TecnoCampus

Centres universitaris adscrits a la

upf. Universitat Pompeu Fabra Barcelona

Grau en Enginyeria Informàtica de gestió i sistemes d'informació

Desenvolupament d'una *pipeline* de CI/CD per a una aplicació de C++

Memòria final

Arnau Reig Méndez
Tutor: Dr. Enric Sesa Nogueras
2023-2024

TecnoCampus
Mataró-Maresme

Taula de continguts

Índex d'imatges

Glossari

***Build*:** Aquest terme té dues connotacions: El procés de *build* compren totes aquelles accions necessàries que transformen el codi font en un executable llest per l'usuari final (Anomenat realitzar una *build*). També pot rebre el nom de *build* l'executable resultant.

CI: *Continuous Integration*. Automatitzar la integració de canvis en el codi. Es comprova que es pot fer una build i que els tests funcionin correctament.

CDE: *Continuous Delivery*. Automatitzar l'entrega dels canvis i deixar-los llestos per a poder ser enviats a producció. En altres paraules, es posen els canvis que s'han comprovat gracies al CI en un ambient controlat molt similar al de producció en busca de possibles errors.

CD: *Continuous Deployment*. Automatitzar la integració dels canvis dins dels ambients de producció. S'agafen els canvis ja controlats en la fase de CDE i s'integren a producció.

CMake: És un dels programes més estàndards a l'hora d'automatitzar el procés de *building* de codi amb C o C++. És gratuït i *open source*.

DevOps: Conjunt de praxis i filosofies que augmenten l'habilitat de l'organització per entregar codi de forma més eficient.

Discord: Programa de comunicació online via veu, text i vídeo.

Epic: En el camp de gestió i producció de projectes, una tasca rep el nom de "epic" quan representa una fita important dins del producte.

Open Source: Codi lliure on es permet la seva modificació i ús pels seus usuaris.

Pipeline: Cadena de processos ordenats on la sortida de cada element és l'entrada del següent.

Production: "El codi de producció" és tot programa que s'està executant de cara al públic i que és sensible a canvis.

Repositori: Lloc on guardar el codi font d'una aplicació, generalment online.

1 Objecte del projecte

El procés de creació de software és tradicionalment complicat. Per aquest motiu, les eines de gestió de tasques o de disseny d'arquitectures tenen un rol essencial en la prevenció de possibles problemes durant el desenvolupament. Tot i així, de la mateixa forma que la tecnologia avança, els processos també necessiten fer-ho. Els equips son més grans, els usuaris demanen més contingut i els sistemes han d'estar disponibles amb els mínims retards possibles. Per fer front a aquesta evolució la indústria ha creat una sèrie de pràctiques (*DevOps)* amb la intenció d'automatitzar processos interns i millorar la relació entre les diverses fases del desenvolupament.

La recerca principal del treball passa per investigar aquestes pràctiques de software i aprendre com aplicar-les dins d'un projecte. Això pot consistir en accions tan bàsiques com implementar un sistema de *testing* automàtic que garanteixi un bon estat del codi font en tot moment. D'altres més avançades passen per millorar la qualitat de vida dels programadors mantenint les dependències al dia i assegurant que l'ambient de treball és reproduïble. L'objectiu final és saber quines de les pràctiques fan falta realment per a que ajudin a entregar millor el software desitjat.

També és important destacar que s'ha acabat escollint C++ com a llenguatge per a realitzar la investigació ja que diferents estudis previs conclouen que la creació d'automatitzacions en aquest llenguatge presenten un grau de dificultat afegit a causa de la gran heterogeneïtat en el seu entorn. Trobar una forma de combatre aquests problemes és una de les altres motivacions del treball.

Finalment, ha fet falta trobar un programa ja desenvolupat per a poder aplicar els resultats del treball sense preocupar-se per la possible implementació de l'aplicació, només dels seus processos interns. Per aquest motiu s'ha elegit *DiscLand*, un bot escrit amb C++ i realitzat amb anterioritat que afegeix elements de rol i de gestió de recursos a Discord i que necessita una base de dades per a poder funcionar.

En resum, l'objecte d'aquest projecte és l'estudi i implementació d'una pipeline d'automatització per a projectes escrits en C++. El treball resultant te la missió de

desmitificar la feina de DevOps i aportar nou coneixement en el desenvolupament d'aquests processos específics del llenguatge de programació.

2 Estudi previ

2.1 Contextualització

El punt de partida d'aquest TFG es basa en unes dades recolzades per diferents investigacions les quals remarquen que poques llibreries construïdes amb C++ (principalment *open source*) donen importància a una *pipeline* de desplegament moderna. Un primer estudi [1] remarca que un 69% dels projectes no disposaven d'un arxiu *CMake* per a realitzar la *build* del codi. A part molts presentaven problemes a l'hora de compilar a causa d'un error en les seves dependències. Un altre estudi [2] constata que un 30% dels projectes amb C++ utilitza una pipeline de CI/CD, un percentatge petit en comparació a altres llenguatges més moderns.

La causa d'aquestes xifres la podem trobar en altres investigacions que identifiquen una discordança entre la forma en que es va concebre el C++ i les noves praxis dels processos de software. En l'article de *Shayan Shajarian* [3] es menciona que aquest llenguatge treballa en un nivell d'abstracció relativament baix, resultant en un codi poc fiable i que requereix d'un gran nombre de tests per assegurar-ne la seva qualitat. A més, el fet que les aplicacions s'hagin de compilar per a diferents arquitectures minva la capacitat de replicació. La gestió de dependències també es veu afectada per aquesta limitació, donat que l'entorn en que es treballa no es heterogeni i existeixen múltiples programes que intenten resoldre el mateix problema [4].

