tenir una fotografia actual de l'estat de tots els entorns sense haver d'anar canviant de branques, cosa que fa perdre el context. Pels motius anteriors, es crearà l'estructura de directoris `gcp/common/cd/infraestructura` en el repositori d'infraestructura i, dins d'aquest, crearem una carpeta per a cada un dels entorns que tinguem en la plataforma. En el cas d'aquest projecte, com s'ha comentat anteriorment, únicament tindrem un entorn (*devel*) per a la infraestructura per reduir les despeses en GCP però crear nous entorns seria tan senzill com replicar el mateix codi que generarem per aquest entorn en un directori amb el nom de l'entorn nou (*stage, pre-production, production, etc...*). El diagrama complet dels processos de CI/CD de la infraestructura queda representat en la següent imatge:



Il·lustració 19 - CI/CD workflow per la infraestructura. Font pròpia.

L'arxiu de configuració de Google Cloud Build, que es troba disponible en aquest enllaç, té únicament dos passos:

- ***Terraform plan*:** aquest primer pas s'encarrega de buscar tots els subdirectoris que contenen arxius de configuració de terraform (extensió *.tf) per a, de forma iterativa, anar executant dins de cada un d'aquests la comanda "*terraform plan*" que únicament mostra els canvis d'infraestructura pendent d'aplicar. És important que aquesta lògica s'executi en primer lloc ja que, si es detecta algun error, atura el procés complet i evita l'aplicació dels canvis d'infraestructura que realitzaria el pas següent.

- ***Terraform apply*:** el següent pas té implementada una lògica idèntica al pas anterior però en comptes de mostrar els canvis pendents a realitzar, els aplica utilitzant la instrucció "*terraform apply*". Un cop acabada aquesta execució, la infraestructura a GCP es trobarà actualitzada amb els canvis definits al repositori.

Les peculiaritats que fan aquest arxiu de configuració de Cloud Build diferent al creat per a l'execució de tests d'integració continua són aquests:

- ***Timeout*:** s'ha incrementat el temps màxim d'execució fins a una hora perquè l'aplicació de canvis d'infraestructura poden tenir una durada llarga en funció dels recursos a canviar o re-generar.

- **Variables d'entorn:** com bé hem dit anteriorment, la idea és que aquest arxiu de configuració pugui ser reutilitzat per futurs entorns a crear així que s'ha decidit eliminar al màxim la definició de valors fixos o constants. Per solucionar-ho, s'ha creat una variable d'entorn anomenada `_ENVIRONMENT` la qual serà passada per la definició del recurs amb terraform així es podrà reaprofitar el mateix codi per qualsevol entorn amb una mínima re-implementació.

Tot seguit, després d'haver explicat quines operacions realitzarà el *trigger* de Cloud Build de desplegament continu d'infraestructura, haurem de crear el recurs `tfg-uoc-cd-devel-infraestructura` a GCP generant el codi de terraform. Ens basarem en l'arxiu de configuració utilitzat pels tests d'integració continua explicat en aquest punt del present document però realitzarem les següents modificacions:

- En comptes de reaccionar davant d'una creació d'una *Pull Request*, aquest *trigger* únicament s'executarà quan es detecti un canvi (*commit* nou) en la branca *main*.

- Es tindrà present que aquest codi pot ser reutilitzat per futurs entorns així que es crearà una variable anomenada *environment*, de tipus local, on s'establirà el nom de l'entorn on s'aplicaran els canvis que servirà perquè el *trigger* únicament s'activi quan es detectin canvis en el directori associat.

- S'assignaran els permisos necessaris d'IAM, replicant els establerts pel compte de servei de terraform sobre el servei de Google Cloud Build per a que pugui realitzar canvis d'infraestructura a GCP

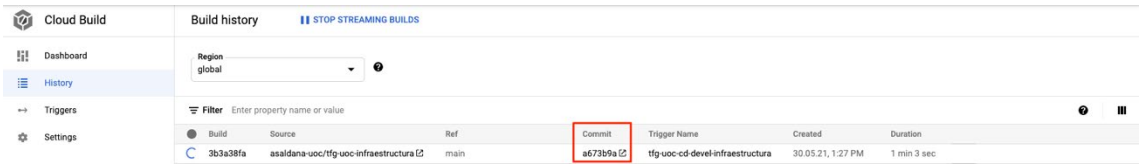
A banda de les justificacions anteriors, es pot consultar el codi complet amb comentaris descriptius de cada recurs en el repositori de GitHub accedint a aquest enllaç. Després d'haver generat la configuració de terraform tan sols queda desplegar el *trigger* a GCP:

```
$ cd gcp/common/cd/infraestructura/devel && terraform apply
```

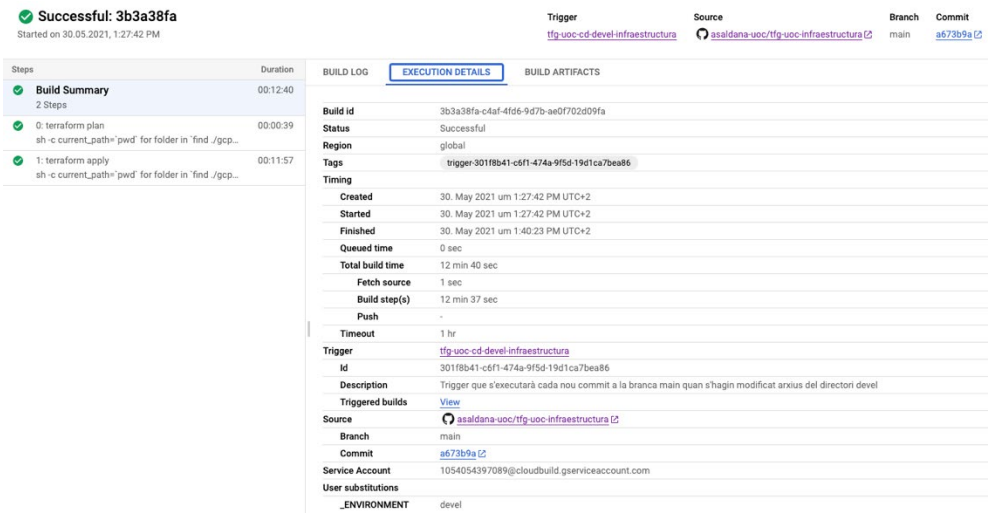
Ara tan sols queda provar que, certament, quan realitzem qualsevol canvi en la infraestructura de l'entorn *devel* i aquests s'afegeixen a la branca *main* generant un nou *commit*, terraform sincronitza automàticament l'estat actual de la plataforma. Per a realitzar-ho, farem un canvi en el clúster de Kubernetes en l'arxiu `gcp/devel/europe-west1/gke/main.tf` i el pujarem a la branca *main*:

```
~/repos/tfg-uoc-infraestructura / main ga .
~/repos/tfg-uoc-infraestructura / main +1 gcmsg "Canvi configuració Kubernetes"
[main a673b9a] Canvi configuració Kubernetes
 1 file changed, 2 insertions(+), 2 deletions(-)
~/repos/tfg-uoc-infraestructura / main ↑1 ggpush
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 583 bytes | 583.00 KiB/s, done.
Total 7 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To github.com:asaldana-uoc/tfg-uoc-infraestructura.git
   ce73323..a673b9a  main -> main
```

Si accedim a la consola de GCP, dins del servei Google Cloud Build, podem comprovar com s'està executant un nou procés generat pel nou git *commit* detectat (a673b9a) que coincideix amb el de la captura anterior:



Després d'uns quants minuts d'execució veurem com el procés ha finalitzat i podem consultar un resum tant de la pròpia execució com de les accions realitzades en la infraestructura:



Per acabar, es comprova que realment els canvis que hem introduït s'han propagat en el clúster de Kubernetes així que donem per validat el desplegament continu de la infraestructura. És important remarcar que hi ha canvis que poden implicar la re-generació total dels recursos, amb la pèrdua d'informació que això implica, així que s'ha de tenir molta cura i coneixement ampli de la repercussió de les modificacions abans d'aprovar una *Pull Request*.

5.3.2.2. Desplegament d'aplicacions

Pel desplegament d'aplicacions utilitzarem una estratègia diferent. El motiu és que les aplicacions residiran en el clúster de Kubernetes i per a gestionar la configuració/versió d'aquestes existeixen eines específiques per dur a terme aquestes tasques com, per exemple, Jenkins X, ArgoCD o Flux. Com ja es va decidir en punts anteriors, apostarem per l'estratègia *"Pull-Based Deployment"* perquè ens evita exposar públicament l'eina que instal·larem a continuació: ArgoCD.