Un punt en comú de moltes d'aquestes recerques és que treballen a partir d'un producte ja en funcionament. Donat que s'està millorant processos interns té sentit utilitzar programari ja desenvolupat que serveixi com a base.

Per a realitzar la part pràctica de les investigacions s'ha escollit DiscLand, una aplicació originada a partir d'un treball previ [5] i que està programada completament en C++. D'aquesta forma es poden obviar els detalls del funcionament intern i es pot posar el focus en la implementació de les millores plantejades. Tot i així, és important dedicar una explicació breu sobre el seu funcionament i quines necessitats tecnològiques presenta.

2.1.1 DiscLand

DiscLand és un programa que afegeix mecàniques de rol i de gestió de recursos a Discord. A partir de les dades dels jugadors guardades en una base de dades, l'aplicació renderitza una imatge amb la informació pertinent.

Imatge 1: Exemple d'una imatge generada per DiscLand. En aquest cas l'usuari pot veure l'estat del seu inventari.

El principal fil d'execució passa per connectar-se a l'API de Discord i escoltar esdeveniments de comandes que els jugadors generen. Quan es detecta una comanda nova, aquesta és enviada a l'aplicació per a que construeixi una resposta de forma automàtica. Per exemple, quan un jugador escriu al xat la comanda *"/inventory",* l'aplicació llegeix l'esdeveniment i genera les dades necessàries per a mostrar en pantalla una imatge similar a la Imatge 1, però específica per a cada usuari.

La base de dades està implementada a partir d'un clúster local de MongoDB. Això significa que l'executable de l'aplicació i la instància de la base de dades estan en el mateix dispositiu. Si una comanda necessita fer ús de la informació d'un usuari, l'aplicació obre una connexió única a la instància i converteix la informació subministrada per la base de dades en objectes compatibles de C++. El procediment és a la inversa en cas que faci falta guardar informació nova.

En el seu estat actual, l'aplicació no compta amb un sistema de testing avançat que s'encarregui de vetllar per un funcionament correcte de les diferents funcionalitats del joc. A part, el sistema d'actualització de la *build* és manual. Formarà part del procés de desenvolupament fer una llista exhaustiva d'aquestes mancances i entendre com poder resoldre-les.

2.2 Antecedents

Tot i que els estudis descriuen una certa dificultat a l'hora de generar ambients d'integració per aplicacions en C++, val la pena mencionar alguns projectes molt coneguts que utilitzen aquests sistemes per assegurar mínimament la qualitat del codi.

2.2.1 SFML

SFML és una llibreria que proveeix d'una interfície de programació per als components de l'ordinador i així facilitar el desenvolupament d'aplicacions multimèdia [6]. Al ser multi plataforma, els seus desenvolupadors han de ser molt conscients d'escriure un codi funcional per a cada una de les diferents arquitectures. El sistema de finestres i recursos gràfics a Windows és molt diferent al de Linux, per exemple.

En la Imatge 2 es pot veure com s'assegura una mínima qualitat per a cada plataforma que es dona suport. És important detallar que no només es prova per a cada arquitectura sinó que a més es realitza un test per a cada compilador i per a cada configuració. En total son 78 tests que necessiten ser completats per donar el *commit* com a vàlid.

Imatge 2: Diversos pipelines dins del projecte SFML

Aquest és un exemple bastant clar de la heterogeneïtat inherent de C++ on inclús les seves eines més bàsiques (com les que realitzen tasques de compilació) necessiten tenir assegurada la seva compatibilitat.

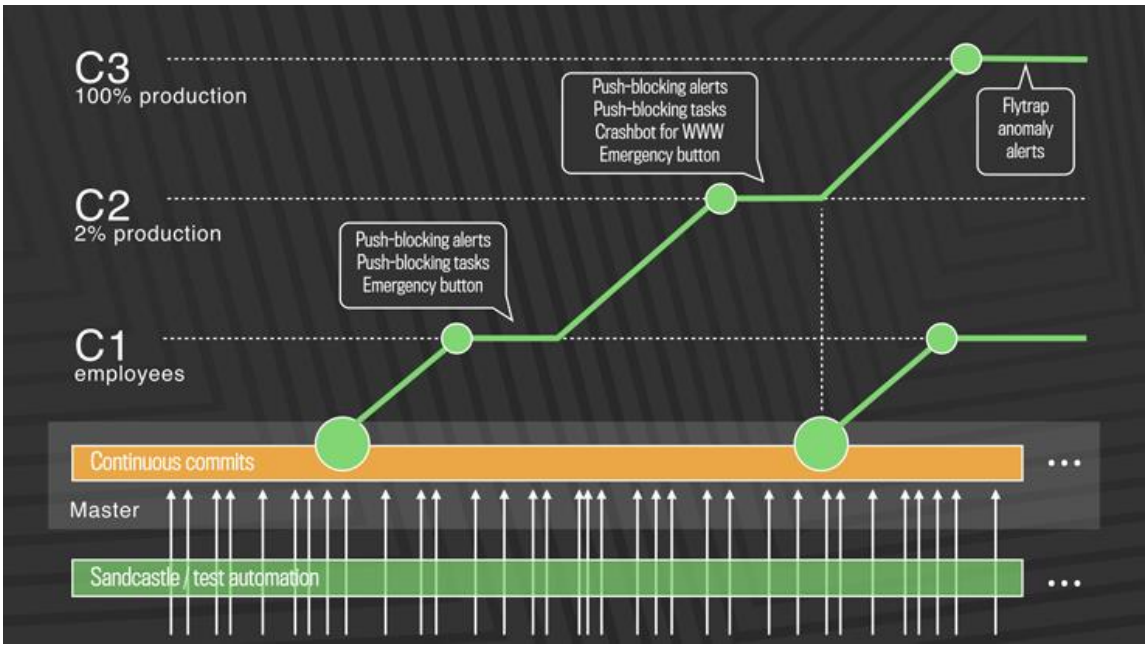
Dins de cada un d'aquests tests, es pot trobar la descripció de com s'executen i quin és el seu objectiu (veure Imatge 4). Generalment, la primera fase es basa en preparar l'ambient de treball i descarregar totes les dependències necessàries per a que, en la segona fase, es pugui configurar i compilar el codi desitjat. Un cop el programa està preparat, ja està tot llest per executar els tests necessaris.

2.2.2 Facebook

Mirant un exemple a gran escala es pot veure quines son les parts que es prioritzen més a causa del seu alt risc a l'hora d'afectar els usuaris finals. En un article publicat per *Meta* [7] es veu clarament com la fase d'adjuntar els canvis dels programadors a l'aplicació final (pujar-ho a producció) és una de les més perilloses.

Segons el text, la necessitat d'absorbir una quantitat elevada de canvis de tots els desenvolupadors contrasta amb el requeriment de mantenir una aplicació funcional en tot moment. Per aquest motiu, els encarregats de millorar aquest procés van idear un sistema de *Continous Deployment* on els canvis son publicats poc a poc al públic general, començant per grups de testing interns o amb uns usuaris molt limitats. Si es detecta un error crític, es pot tornar endarrere mitjançant un *rollback*.

En la Imatge 3 es pot veure un exemple gràfic d'aquesta filosofia dibuixada per *Meta*. Cada fletxa blanca al peu de la imatge representa un canvi d'un desenvolupador en un temps determinat i que ha estat integrat dins la branca "*master*". Per poder realitzar aquesta acció, el codi ha d'haver passat abans per una sèrie de tests de forma satisfactòria que comproven que no s'ha malmès cap funcionalitat. Cada cert temps, l'estat actual de la branca "*master*" es publica a un grup molt reduït de persones. D'aquesta forma es permet provar les noves funcionalitats sense afectar a tot el gruix d'usuaris que utilitzen la plataforma. Si cap error és detectat, de forma orgànica l'actualització s'allibera cada cop a més persones fins arribar a la totalitat dels usuaris o el "*100% production*".



Imatge 3: Funcionament d'un desplegament de codi a Facebook

2.2.3 DPP

Discord PlusPlus és una llibreria en C++ que ajuda als programadors a crear bots i automatitzacions dins la plataforma de comunicació Discord sense haver de preocupar-se pels protocols interns. En la pàgina oficial [8] es pot trobar informació sobre com utilitzar el codi a partir de documentació amb exemples i explicacions sobre les diverses implementacions.

De la mateixa forma que la pipeline estudiada amb SFML, DPP utilitza les *Github Actions* per comprovar el codi pujat al repositori en els diversos ambients als que donen suport. Aquesta última part és important: només s'ha de validar el funcionament en aquells ambients que realment l'usuari final necessiti, encara que impliqui haver d'utilitzar diferents compiladors per al mateix sistema operatiu. En el cas dels creadors de DPP, utilitzen tot aquest procediment per a verificar que el codi compila satisfactòriament en les diverses arquitectures a les que donen suport.

Imatge 4: Fases internes d'un test per a Linux AMD de la llibreria DPP

L'exemple que es veu en la Imatge 4 és una prova específica dins de l'ambient de Linux. Cada sistema operatiu té una forma de funcionar molt diferent en quant a la instal·lació de binaris o a la realització de processos d'automatització. Per exemple, en el cas de Linux, moltes dependències es poden obtenir mitjançant el seu repositori de binaris públic a través la comanda "*apt-get*". En els sistemes Windows aquesta funcionalitat no està present, pel que molts desenvolupadors han d'adaptar-se instal·lant un gestor de dependències externes com Chocolatey o un compilador de C++ no oficial com MinGW.

2.2.4 LibGD

LibGD és una llibreria molt petita d'edició d'imatges en temps real que serveix com a base en l'aplicació *Discland* per a renderitzar el joc per a cada usuari. Ja que és bastant més antiga que els casos explicats anteriorment, serveix com a exemple ideal per veure altres metodologies de *pipelines* de CI/CD. En la Imatge 5 es pot veure la gran diversitat d'arxius que es necessita per a realitzar aquests processos per als diferents sistemes operatius. Per a donar servei a Windows s'utilitza MSYS, una col·lecció d'eines específiques per poder desenvolupar C++ natiu en el sistema. Per a Linux cal executar el bootsrap.sh, un Shell script pensat pels sistemes UNIX. A part, es pot fer servir *CMake* si ja es tenen les dependències necessàries. Està clar que la

flexibilitat del llenguatge i de les necessitats dels usuaris juga un rol clau en aquesta llibreria.