5.3.2.2.1. Instal·lació d'Argo CD

Per realitzar la instal·lació d'Argo CD seguirem el procediment descrit en la pàgina web del producte adaptant-lo als requeriments que necessitem pel projecte:

1. Creem un *namespace* dedicat a Kubernetes per ArgoCD i, posteriorment, apliquem el manifest d'instal·lació proporcionat que crearà els recursos necessaris dins del clúster en el *namespace* argocd creat anteriorment:

```
$ kubectl create namespace argocd
$ kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-
cd/stable/manifests/install.yaml
```

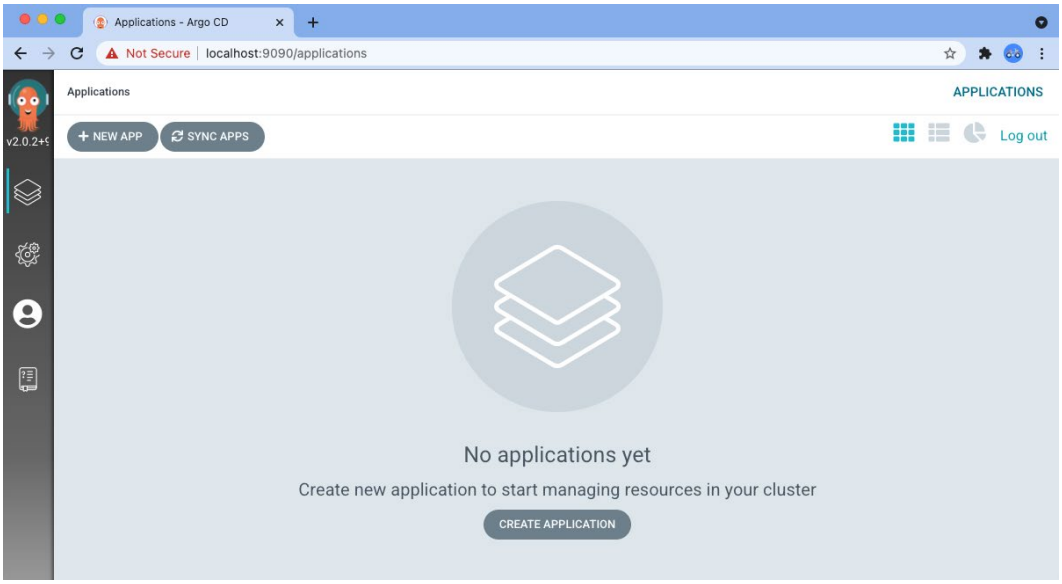
2. Descarregarem el client ArgoCD per a poder gestionar la utilitat des de la línia de comandes:

```
$ VERSION=$(curl --silent "https://api.github.com/repos/argoproj/argo-
cd/releases/latest" | grep '"tag_name"' | sed -E 's/.*"([^"]+)".*/\1/')
$ curl -sSL -o /usr/local/bin/argocd https://github.com/argoproj/argo-
cd/releases/download/$VERSION/argocd-darwin-amd64
$ chmod +x /usr/local/bin/argocd
$ argocd version
```

3. Després de verificar que tenim la versió 2.0.3 del client d'ArgoCD decidim, per temes de seguretat, no exposar l'aplicació públicament així que accedirem utilitzant la comanda *`kubectl port-forward`* de Kubernetes que permet crear un túnel entre un servei intern amb IP privada del clúster amb la màquina local. Tot seguit, canviarem la password per defecte d'administració d'ArgoCD:

```
$ kubectl -n argocd get secret argocd-initial-admin-secret -o
jsonpath="{.data.password}" | base64 -d
$ kubectl port-forward svc/argocd-server -n argocd 9090:443
$ argocd login localhost:9090
$ argocd account update-password
```

4. A continuació, accedirem a la interfície gràfica d'ArgoCD accedint a la URL http://localhost:9090 utilitzant les credencials anteriors. Com veiem, podem veure la pàgina principal de l'aplicació preparada per a afegir noves aplicacions, procés que veurem en el cas pràctic amb l'aplicació de proves que es desenvoluparà.



6. Cas pràctic: desenvolupament i desplegament d'aplicacions

Arribats a aquest punt ja tenim tots els components necessaris i la infraestructura preparada per a desplegar una o moltes aplicacions fent servir metodologies GitOps. En aquest cas pràctic, s'implementarà una aplicació web molt senzilla però que ens permeti comprovar que el procés funciona correctament. Evidentment, es podria haver realitzat les proves amb alguna aplicació de codi obert ja existent però, per interès personal en l'aprenentatge de nous llenguatges de programació, s'ha decidit implementar una aplicació nova per aquest projecte tot i l'esforç addicional que ha suposat.

6.1. Desenvolupament d'una aplicació Golang

El llenguatge utilitzat per a desenvolupar l'aplicació web per a provar la consistència de la plataforma implantada i de les metodologies GitOps és Golang. Aquest llenguatge de programació, conegut també amb el nom de Go, va ser dissenya per Google l'any 2007 per a millorar la productivitat en el desenvolupament i l'eficiència en l'ús dels recursos de computació. Els motius d'elecció d'aquest llenguatge, a banda de l'interès personal d'aprenentatge, també s'ha basat en les característiques i beneficis demostrats:

- És un llenguatge compilat que facilita la portabilitat entre sistemes al poder ser afegit en una imatge d'un contenidor fàcilment. A més a més, garanteix la immutabilitat al no poder-se modificar en el sistema on s'executa sense haver de compilar-se de nou.
- És un llenguatge *tipat*, és a dir, al declarar una variable s'ha de definir el tipus o sinó el procés de compilació falla.
- Té suport de sèrie per *multi-threading* optimitzant l'ús de CPU del sistema.
- És multi-plataforma: una vegada compilat es pot executar en Linux, Windows o MacOS.
- Té un excel·lent gestor de dependències i porta per defecte la utilitat per executar tests unitaris del codi.
- Està adquirint molt protagonisme i popularitat en el sector de desenvolupament de programari on molts projectes estan adoptant aquest llenguatge de programació (terraform i ArgoCD. que s'utilitzen en aquest projecte, en són un exemple)

6.1.1. Entorn de desenvolupament

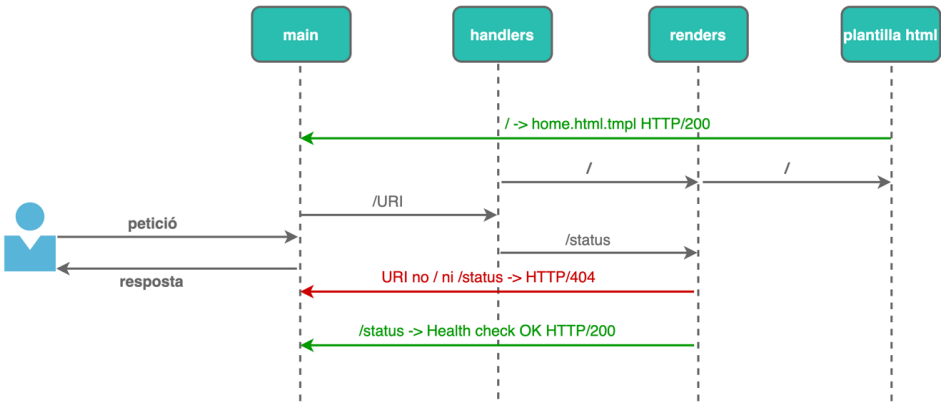
Per a desenvolupar una aplicació amb Go, primerament hem d'instal·lar el binari en el sistema seleccionant la versió que vulguem utilitzar des de la pàgina web https://golang.org/dl/ i tenint en compte el sistema operatiu que utilitzem. La instal·lació consisteix en instal·lar el paquet descarregat, seguir l'assistent i, un cop acabat, podrem utilitzar la comanda ***go version*** per veure que funciona correctament i obtenir la versió instal·lada:

```
~ go version
go version go1.16.3 darwin/amd64
```

Tot seguit, haurem de seleccionar un IDE per a començar el desenvolupament entre les múltiples opcions disponibles: GoLand, VSCode, Atom, etc... Com s'ha explicat anteriorment, en el cas d'aquest projecte s'ha optat per la solució de JetBrains GoLand amb la llicència d'estudiant.

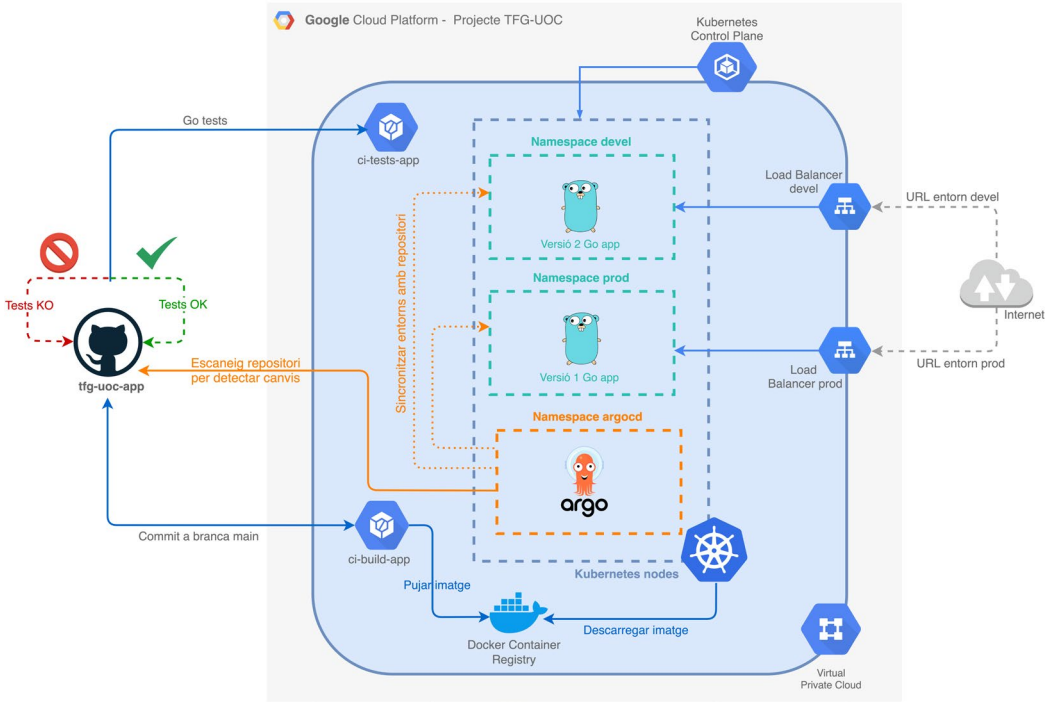
6.1.2. Disseny de l'aplicació i processos de CI/CD

Bàsicament, les funcionalitats que necessitem de l'aplicació per validar que el *workflow* GitOps funciona és que mostri una pàgina principal on aparegui el nom del host on s'està executant, l'adreça IP d'aquest i la versió de l'aplicació en format de *git commit hash*. D'aquesta manera, podrem demostrar d'una manera visual la diferència de versions de codi entre els entorns i, tanmateix, que tenim desplegats diversos *pods* servint l'aplicació per garantir l'alta disponibilitat i la tolerància a fallades. A més a més, s'ha creat un altre *endpoint* que respon a la URL `/status` que s'utilitzarà com a *health check* del *deployment* de Kubernetes per a validar que el *pod* respon correctament i no cal substituir-lo. Així, el diagrama lògic del funcionament de l'aplicació a desenvolupar queda representat de la següent manera:



Il·lustració 20 - Disseny aplicació web Go. Font: pròpia.

A més a més, es crearan dos entorns (*devel* i *prod*) utilitzant els *namespaces* de Kubernetes per tal de simular el màxim possible un cas real d'una aplicació productiva. En el diagrama següent, s'ha representat l'arquitectura de l'aplicació i el *workflow* dissenyat pels processos d'integració i entrega continu:



Il·lustració 21 - Arquitectura aplicació i CI/CD. Font: pròpia.

6.1.3. Implementació de l'aplicació

Per començar amb el desenvolupament s'ha creat un repositori a GitHub anomenat *tfg-uoc-app* seguint el mateix procediment que es va seguir anteriorment per crear el repositori de la infraestructura, creant una estructura de directoris seguint l'estàndard definit per projectes Go en aquest repositori. Per reduir contingut en el present document, no s'adjuntarà la totalitat del codi ja que es troba perfectament documentat i es pot consultar tant accedint al repositori de GitHub com en els arxius adjunts que es proporcionaran amb l'entrega. En canvi, es detallarà la utilitat de l'estructura de directori i les decisions d'implementació més importants:

- **k8s:** en aquest directori tenim els subdirectoris corresponents a cada entorn i, dins d'aquests, es trobaran les definicions dels objectes que es desplegaran a Kubernetes que seran de tipus *Deployment* (desplegament de pods amb els contenidors de la imatge Docker de l'aplicació), *Service* (agrupació lògica dels pods del deployment) i *Ingress (*balancejador de càrrega per exposar l'aplicació a l'exterior).

- **cmd/web:** aquí trobem el paquet principal *main* on l'arxiu *main.go* actua com a punt d'entrada de l'aplicació per a qualsevol petició. Aquí, es troba definit la creació d'un servei web escoltant a través del port 8888 que actuarà com un router, en funció de la URI rebuda, derivant les peticions als *handlers* implementats (*Home* i *Status*)

- **internal/handlers:** aquest paquet implementa la lògica de la resposta a generar en funció de les peticions rebuda. S'han definit dos *handlers*, un per la pàgina principal (*/*) i un altre pel *health check* (*/status*). El primer retorna una plana HTML amb contingut estàtic sobre aquest projecte però també dinàmic com la versió del codi, l'adreça IP, o el nom del host on s'executa el binari. En canvi, l'altre únicament retorna el text "*Health check OK*". En tots dos casos, el codi de resposta HTTP és 200, qualsevol altra petició retorna "*Pàgina no trobada*" amb un HTTP/404.

- **internal/helpers:** aquí s'ha implementat un paquet per funcions auxiliars com, per exemple, obtenir l'adreça IP local.

- **internal/render:** aquest paquet és l'encarregat de renderitzar el contingut de la plantilla HTML a partir de les dades que rep dels *handlers.* Com a peculiaritat, comentar que s'ha utilitzat la llibreria *embed* que permet adjuntar contingut estàtic (HTML en el nostre cas) al binari generat després de la compilació.

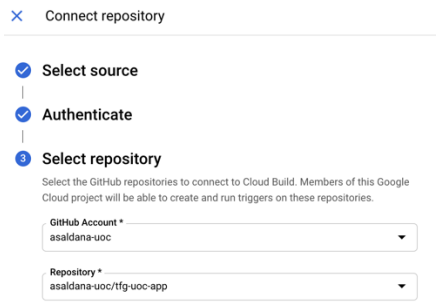
- **web:** aquí trobem definida un tipus de dades que utilitzarem per incloure les variables que s'han de passar a la plantilla HTML.

- **tf:** crearem aquest directori en cas que es requerís desplegar infraestructura pròpia per aquesta aplicació.

Tanmateix, dins de cada directori on hi ha codi Go es troben els tests unitaris de cada arxiu. Els arxius de tests, han de tenir el mateix nom que l'arxiu que conté el codi però amb el sufix *_test*.

6.2. Integració continua de l'aplicació

Després de tenir l'aplicació implementada haurem d'automatitzar els processos de CI/CD d'aquesta. Aplicarem el mateix criteri que hem fet servir per la infraestructura, és a dir, es llençarà automàticament un *trigger* de Google Cloud Build per executar els tests en les "*Pull Requests*" i, un cop els canvis es trobin en la branca *main*, s'activarà un altre *trigger* per a construir la imatge de Docker i publicar-la en el Container Registry. Per poder posar en marxa aquest procés, el primer pas és autoritzar a Google Cloud Build a accedir al repositori on tenim el codi tal com ho vam fer amb la infraestructura:



Després, haurem de protegir la branca *main* repetint la configuració que vam aplicar al repositori d'infraestructura, és a dir, requerint que els tests passin i que hi hagi una aprovació d'una altra persona abans de poder afegir els canvis d'una "*Pull Request*" a la branca principal.

6.2.1. Execució de tests unitaris

En aquesta primera fase, automatitzarem l'execució dels tests unitaris implementats en Go cada vegada que s'obri una "*Pull Request*" en el repositori. Aquest procés estarà gestionat per un *trigger* nou que crearem dins dels servei Google Cloud Build amb el nom `tfg-uoc-ci-tests-app` que es trobarà a la ruta `tf/ci/cloudbuild-tests.yaml` dins del repositori. Les particularitats d'aquest *trigger*, que es pot consultar en la seva totalitat en aquest enllaç, són les següents:

- Utilitza la imatge oficial de golang, versió 1.16, per a coincidir amb la instal·lació local.
- Executa un únic pas on executa la comanda `go tests -v` per verificar els tests unitaris definits en tots els directoris.
- El *trigger* no s'executarà quan s'hagin realitzat canvis en els directoris `k8s` i `tf` ja que no contenen codi Go, per tant, és totalment innecessari.