Imatge 5: Diferents arxius dedicats a fer la build i la comprovació del codi.

2.3 Interpretació i reflexions

A partir de la informació recopilada, sembla evident que no hi ha una regla general d'implementació de sistemes de CI/CD dins de C++. En altres llenguatges com en el cas de Java es gaudeix de certa homogeneïtat gràcies als seus Package Managers (*Maven* i *Gradle*) i a que tenen una capa d'abstracció del hardware, alliberant-los de ser afectats per l'ambient d'execució. Encara que pot ser interessant comparar com gestionar un ambient de treball en un llenguatge més tancat on aquestes eines ja estan integrades, aquest estudi es troba fora de l'abast del treball.

D'altra banda, part de l'èxit del C++ es deu a aquesta llibertat per a crear des de zero els teus propis *pipelines*, podent adaptar a la perfecció els requeriments de l'aplicació de la millor forma possible. Si ja des d'un inici el hardware i els drivers presenten diferencies en l'arquitectura, un llenguatge que necessita treballar a nivell baix s'ha d'adaptar a totes aquestes peculiaritats.

Imatge 6: Una mostra de diferents *build Systems* dins de C++. D'esquerra a dreta: Xmake, CMake, Premake i Meson

3 Objectius i abast

Principals:

- **Documentar l'estat de l'art** dels pipelines de CI/CD en els desenvolupaments basats en C++.
- **Dissenyar i desenvolupar** una pipeline de CI/CD on un programador pugui realitzar un canvi en el codi i s'actualitzi de forma automàtica en el núvol.
- **Assegurar una qualitat mínima del codi** en tot moment mitjançant testing i anàlisi estàtica.

Secundaris:

- **Monitoritzar** l'estat del codi i generar informes on es pugui saber la causa dels errors i especular possibles millores.
- **Millorar la qualitat de vida** dels desenvolupadors configurant l'ambient de programació o generant documentació de forma automàtica.

4 Metodologia

Per a realitzar la millora dels processos de desplegament de l'aplicació definits en els objectius s'utilitzarà la metodologia *scrum* [9]. D'aquesta forma es podrà comprovar de forma empírica i ordenada els avenços del desenvolupament detectant possibles errors durant les fases més prematures. Donat que s'està experimentant amb tecnologies noves, es creu adient tenir aquesta seguretat per així reduir els riscs.

El *scrum* es basa en dividir el desenvolupament en *sprints* per aportar valor al producte de forma consistent [9]. La primera fase o "*sprint planning*" consisteix en plantejar el l'abast del *sprint* i escollir una sèrie de tasques que els desenvolupadors hauran de realitzar. Durant la fase de desenvolupament, les persones encarregades de cada tasca intenten acabar-les dins del temps límit. Cada persona és responsable de la gestió del seu temps. Finalment, en la fase de *review* es valora la feina feta i l'estat de les tasques. El procediment es repeteix fins que el producte es considera acabat.

En el cas d'aquest treball, els *sprints* tindran una longitud de dues setmanes i les seves tasques seran realitzades per un únic desenvolupador començant la setmana de l'1 de Gener de 2024. Tenint en compte que l'entrega del producte es vol completar el 20 de Maig del mateix any, el número total de sprints disponibles son 10.

Tot i que la metodologia plantejada està pensada per equips ja que obliga a mantenir una mínima comunicació entre els diversos agents que participen en el desenvolupament, en aquest projecte s'aplicarà de forma unipersonal. Amb això es pretén evitar possibles prejudicis o tendències errònies, doncs cada dues setmanes l'estat de la feina es posa sota revisió.

4.1 Fites del projecte

El disseny a alt nivell d'un projecte [10] consisteix en definir les fites més importants que conformen el producte acabat per així estimar el temps i altres recursos necessaris. Aquestes tasques més generals reben el nom de "epics". Un cop acabat aquest primer plantejament, la feina de desgranar i quantificar les tasques més específiques es simplifica substancialment.

En aquest projecte, les principals èpiques son les següents:

- **Disseny de l'arquitectura i de les comunicacions**: Primer s'ha de definir quines parts faran falta i entendre els requeriments tècnics que comporten. Entendre els problemes als que es vol donar solució i dissenyar els sistemes necessaris.
- **Sistema gestor de versions de llibreries i dependències:** S'ha vist que és important en C++ tenir cura de les dependències portant un control de la seva versió i compatibilitat. Per aquest motiu es busca automatitzar el seu procés de *build* utilitzant eines d'integració continua, facilitant-ne la gestió.
- **Comprovació automàtica de codi**: Quins tests fan falta i com activar-los de forma automàtica per acabar creant-los dins l'aplicació.
- **Automatització de tasques entre Dockers**: Algunes de les tasques es realitzen en containers separats. Cal crear els productors i els consumidors dels events prèviament dissenyats per a que tot funcioni de forma automàtica.

5 Definició de requeriments funcionals i tecnològics

L'objectiu de les llistes presentades a continuació es identificar els mínims requeriments de la pipeline de CI/CD que es vol desenvolupar en tots els seus aspectes.