6.2.2. Construcció de la imatge Docker

La construcció de la imatge de Docker només ens interessa que es realitzi quan tinguem una versió definitiva preparada per ser desplegada en un entorn real. Per aquesta raó, només activarem aquest *trigger* quan es detecti un canvi en la branca *main* la qual cosa voldrà dir que s'han passat les verificacions requerides (tests passats i aprovació d'una altra persona). El procés de construcció de la imatge de Docker i publicació en el Container Registry (repositori d'imatges Docker propi de GCP) la realitzarà un *trigger* amb el nom de `tfg-uoc-ci-build-app`, present en la ruta `tf/ci/cloudbuild-build.yaml`. La definició del *trigger* i del codi de terraform per a desplegar-la en GCP residirà en el mateix directori que el *trigger* per l'execució del tests de Golang, és a dir, en aquest enllaç. A continuació, descriurem breument les característiques més rellevants de la implementació d'aquest procés:

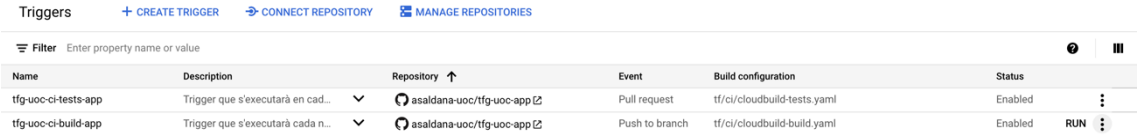
- S'utilitza una imatge proporcionada per GCP, optimitzada per a la construcció d'imatges de Docker, anomenada `gcr.io/cloud-builders/docker`.

- Es construeix la imatge de partir de la definició del fitxer Dockerfile present a l'arrel del repositori utilitzant una estratègia *multi-stage*[35] on la primera capa genera el binari de *go* i, la segona, copia el binari generat anteriorment per a construir la imatge final. La imatge base final és *distroless*[36], és a dir, que únicament contenen el mínim imprescindible per a executar l'aplicació oferint així una alt nivell de seguretat.

- En la construcció de la imatge de Docker, s'injecta el *git commit hash* en el procés de compilació per a que pugui ser mostrat per la pàgina principal de l'aplicació. Aquesta mateixa variable s'utilitza per a etiquetar la imatge a banda del *tag* per defecte *latest*.

- El *trigger* no s'executarà quan s'hagin realitzat canvis en els directoris `k8s` i `tf` ja que no contenen codi Go, per tant, és totalment innecessari.

Desplegarem tant aquest *trigger* com l'anterior generant un únic arxiu de terraform anomenat `main.tf` ubicat en el propi directori i disponible en aquest enllaç. Com s'ha repetit en diverses ocasions, executarem la comanda `terraform apply` per completar la creació dels recursos i posteriorment verificarem que s'han creat correctament en GCP:



6.2.3. Comprovació processos d'integració continua

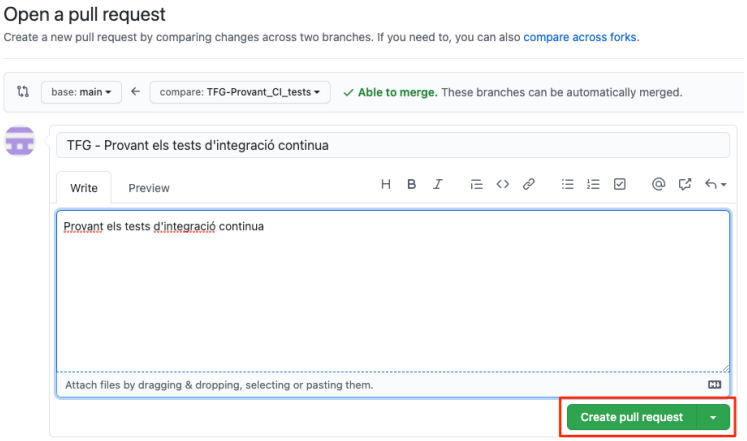
Després d'haver configurat els processos d'integració continua de l'aplicació passarem a provar de l'automatisme implementat. Primerament, realitzarem una modificació en el codi de Golang per a provocar que un test generi un error, d'aquesta manera, al obrir una *Pull Request* ens hauria d'aparèixer

1. Creem una branca nova amb el nom "*TFG-Prova_CI_tests*" i, en aquesta, introduirem l'error en el codi, per exemple, modificant el contingut de la resposta de l'endpoint */status*. Posteriorment, pujarem els canvis al repositori remot.

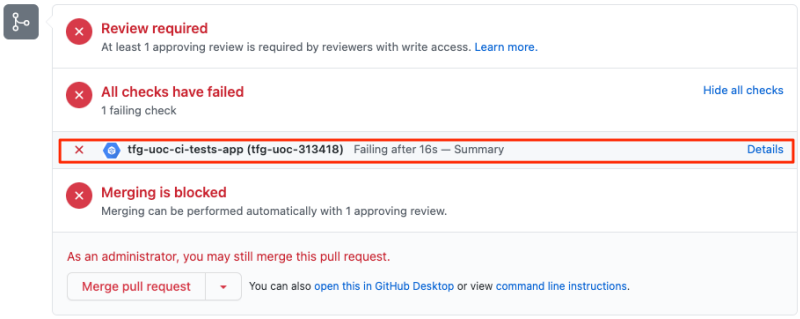
```
$ git checkout -b TFG_Prova_CI_tests
$ git add internal/handlers/handlers.go
$ git commit - "Introducció d'error a propòsit"
$ git push origin TFG_Prova_CI_tests
```

2. A continuació, accedirem a GitHub per obrir la *Pull Request* a partir de la branca pujada en el pas anterior:

[35] *Les construccions multi-statge permeten obtenir imatges finals amb un espai reduït al permetre realitzar instal·lacions de components necessaris utilitzant imatges temporals.*
[36] *Les imatges distroless van ser creades per Google i únicament contenen l'aplicació i les llibreries dependents.*



3. Després d'uns segons d'execució, el *trigger* de Google Cloud Build ens marca els tests com a fallats i impedeix que els canvis es puguin afegir a la branca principal:



4. Consultarem la consola de GCP per a veure en detall els motius de l'errada i, efectivament, coincideix amb els canvis que hem introduït per a provocar que fallin els tests:

```
25 === RUN   TestHandler
26 2021/05/31 10:39:27 Petició rebuda a la URL /
27 2021/05/31 10:39:27 Indentificada l'adreça IP 192.168.10.2 en el host local
28 2021/05/31 10:39:27 Petició rebuda a la URL /status
29     handlers_test.go:41: Resposta obtinguda invàlida, s'esperava Health check KO però s'ha rebut  Health check OK
30 2021/05/31 10:39:27 Petició rebuda a la URL /testing però no es troba disponible
31 --- FAIL: TestHandler (0.00s)
32 FAIL
33 FAIL    github.com/asaldana-uoc/tfg-uoc-app/internal/handlers   0.005s
```

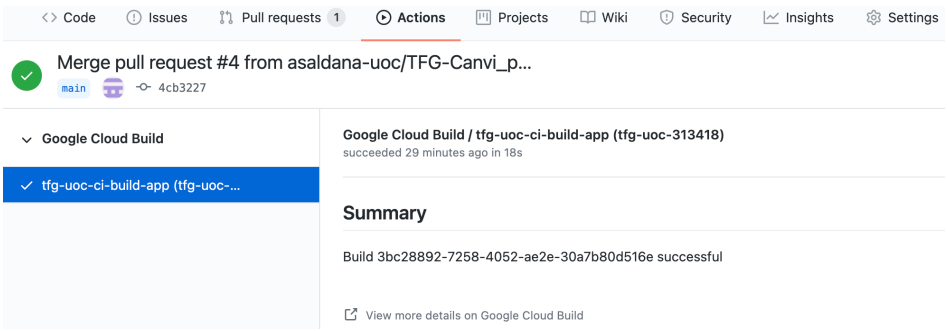
Fins aquí hem provat la primera fase de la integració continua, ara queda provar què passa cada vegada que s'afegeix nou codi en la branca *main.* Realitzarem els passos següents:

1. Introduirem un petit canvi en la plantilla HTML i crearem una *Pull Request* per afegir els canvis en la branca principal.

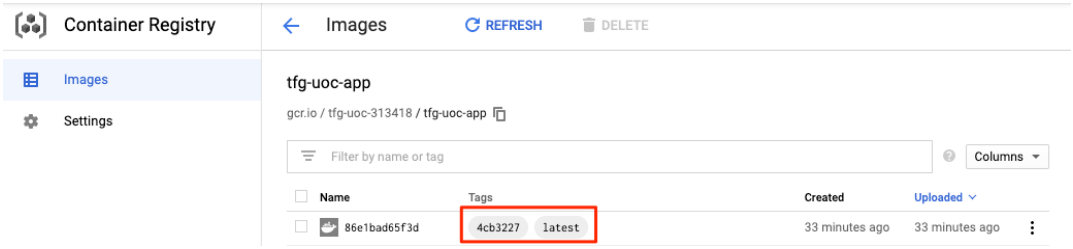
2. Després de comprovar que els tests han passat satisfactòriament fem *merge*[37] d'aquesta *Pull Request* a la branca principal. Cal dir que faria falta l'aprovació d'una altra persona però al ser un treball individual aquest pas s'omet.