5.1 Requeriments funcionals:

- Analitzar el codi entregat per un desenvolupador i decidir si compleix amb els estàndards definits.
- Pujar una actualització del software al núvol de forma automàtica sense que repercuteixi de forma greu a l'usuari final.
- Mantenir les dependències del codi al dia i assegurar que s'està fent servir la versió adient per a minimitzar errors.
- Mantenir l'ambient de treball reproduïble.
- Obertura d'un canal web per on es pot visualitzar l'estat del procés i el seu resultat.
- Dotar a l'aplicació la possibilitat de reportar errors i altres mètriques importants pels desenvolupadors.
- Realitzar el procés de *build* de forma automàtica i generar els tests adients.

5.2 Requeriments no funcionals:

- Assegurar temps ràpids de compilació i de comprovació de tests.
- Redacció d'errors entenedora que ajudi al seguiment de possibles inconvenients.
- Vetllar per a una mínima seguretat en l'aplicació al núvol: Tenir el mínim codi per evitar filtracions i limitar l'accés a la base de dades.
- Procurar que la metodologia plantejada sigui fàcil d'utilitzar i ajudi al desenvolupador a entregar codi de millor qualitat invertint menys temps.

5.3 Requeriments tecnològics:

- Servidor al núvol capaç d'executar Docker amb els ports d'entrada i sortida ben configurats.

- Ordinador amb un Ubuntu 22.04 per a desenvolupar l'aplicació i actualitzar el seu contingut.
- Discord.
- Navegador web.
- Connexió a Internet estable.

6 Marc teòric i tecnològic

6.1 Introducció al C++

La principal propòsit del treball és realitzar les investigacions del desplegament de software sota els mecanismes i estructures del C++. Durant els apartats anteriors s'han estudiat els números que donen suport a l'afirmació que aquest llenguatge, malgrat la seva flexibilitat, no sol ser gaire compatible en aquest camp. Així doncs, l'objectiu de la secció actual és exposar les característiques tecnològiques més importants del llenguatge per entendre la font d'aquestes incompatibilitats.

6.1.1 Principis fonamentals

C++ és un llenguatge compilat orientat a objectes que dona un gran control sobre la memòria. El terme compilat significa que per a poder executar un programa el codi font ha de ser tractat i preparat per un hardware específic [11]. Al contrari de Java, que va ser creat amb la missió de "*Escriu-lo una vegada, executa'l on vulguis*", C++ no és considerat com un llenguatge portable i sovint cal tenir present les diferents arquitectures. De fet, això és una conseqüència que parteix inherentment del seu disseny. *Bjarne Stroustrup* en el seu llibre "*The Design and Evolution of C++"* [12] defineix en trets generals els objectius principals del C++:

- "*No ha d'haver cap llenguatge per sota de C++"* a excepció de l'ensamblador. En altres paraules, ha de compilar directament al codi màquina del xip en qüestió.
- "*El que no utilitzes no ho pagues*", també anomenat com el principi del "*zero-overhead*".

Aquests principis es van dissenyar per a poder crear un llenguatge robust a l'hora de programar sistemes i algorismes mantenint una eficiència molt més elevada respecte els altres.

Finalment, a l'hora de programar en C++ cal tenir en compte la diferència essencial entre una declaració i una definició [13]:

6.1.1.1 Declaració:

Significa establir les característiques i els noms de les variables i les funcions, però no es reserva cap espai en la memòria.

6.1.1.2 Definició:

Per cada declaració ha d'haver-hi una sola definició. En aquest punt s'estableix el cos i el contingut de la declaració.

6.1.2 Procés de compilació

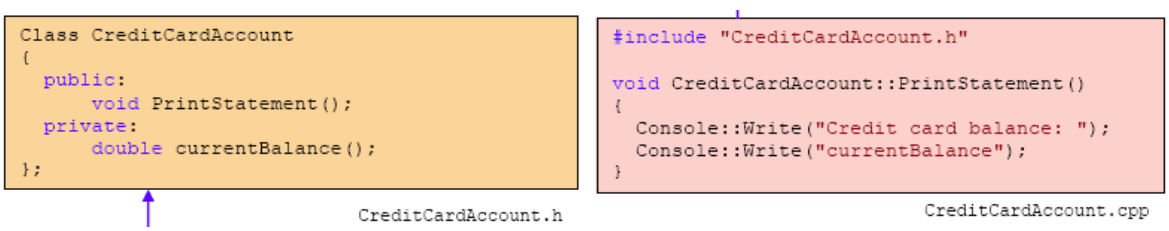
Per a poder executar un programa en C++ primer cal passar-lo per un procés de compilació. Això significa passar de tenir un arxiu escrit en un llenguatge que una persona pot entendre a una sèrie de instruccions ordenades en codi màquina per a la CPU [11].

El compilador és el programa encarregat de realitzar la conversió a codi màquina. Dins de C++ hi ha diferents opcions a escollir, però sovint aquesta va vinculada als requeriments de l'arquitectura (MSVC per a Windows, o gcc/g++ per a Linux).

6.1.2.1 Estructura dels arxius

Actualment hi ha moltes extensions que serveixen per reconèixer arxius que contenen codi escrit en C++. Per simplificar, aquest treball utilitzarà l'extensió ".cpp".

Una de les normes principals que afecta l'estructura de tot programa en C++ defineix que cada arxiu ".cpp" és tractat independentment [14]. Com a conseqüència, tota la informació sobre les diferents declaracions en fitxers externs ha de ser present en cada arxiu. Per evitar redundàncies, la comunitat del llenguatge va crear els header files o ".h" (també acceptada el ".hpp"). D'aquesta forma les declaracions poden ser compartides en molts arxius ".cpp" sense duplicar-ne la seva definició.



Imatge 7: Exemple on es veu com es declara la classe CreditCardAccount (arxiu .h) i després és defineix la seva lògica (arxiu .cpp).

Un cop clarificats els arxius necessaris per un programa en C++, ara es pot passar a parlar sobre el procés de compilació passant per cada una de les seves fases:

6.1.2.2 Preprocessament

En aquesta fase es preparen els diferents arxius del programa per a la seva compilació a través de la resolució i expansió de macros. En C++ una macro és una comanda que, per cada cop que es trobi definida en el codi, el preprocessador la substitueix per la seva definició.

És important destacar aquesta fase pel fet que és aquí on es resolen les directrius "#include". El seu resultat és la substitució del contingut complet de l'arxiu que s'està fent referència. En l'exemple de la Imatge 8, el fitxer "iostream" serà copiat per complet en el codi obtenint així totes les declaracions necessàries per a poder utilitzar la funció "cout" i així imprimir en la terminal.

```
#include <iostream>

#define THIS_IS_A_MACRO 2

using namespace std;

int main()
{
    if (THIS_IS_A_MACRO) {
        std::cout << "La variable és igual a "+std::to_string(THIS_IS_A_MACRO) << std::endl;
    } else {
        std::cout << "Això no s'imprimirà" << std::endl;
    }
}
```

Imatge 8: Exemple d'ús d'una macro en C++. "THIS_IS_A_MACRO" expandeix a 2. Com a resultat per terminal es veurà "La variable és igual a 2". Elaboració pròpia.

Al finalitzar el preprocessament cada arxiu .cpp queda enllestit per a la seva compilació. El resultat de totes aquestes operacions és una unitat de traducció [11].

6.1.2.3 Compilació

En aquesta fase és on s'agafa el programa i es converteix en codi màquina utilitzant les normes del propi C++. Cada unitat de traducció és transformada en un arxiu binari de tipus .obj que conté les instruccions i altra informació necessària per a la següent

etapa. Els arxius .obj no es poden executar ja que no tenen cap adreça real assignada, només una sèrie de referències i definicions encara no resoltes.

Gràcies a que cada arxiu és compilat de forma separada, no cal realitzar tot el procés de nou si es decideix canviar el codi en un arxiu. Només cal tornar a compilar l'arxiu afectat.

6.1.2.4 Enllaçament

Finalment, en l'enllaçament és on s'uneixen tots els diferents arxius .obj de la compilació en un sol executable. Això significa trobar i unir les declaracions realitzades amb les seves definicions. A part, també s'assignen adreces de memòria i dades de forma absoluta. És per aquest motiu que els executables no son del tot reubicables a altres dispositius.

6.1.3 Eines de compilació

En aquest punt hauria d'estar clar que per compilar codi en C++ cal tenir un coneixement profund de les comandes del compilador, les seves fases i què impliquen. Aquesta dificultat és la causa del desenvolupament de les anomenades "build tools" com CMake, Premake o Conda.

CMake és l'actual estàndard en la compilació de projectes en C++ [15]. A partir de sentencies bàsiques es pot automatitzar la inclusió de binaris o de header files (.h). L'objectiu principal de CMake no és la creació del producte final com a tal, sinó de la configuració del sistema capaç de fer-ho. En un Linux, un desenvolupador pot configurar CMake per a que exporti un sistema basat en *Makefiles*. En canvi, a Windows, es pot configurar per a un projecte de Visual Studio.

```
cmake_minimum_required(VERSION 3.22)

project(dcl_db VERSION 0.0.1 DESCRIPTION "Database Handler library for DiscLand")
include_directories("include/")
add_library(${PROJECT_NAME} SHARED "src/db_write.cpp" "src/db_query_operations.cpp")
```

Imatge 9: Exemple d'un arxiu CMakeList.txt per a configurar la compilació de la llibreria encarregada de comunicar-se amb la base de dades. Elaboració pròpia.

Finalment, també permet configurar de forma ràpida quin tipus de programa es vol exportar. Principalment n'hi ha de dos tipus:

- Un executable és un codi que té un punt d'inici (generalment una funció main) i que conforma un procés un cop obert. En altres paraules, utilitza processament de la CPU per obtenir un resultat.
- Una llibreria és un conjunt de funcions que simplifiquen un problema, com ara renderitzar formes simples en una pantalla o donar el resultat a una funció matemàtica a partir d'uns valors concrets. La llibreria no té un punt d'inici ni un fil d'execució.

6.1.4 Gestió de dependències

Sovint per a satisfer requeriments en un programa cal enfrontar-se a problemes que altres desenvolupadors han resolt. Quan un codi depèn d'una solució externa per a poder funcionar (generalment una llibreria), aquesta rep el nom de dependència [16]. En funció del seu tipus, la forma amb la que s'ha de satisfer varia:

- Una llibreria estàtica ha de ser compilada i enllaçada en el moment de compilació del codi. En Linux aquests fitxers tenen l'extensió ".a"
- Una llibreria dinàmica o compartida permet l'enllaçament en temps d'execució, però això també significa que l'arxiu ha de ser present dins de l'ordinador. En Linux aquests fitxer tenen l'extensió ".so"

Tenir present la versió i el lloc on son les dependències simplifica el procés de satisfacció i és essencial per a la replicació que necessita el testing de software. Els gestors de paquets presents en sistemes Unix com el "apt-get" en Ubuntu o "pacman" en "Arch Linux" automatitzen aquesta gestió, però a vegades és necessari haver de compilar de forma manual les llibreries externes.