[37] *L'operació merge fusiona canvis entre dues branques de git.*

3. Si consultem el *commit* (4cb3227) generat en la branca *main,* veiem con aquest ha produït una execució del *trigger* `tfg-uoc-ci-build-app` amb resultat satisfactori:



4. Tan sols queda accedir al Container Registry per comprovar que s'han generat noves imatges de Docker amb els *tags* que esperem: *4cb3227* i *latest*:



En aquest punt, podem donar per finalitzats i comprovats els processos d'integració continua de l'aplicació que inclouen dues fases: el tests de l'aplicació i la construcció de la imatge Docker.

6.3. Entrega continua de l'aplicació

En aquest apartat tractarem els passos requerits per lliurar l'artefacte generat en el procés d'integració continua als entorns que l'aplicació disposi i, seguint els principis de GitOps, assegurar que en tot moment estigui en execució els objectes de Kubernetes que tenim definits en el repositori de GitHub.

6.3.1. Creació d'entorns

El primer que farem serà crear els entorns que utilitzarà l'aplicació segons el disseny d'arquitectura mostrat anteriorment. Aquests entorns seran realment *namespaces* de Kubernetes que són objectes que permeten establir una capa de separació entre els objectes que es troben en execució en el clúster, principalment possibilitant la duplicació de noms. Aquesta agrupació lògica no impedeix que hi pugui haver connectivitat TCP/IP entre *pods* de diferents *namespaces* ja que aquesta funcionalitat correspon als objectes de tipus *network policy*.

Per a crear els dos *namespaces* generarem un arxiu YAML de Kubernetes al repositori de l'aplicació en la ruta `k8s/common/namespaces.yaml`, al qual s'han afegit comentaris a tall explicatiu

que es pot consultar en aquest enllaç. Després de crear-lo, queda propagar la nova configuració al clúster:

1. Ens situem en del directori on està l'arxiu YAML i apliquem els canvis:

```
$ cd k8s/common/
$ kubectl apply -f namespaces.yaml
```

```
~/repos/tfg-uoc-app/k/common / main kubectl apply -f namespaces.yaml
namespace/devel created
namespace/prod created
```

2. Verifiquem que podem llistar els *namespaces* en el clúster:

```
$ kubectl get ns
```

```
~/repos/tfg-uoc-app/k/common / main kubectl get ns ✔ / tfg-uoc-k8s-cp *
NAME              STATUS   AGE
argocd            Active   22h
default           Active   24h
devel             Active   9s
kube-node-lease   Active   24h
kube-public       Active   24h
kube-system       Active   24h
prod              Active   9s
```

Tot seguit, el que farem serà realitzar unes reserves d'adreces IP públiques per als balancejadors de cada un dels entorns. D'aquesta manera, garantirem que les aplicacions siguin accessibles amb la mateix URL. Aquesta reserva, la crearem amb terraform afegint un arxiu de configuració nou, amb el nom *main.tf*, dins del directori `gcp/devel/europe-west1/apps/static-ips` del repositori d'infraestructura. Un cop implementat, pugem els canvis a GitHub i el procés de CI/CD de la infraestructura aplicarà automàticament les modificacions a realitzar a GCP. Anirem al *log* detallat de l'execució del *trigger* per conèixer les adreces IPs assignades:

```
59 Outputs:
60
61 tfg_uoc_app_devel_ip_address = "35.186.232.147"
62 tfg_uoc_app_prod_ip_address = "34.117.202.192"
```

Per completar aquesta secció, registrarem un domini per a apuntar les adreces IPs de les aplicacions a un FQDN[38] per a facilitar l'accés a les persones. Aprofitant els avantatges que ofereix GitHub als estudiants, registrarem el domini *asaldana.tech* de forma gratuïta en el proveïdor https://get.tech/ i crearem els següents registres DNS:

Entorn	Adreça IP	Tipus de registre	Registre DNS
Devel	35.186.232.147	A	app-devel.asaldana.tech
Prod	34.117.202.192	A	app-prod.asaldana.tech

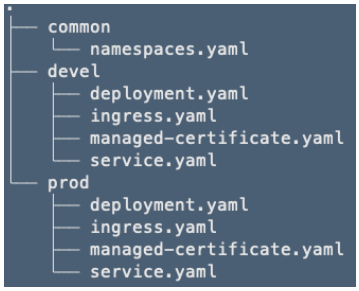
Així, podrem accedir als entorns d'una forma més senzilla accedint a les URLs següents:

- **Entorn devel:** http://app-devel.asaldana.tech
- **Entorn prod:** http://app-prod.asaldana.tech o https://app-prod.asaldana.tech

38 Sigles de "Fully qualified domain name" que correspon al nom complet d'un equip o host.

6.3.2. Creació d'objectes de Kubernetes

Després de preparar els dos entorns haurem d'implementar els objectes de Kubernetes que necessitarà l'aplicació web per al seu funcionament, tenint en compte que els recursos a desplegar seran idèntics a excepció de certs valors propis de cada entorn (nom del *namespace*, versió del codi, adreça IP estàtica i nom del certificat SSL). Utilitzarem el directori k8s per a emmagatzemar els fitxers YAML de Kubernetes realitzant una separació entre entorns, quedant distribuït de la següent manera:



Per reduir el contingut del present document, no es mostrarà la definició dels YAMLs dels objectes de Kubernetes de cada entorn tot i que es poden visualitzar a través de la plataforma GitHub accedint a les URLs següents:

- **Entorn devel:** https://github.com/asaldana-uoc/tfg-uoc-app/tree/main/k8s/devel
- **Entorn prod:** https://github.com/asaldana-uoc/tfg-uoc-app/tree/main/k8s/prod

Únicament, passarem a explicar els tipus d'objectes creats i aquelles configuracions que es consideren més rellevants:

- **Deployment:** són objectes que gestionen els *Pods* i les *ReplicaSets* amb la característica que permet controlar i orquestrar el desplegament de noves versions. En aquest cas pràctic, establirem que l'aplicació tingui almenys 3 pods (directiva *replica*), que utilitzi la imatge de Docker `tfg-uoc-app` que es troba en el Container Registry amb adreça `gcr.io/tfg-uoc-313418/tfg-uoc-app` i que l'aplicació escolta en el port 8888. Finalment, s'han afegit també un *health check* utilitzant la directiva `livenessProbe` que s'executa cada 10 segons i verifica l'estat de l'aplicació accedint a la URI implementada per aquesta finalitat.

- **Ingress:** aquest objecte s'encarrega d'exposar l'aplicació a Internet aprovisionant un recurs a Cloud Load Balancing a GCP. S'especifica quina reserva d'adreça IP ha d'utilitzar utilitzant l'anotació `kubernetes.io/ingress.global-static-ip-name` i també l'ús d'un certificat SSL autogestionat per GCP utilitzant la parametrització `networking.gke.io/managed-certificates`.

- **Service:** el *Service* ens permetran agrupar els pods que coincideixin amb el selector *app* que indiquem, el qual, coincidirà amb l'establert en l'objecte *Deployment*. El tipus seleccionat serà *NodePort* que mapejarà el port 8888 en tots els nodes per a què el balancejador pugui distribuir el tràfic sobre aquests correctament.

- **ManagedCertificate**: és un objecte exclusiu de GCP que permet crear un certificat SSL autogestionat per la pròpia plataforma. Únicament, s'ha d'establir un nom pel certificat i assignar sobre quins noms de domini aquest tindrà validesa. **Per limitacions de quotes en GCP, únicament tindrà certificat SSL actiu a l'entorn de *prod*.**

Un cop finalitzada la implementació dels objectes de Kubernetes tan sols queda desplegar-los utilitzant l'eina que hem instal·lat per aquestes funcions: ArgoCD.