A mesura que un programa va acumulant funcionalitats i dependències, el procés de construcció es va complicant. A més, si múltiples desenvolupadors han de tenir accés al codi i han de pujar canvis amb freqüència, és fàcil arribar en un punt on sigui vital organitzar processos de comprovació de forma automàtica. És en aquest moment on la integració i el desplegament de software entren en joc.

6.2 La integració de Software

No és difícil trobar en un mateix sistema de software diferents funcionalitats separades que han de treballar de forma interrelacionada. Una aplicació web per a demanar menjar des de casa ha de unir el sistema que dona suport la interfície amb el sistema que es comunica amb la base de dades. A més a més, en equips on hi participen diversos desenvolupadors la sincronització amb els canvis ha de ser meticulosa per evitar conflictes.

Així doncs, la integració de software és el procés en el que els diferents agents que conformen una aplicació son acoblats [17]. L'objectiu final de la integració del software no és augmentar la qualitat de l'aplicació o la seva eficiència directament sinó més aviat millorar tots aquells processos que l'acompanyen. D'aquesta forma, els desenvolupadors tenen una visió clara de l'estat actual del projecte, reduint riscs de forma substancial. No obstant, aquests beneficis han de ser recolzats per unes praxis constants en el dia a dia del desenvolupament i que tot l'equip ha de lluitar per mantenir.

6.2.1 Introducció

Per començar, primer cal enumerar els sistemes més importants en qualsevol pràctica d'integració continua: Els controladors de versions i el testing de funcionalitats [18].

6.2.1.1 Controladors de versions

Els controladors de versions permeten principalment mantenir a la vegada diferents versions dels mateixos fitxers i tenir un historial complet de la seva evolució. A part, també ajuda en la distribució de codi millorant així la capacitat de col·laboració entre desenvolupadors [18].

El sistema actualment més utilitzat i sobre el que es basa la part pràctica del treball és Git. Aquest software permet guardar les diferents versions d'un arxiu en forma de *commits*: paquets de canvis signats per un desenvolupador. Posteriorment aquests canvis poden ser pujats al núvol per a que tot l'equip pugui tenir accés a ells. A part, quan és requereix una unió entre diferents *commits*, el propi sistema intern de Git actualitza els fitxers de forma automàtica. Si hi ha conflictes el programa avisa que cal una resolució manual.

6.2.1.2 Testing

Per assegurar la qualitat en el codi cal una sèrie de processos que s'encarreguin de comprovar-ho. Normalment aquesta comprovació passa per verificar les funcionalitats principals de l'aplicació, però la tipologia dels tests és molt diversa i s'han d'adaptar segons el seu objectiu final [18].

Per una banda, el "Unit Testing" s'utilitza quan es vol realitzar una comprovació d'una funcionalitat en un ambient totalment separat al de l'aplicació. D'aquesta forma és pot verificar que una funció està realitzant el seu objectiu de forma satisfactòria. Ja que no necessiten cap component extraordinari per funcionar, son molt ràpids a l'hora d'executar-se.

El resultat d'un test és binari: O bé la prova ha estat passada o s'ha trobat un error. La determinació d'aquest resultat be donada per dos casuístiques:

- Un error d'execució com ara una violació de segment
- Una excepció en temps d'execució que no ha estat tractat per una clàusula try catch
- A partir de les afirmacions (assertions): Igualtats en el codi que determinen la funcionalitat que s'està comprovant. Per exemple, la validació d'una suma entre dos números és pot realitzar amb la següent afirmació: *REQUIRE(2+2==4)*.

```
-------------------------------------------------------------------------------
/home/shrishail/workspace/cpp/catch2_demo (copy)/tests/test_tdd_fizzbuzz.cpp:20
...............................................................................

/home/shrishail/workspace/cpp/catch2_demo (copy)/tests/test_tdd_fizzbuzz.cpp:21: FAILED:
  REQUIRE( fizzBuzz(3) == "Fizz" )
with expansion:
  "3" == "Fizz"

===============================================================================
test cases: 2 | 1 passed | 1 failed
assertions: 2 | 1 passed | 1 failed
```

Imatge 10: Exemple d'un resultat d'un test unitari amb la llibreria Catch2 amb C++, extret del blog "*Test Driven Developement (TDD) using C++ and Catch2*"

No només cal executar el codi per obtenir una validació sobre el seu estat. Els "static tests" o tests estàtics es basen en analitzar el propi fitxer del codi per obtenir la

informació que es desitja. D'aquesta forma es simplifica encara més el procés de validació ja que no els cal compilar l'aplicació. A partir dels tests estàtics es pot obtenir dades sobre la complexitat ciclomàtica, l'acoblament o la cobertura del codi. També son útils per garantir que els desenvolupadors segueixen les convencions establertes de nomenclatura de variables i classes.

6.2.2 Pràctiques de la integració de Software

Com s'ha indicat prèviament en la introducció del tema, per a obtenir la sèrie de beneficis que un sistema d'integració continua demana cal ser meticulós amb les diferents pràctiques que l'acompanyen.

Aquesta llista de praxis s'ha extret a partir de la unió de diferents articles [18, 19, 20] que tracten el tema amb profunditat. És pot llegir la llista exhaustiva seguint les referències en la bibliografia.