6.3.3. Afegir l'aplicació en ArgoCD

En aquesta secció afegirem els dos entorns de l'aplicació en l'eina ArgoCD per a que pugui gestionar les aplicacions i sincronitzi l'estat de la plataforma amb la configuració definida en el repositori de GitHub. Els passos que seguirem són els següents:

1. Ens autentiquem en ArgoCD amb el client de línia de comandes:

```
$ argocd login URL_ARGOCD
```

```
~/repos/tfg-argo argocd login                                    INT ✘
WARNING: server certificate had error: x509: cannot validate certificate for  because
it doesn't contain any IP SANs. Proceed insecurely (y/n)? y
Username:
Password:
' ' logged in successfully
Context '  ' updated
```

2. Veiem com ens apareix un missatge d'invalidesa del certificat. Aquesta situació és normal perquè el certificat és auto-signat i, per tant, no ha sigut emès per cap entitat certificadora (CA). Tot seguit, crearem l'aplicació per a l'entorn de desenvolupament indicant la ruta dins del repositori de codi que ha de consultar constantment ArgoCD per detectar diferències (*k8s/devel*):

```
$ argocd app create tfg-uoc-app-devel --repo https://github.com/asaldana-uoc/tfg-uoc-
app --path k8s/devel --dest-server https://kubernetes.default.svc --dest-namespace
devel
```

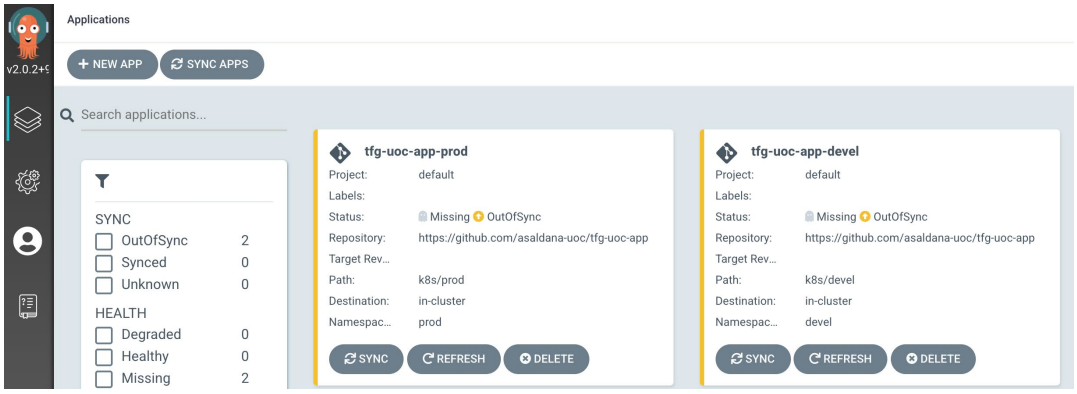
```
~/repos argocd app create tfg-uoc-app-devel --repo https://github.com/asaldana-uoc/tfg-uoc-ap
p --path k8s/devel --dest-server https://kubernetes.default.svc --dest-namespace devel
application 'tfg-uoc-app-devel' created
~/repos
```

3. Repetim la mateixa operació però en aquest cas per l'entorn de producció així que haurem de modificar el nom de l'aplicació a crear i el directori on ArgoCD haurà d'escanejar:

```
$ argocd app create tfg-uoc-app-prod --repo https://github.com/asaldana-uoc/tfg-uoc-
app --path k8s/prod --dest-server https://kubernetes.default.svc --dest-namespace
prod
```

```
~/repos argocd app create tfg-uoc-app-prod --repo https://github.com/asaldana-uoc/tfg-uoc-app
 --path k8s/prod --dest-server https://kubernetes.default.svc --dest-namespace prod
application 'tfg-uoc-app-prod' created
~/repos
```

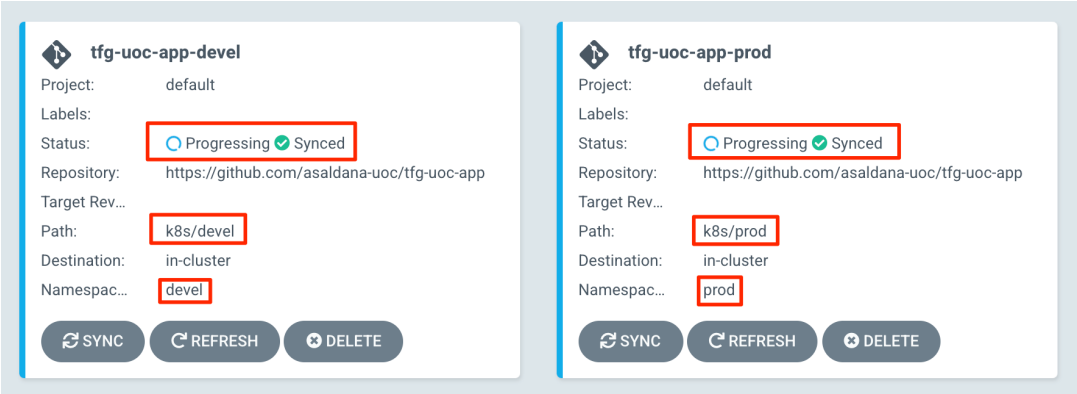
4. Accedirem a la interfície gràfica d'ArgoCD per a veure l'estat de les aplicacions creades en els passos anteriors:



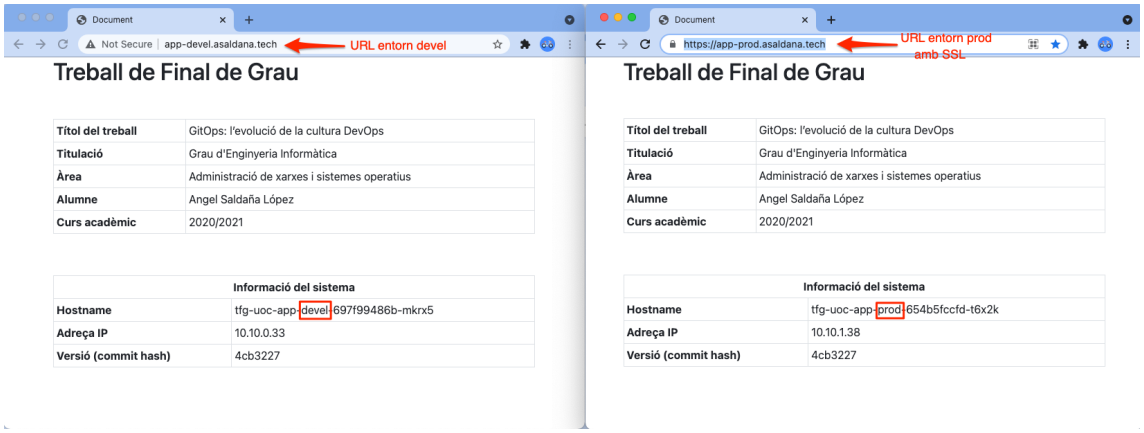
5. El que comprovem és que tot i que les aplicacions estan ben configurades encara no s'ha desplegat cap objecte al clúster de Kubernetes. Això és perquè, per defecte, ArgoCD al crear una nova aplicació no activa la sincronització automàtica per precaució. En el cas pràctic, necessitem que davant qualsevol canvi en el repositori de GitHub, s'actualitzi la configuració a Kubernetes així que activem l'auto-sincronització en tots dos entorns. A més a més també forçarem l'eliminació dels objectes que no s'utilitzin amb el paràmetre *--auto-prune* i, també establirem la directiva *--self-heal* que reverteix els canvis en el clúster davant d'intervencions manuals:

```
$ argocd app set tfg-uoc-app-devel --sync-policy automated --auto-prune --self-heal
$ argocd app set tfg-uoc-app-prod --sync-policy automated --auto-prune --self-heal
```

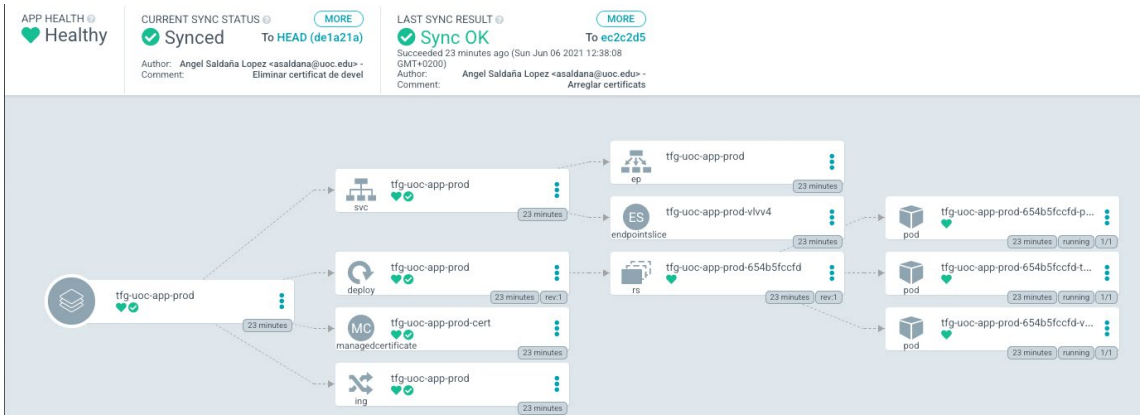
6. Tot just després d'executar les comandes anteriors tornem a la interfície gràfica i veiem com les aplicacions s'estan sincronitzant automàticament amb la configuració que ha trobat al repositori. Aquesta acció crearà tota l'arquitectura de l'aplicació desplegant els objectes de Kubernetes definits a GitHub:



7. Després d'uns quants minuts, perquè s'han de crear recursos a GCP i porta el seu temps, accedirem a les URLs de l'entorn de *devel* i de *prod* per a comprovar que les aplicacions funcionen correctament:



Després de tenir les aplicacions funcionant, investigarem la interfície gràfica d'ArgoCD on trobem informació molt interessant sobretot els objectes que formen part de la plataforma com logs, esdeveniments recents, estat dels objectes, estat de la sincronització respecte el repositori GitHub, etc... Tanmateix, l'eina ArgoCD genera un diagrama de l'arquitectura de l'aplicació amb tots els objectes de Kubernetes que formen part, tal com es pot visualitzar en la captura següent:



Si s'accedeix múltiples vegades a cada una de les URLs anteriors es podrà veure com el balancejador aplica l'algorisme round-robin i cada petició nova cau en un *pod* diferent, com es pot veure comparant les files "*Hostname*" i "*Adreça IP*"

6.4. Prova automàtica de sincronització

En aquest apartat final simularem com es desplegaria automàticament una nova versió de codi sobre un entorn. Per a visualitzar-ho, s'ha modificat la plantilla HTML del repositori de codi per afegir una nova fila per indicar el semestre del curs, creant una *Pull Request* amb els canvis proposats. Després de passar els test unitaris, farem *merge* de la *Pull Request* anterior generant un nou commit a git i, automàticament, es generarà una imatge de Docker amb el *commit* anterior (9d22c9f) gràcies a l'automatització de la integració continua que hem implementat per

aquest projecte. Ara, tan sols quedaria modificar l'arxiu YAML del *deployment* de l'entorn de devel per establir la nova imatge de Docker que s'ha de desplegar.

1. Creem una nova branca al repositori tfg-uoc-app i afegim el canvi de versió.

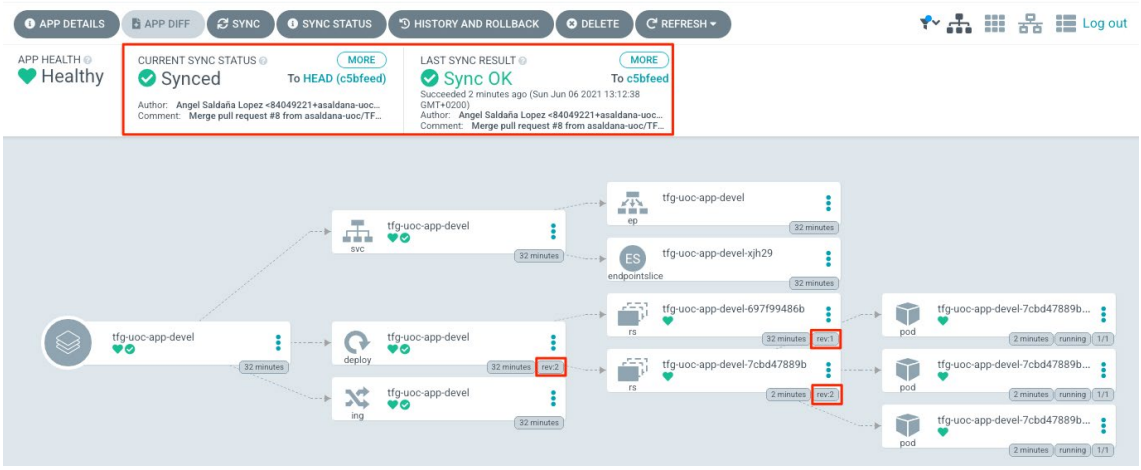
```
$ git checkout -b TFG-Actualitzar_versio_devel
```

2. Modifiquem en l'arxiu YAML del *deployment* de devel, la línia on es declara la imatge de docker a utilitzar, i indiquem `gcr.io/tfg-uoc-313418/tfg-uoc-app:9d22c9f` que coincideix amb la que s'ha generat després de l'últim *commit*. Pugem els canvis a GitHub i creem una *Pull Request*.

```
$ git add k8s/devel/deployment.yaml
$ git commit -m "Actualitzar versió a l'entorn de devel"
$ git push origin TFG-Actualitzar_versio_devel
```

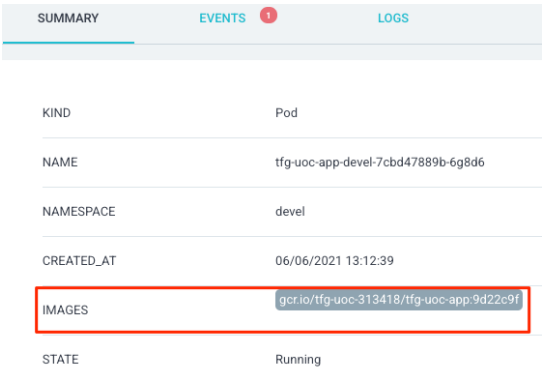
3. En aquest punt, hauríem d'esperar que una altra persona aprovi el canvi però, com és un treball individual, farem *merge* dels canvis a la branca *main*.

4. En uns instants, ArgoCD detectarà la diferència que hi ha entre el clúster de Kubernetes i el repositori de GitHub i automàticament desplegarà nous *pods* perque utilitzin la imatge de Docker nova.

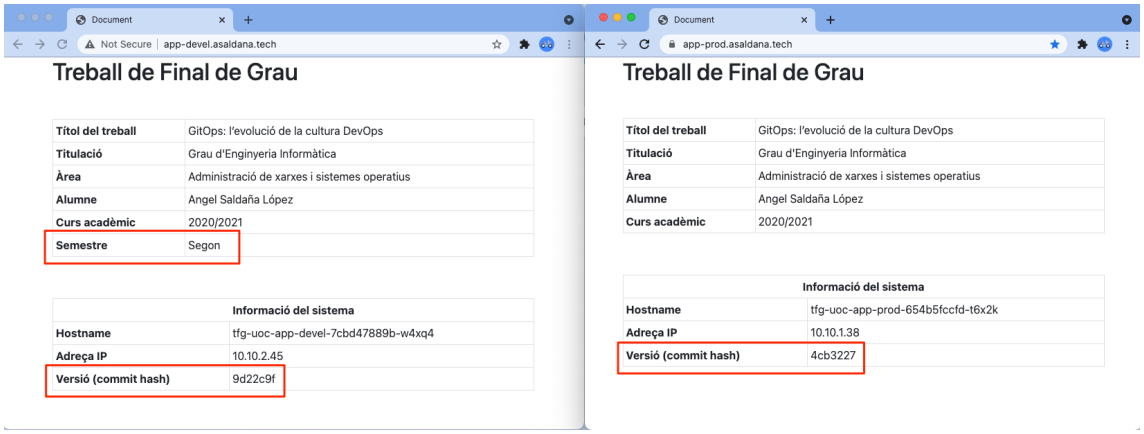


Com mostra la imatge anterior, aquesta acció ha creat una nova revisió o versió de l'objecte Deployment i ha desplegat pods nous a partir d'aquesta configuració. Aquesta dada és important perquè es podria fer un rollback fàcilment tornant a la revisó anterior, la número 1. A més a més, disposem de molta informació tant en la part superior sobre la informació del repositori de GitHub (autor i el missatge de l'últim *commit*) com si seleccionem els objectes individualment.

5. Per comprovar la versió de la imatge de Docker en ús, seleccionem un pod qualsevol i mirem els seus detalls. Com es pot veure, coincideix amb la imatge definida en el repositori.



6. Finalment, obrim en el navegador les URLs de les aplicacions dels entorn i veiem les diferències que esperàvem: versió diferent de codi i una nova fila en la versió de *devel* que no apareix en la versió de *prod*.



En el cas invers funcionaria de la mateix manera, és a dir, si alguna persona realitzés un canvi manual en el clúster de Kubernetes, automàticament, ArgoCD tornaria aplicar la configuració definida en el repositori de GitHub per a tenir l'entorn desitjat.

7. Millores recomanades

Després d'haver acabat amb la implementació la plataforma que fan possible la implantació de metodologies GitOps, arriba el moment de fer balanç d'aquelles àrees que hauria desitjat aprofundir però que no s'ha pogut destinar més temps a causa de la durada delimitada que té el Treball de Final de Grau i la quantitat de noves tecnologies que s'han hagut d'aprendre pel desenvolupament d'aquest projecte.

Primerament, després de tenir la plataforma preparada m'hauria agradat executar tests de rendiment per a posar en pràctica les propietats d'escalabilitat, resiliència i tolerància a fallades dels sistemes que s'han desplegat. A més a més, i relacionat amb l'àrea de la disponibilitat de les aplicacions informàtiques, hagués sigut interessant aplicar tècniques de *Chaos Engineering* utilitzant les eines *Chaos Monkey* o *Simian Army*. Al marge de les proves esmentades anteriorment, en un escenari amb una aplicació productiva, mancaria afegir a la proposta plantejada en aquest TFG tests automatitzats d'integració i d'acceptació de l'aplicació dins dels processos CI/CD definits. Tanmateix, ha mancat implementar una estratègia de versionat del codi utilitzant alguns dels estàndards definits com, per exemple, *semantic versioning*[39].

En segon lloc, trobo que faltaria instal·lar eines de monitoratge i d'agregació de logs sobre tot si es tracta d'un cas real per a un entorn productiu. Sense aquestes utilitats és molt complicat tenir una visió de l'estat global de la plataforma i identificar problemes. A banda de la pròpia monitorització de la plataforma també seria aconsellable implementar un sistema de notificacions amb la finalitat que els processos de CI/CD notifiquin als equips tècnics per correu electrònic o a través d'aplicacions de missatgeria com *Slack* o *Teams*.

L'altre element que m'hauria agradat aprofundir és realitzar el desplegament d'aplicacions amb dades persistents, és a dir, que es connectin a base de dades o altres serveis que requereixen emmagatzemar informació important. Tanmateix, en el cas pràctic implementat, tots els components de l'aplicació residien en Kubernetes, és a dir, eren *stateless* però manca per provar com es podria integrar en el desplegament d'aplicacions utilitzant metodologies GitOps la creació de recursos en la plataforma cloud.

La seguretat és un altra àrea interessant que manca aprofundir i que en un projecte real s'ha de tenir present. D'una banda, tenim la gestió de la seguretat IAM de la plataforma cloud i, de l'altra les RBAC (*Role-Based Access Control*) que implementa Kubernetes. Respecte la primera cal dir que s'ha sigut especialment curós ja que un error de seguretat en GCP podria suposar una desviació inesperada en la facturació però, en canvi, per Kubernetes no s'ha establert cap política d'accés interna (RBAC) ni polítiques de seguretat de xarxa (*network policies*).

Finalment, la gestió automàtica de la zona DNS no s'ha implementat i únicament s'han creat manualment 2 registres tipus A. Caldria investigar amb més detall com es podria administrar els registres DNS simultàniament amb el desplegament de l'aplicació.

[39] *Estàndard de versionat d'aplicacions que segueix el següent patró: MAJOR.MINOR.PATCH.* https://semver.org/

8. Conclusions

Durant el desenvolupament d'aquest treball final he pogut comprovar en primera persona els avantatges que ofereix, a les organitzacions que es dediquen al desenvolupament de programari, l'aplicació de cultures que promouen l'automatització dels processos com ho fa DevOps o el cas d'estudi del present document: GitOps.

Principalment, aquestes metodologies aconsegueixen que els cicles de desenvolupament siguin més curts i es pugui entregar valor a les persones usuàries d'un producte amb major freqüència ja que es redueix dràsticament el temps de lliurament de noves funcionalitats.

Segons el meu punt de vista, no està indicat per a organitzacions molt petites o que el seu negoci principal no sigui el sector digital atesa la complexitat de l'arquitectura a preparar. Per a posar en funcionament tots els automatismes ha calgut aprendre moltes tecnologies noves que evolucionen molt ràpidament i, per aquesta raó, fa falta personal tècnic molt especialitzat que pugui evolucionar i tenir actualitzada la plataforma.

Això si, un cop es troben totes les peces integrades, la facilitat i la despreocupació de la gestió dels sistemes fa que millori notòriament l'eficiència dels processos que participen en els cicles de desenvolupament. Aquesta situació, repercuteix positivament en la productivitat dels equips tècnics i fa possible que es pugui invertir el temps estalviat en tasques que aportin més valor al negoci. A més a més, hem comprovat amb quina facilitat es podria crear un entorn nou o afegir una aplicació nova a la plataforma implementada.

Tanmateix, hem pogut verificar la quantitat d'eines disponibles per realitzar tasques similars. Aquest punt pot ser positiu i negatiu alhora perquè, d'una banda, ofereix una flexibilitat molt àmplia a l'hora de seleccionar tecnologies però, de l'altra, pot suposar un problemes en algunes organitzacions perquè poden haver-hi decisions arbitràries en l'elecció que pot generar conflictes entre l'equip tècnic.

Des de la vessant més personal, aquest treball final ha superat les expectatives que tenia marcades inicialment, tant en la dedicació com en l'aprenentatge adquirit. Per a poder executar el projecte he hagut de realitzar moltes lectures, entendre conceptes nous, aprendre noves eines, explorar nous llenguatges de programació, integrar tots els components i superar els problemes tècnics que m'he trobat. Pels motius anteriors, considero que he hagut de realitzar un esforç personal important que ha requerit molta dedicació, potser superior a la que esperava inicialment. Això sí, penso que l'esforç s'ha vist recompensat perquè he après infinitat de tecnologies noves que poden ser importants durant la meva trajectòria professional.

9. Annex

Aquesta secció agruparà enllaços importants sobre el projecte.

Repositoris de codi implementat a GitHub

- https://github.com/asaldana-uoc/tfg-uoc-app
- https://github.com/asaldana-uoc/tfg-uoc-infraestructura

URLs de les aplicacions

- http://app-devel.asaldana.tech/
- https://app-prod.asaldana.tech/

Documentació oficial de les tecnologies utilitzades

- **ArgoCD:** https://argoproj.github.io/argo-cd/
- **Docker:** https://docs.docker.com/
- **Git:** https://git-scm.com/doc
- **GitHub:** https://docs.github.com/
- **Golang:** https://golang.org/doc/
- **Google Cloud:** https://cloud.google.com/docs
- **Kubernetes:** https://kubernetes.io/docs/home/
- **Terraform:** https://www.terraform.io/docs/

10. Bibliografia i referències

[1] DevOps: Breaking de development-operations barrier [data de consulta: 28/02/2021]. https://www.atlassian.com/devops/what-is-devops

[2] Cloud Computing: Everything you need to know [data de consulta: 01/03/2021]. https://www.salesforce.com/products/platform/best-practices/cloud-computing/

[3] Microservices: what they are and why use them [data de consulta: 02/03/2021]. https://www.leanix.net/en/blog/a-brief-history-of-microservices

[4] SOLID principles [data de consulta: 02/03/2021]. https://howtodoinjava.com/best-practices/solid-principles/

[5] Guide to GitOps [data de consulta: 10/03/2021]. https://www.weave.works/technologies/gitops/

[6] What is GitOps? Why is it important? How you can get started? [data de consulta: 10/03/2021]. https://www.youtube.com/watch?v=JtZfnrwOOAw

[7] Tutorial: Everything You Need To Become a GitOps Ninja [data de consulta: 12/03/2021]. https://www.youtube.com/watch?v=r50tRQjisxw

[8] Automate Kubernetes with GitOps [data de consulta: 15/03/2021]. https://www.weave.works/blog/automate-kubernetes-with-gitops

[9] Morris, Kief. *Infrastrure as Code, 2n Edition*. O'Reilly Media, 2020. ISBN 9781098114671.

[10] Sayfan, Gigi. *Mastering Kubernetes*. Packt Publishing, 2020. ISBN 9781839211256.

[11] Usecase: GitOps. [data de consulta: 23/03/2021]. https://about.gitlab.com/handbook/marketing/strategic-marketing/usecase-gtm/gitops/

[12] Become a git guru. [data de consulta: 25/03/2021]. https://www.atlassian.com/git/tutorials

[13] Pro Git. [data de consulta: 26/03/2021]. https://git-scm.com/book/en/v2

[14] What is Continuous Integration? [data de consulta: 01/04/2021]. https://www.atlassian.com/continuous-delivery/continuous-integration

[15] Continuous Delivery. [data de consulta: 01/04/2021]. https://martinfowler.com/bliki/ContinuousDelivery.html

[16] What is a Container? [data de consulta: 03/04/2021]. https://www.docker.com/resources/what-container

[16] What is a Container Orchestration? [data de consulta: 04/04/2021]. https://newrelic.com/blog/best-practices/container-orchestration-explained

[16] Kubernetes Documentation [data de consulta: 04/04/2021]. https://kubernetes.io/docs/home/

[17] GitOps [data de consulta: 05/04/2021]. https://www.gitops.tech/

[18] GitHub [data de consulta: 06/04/2021]. https://github.com/

[19] ArgoCD [data de consulta: 06/04/2021]. https://argoproj.github.io/argo-cd/

[20] Primeros pasos con Google Cloud [data de consulta: 06/04/2021]. https://cloud.google.com/docs

[20] Terraform documentation [data de consulta: 07/04/2021]. https://www.terraform.io/docs/index.html

[21] Cloud-Native Patterns [data de consulta: 03/05/2021]. https://www.youtube.com/watch?v=fZEFWh4xBEw

[22] TFenv [data de consulta: 08/05/2021]. https://github.com/tfutils/tfenv

[24] Sawler, Trevor. *Building Modern Web Applications with Go (Golang).* Udemy, 2021. https://www.udemy.com/course/building-modern-web-applications-with-go/

[23] Testing your HTTP Handlers in Go [data de consulta: 28/05/2021]. https://blog.questionable.services/article/testing-http-handlers-go/