6.2.2.1 Automatització de la *build*

En el punt 6.1.2 sobre la compilació d'un programa en C++ s'ha definit què significa realitzar una *build*. Dins del context de la integració continua aquesta semàntica varia significativament i augmenta les fases que cal realitzar: Compilació, execució de tests, guardar les diverses configuracions...

No només és més complicat realitzar el procés d'una build, sinó que el propi sistema d'integració contínua demana automatitzar-la. Això significa donar eines als desenvolupadors per a que no hagin de preocupar-se del procés intern, només del anàlisi del resultat. La forma més senzilla i documentada per automatitzar aquests processos sol ser l´ús dels llenguatges de scripting integrats en el propi sistema operatiu, com ara els Shell scripts o els ".bat". També les eines de compilació com el CMake permeten aquesta automatització i configuració de projectes.

6.2.2.2 Centralització dels artefactes

Per realitzar l'automatització efectiva de builds a partir del codi cal mantenir tota una sèrie d'arxius i binaris en un mateix lloc. Per una banda, els canvis que els diversos desenvolupadors realitzen en el codi han d'estar organitzats, generalment per mitjà dels sistemes de control de versions.

Les dependències dins d'una pipeline d'integració continua han d'estar molt ben controlades. Les diverses versions d'un programa extern poden afectar de forma substancial el rendiment o la funcionalitat de l'aplicació que s'està construint. Per aquest motiu és bastant comú utilitzar els anomenats magatzems d'artefactes: Col·leccions enumerades i versionades dels diferents binaris que es poden obtenir de forma centralitzada en un sol lloc, generalment en un servidor local de l'empresa, i mantinguda per un sol responsable. Amb això es pot treure aquesta responsabilitat als programadors de l'aplicació evitant així diferències entre els seus dispositius de desenvolupament.

6.2.2.3 Realitzar *commits* i *builds* amb freqüència

Una de les claus més importants per a un sistema d'integració continua funcional és executar les builds recurrentment. Per això cal realitzar *commits* quan s'han realitzat molts pocs canvis en el codi, assegurant que un error amb el mínim canvi possible ja és detectat pels sistemes de testeig.

Aquest punt dona molta importància al fet que hi hagi un procés de build que ha estat automatitzat, doncs se suposa que els desenvolupadors ho hauran d'estar activant constantment.

6.2.2.4 Agilitzar els processos de compilació

Tant el primer com l'últim punt necessiten d'una característica crucial per a fomentar les pràctiques de la integració contínua: Agilitat en la compilació. El feedback de les diferents proves i tests es poden veure afectats negativament si es tarda molt a compilar el programa. Per això, és important ser conscient de la branca de dependències que s'ha implementat en el programa i evitar la recompilació de codi duplicat (Per exemple incloent llibreries sense estudiar si calen en aquell apartat).

Un altre error molt comú que augmenta els temps de compilació és posar codi i lògica que pot canviar en els fitxers de tipus ".h", ja que obliga a tots aquells fitxers ".cpp" que els inclouen a ser recompilats. De fet, en el propi article sobre la integració continua en projectes de C i C++ [19] amb els punts més importants a tenir en compte, com ara incrementar el hardware dedicat a compilar, optimització de la infraestructura i del sistema per fer la build o distribuir el sistema d'integració en diferents equips.

6.3 El desplegament de Software

Un dels puts més crucials del treball consisteix a entendre que tot programa desenvolupat té com a objectiu executar-se en una sèrie de dispositius finals (ordinador, mòbil, xip...) definits per uns requeriments en la fase de disseny [21] per tal d'aportar valor als usuaris finals. Sovint aquesta decisió va acompanyada per aspectes que s'escapen d'un camp purament tecnològic com ara la usabilitat, la quota de mercat per a monetitzar l'ús del software o l'interès dels *stakeholders* [22]. A part, el procés d'enviament d'un programari al dispositiu de l'usuari final també comporta dificultats que a la llarga no es poden negligir. Amb aquestes problemàtiques en ment neix la necessitat de dissenyar una sèrie de processos que asseguren la disponibilitat del software i mantenir-lo operatiu en els dispositius "objectius" [21, 22].

6.3.1 Conceptes introductoris

Les dinàmiques de desplegament de software neixen a partir de la necessitat de connectar dos agents diferents: Els dispositius finals (consumidors) i el programa desenvolupat (sistema de software). Sovint, els dispositius on s'ha creat el software (també anomenats productors) no tenen les mateixes característiques que els seus consumidors. El component més important d'aquests dos agents es pot trobar en l'article "*A Characterization Framework for Software Deployment Technologies*" [16] on s'esmenen les següents definicions:

- Un sistema de software és una col·lecció coherent d'artefactes, com per exemple arxius executables, codi o dades i que son necessaris en el dispositiu final per a poder funcionar.
- Un dispositiu conté una sèrie de recursos necessaris per executar codi, com la memòria, l'espai en el disc o l'arquitectura.

D'aquesta forma es pot definir l'acció de desplegar un software com "*la transferència d'un programa des del dispositiu productor a un o més dispositius consumidors*". Donat que el fet de realitzar aquesta transferència requereix d'un control meticulós, l'article prossegueix desgranant aquesta acció en diverses fases, algunes de gran importància pel desenvolupament teòric del treball.

6.3.2 El procés de desplegament

6.3.2.1 Release

La primera fase anomenada "*Release*" o publicació consisteix en combinar els processos de desenvolupament amb els de desplegament. D'aquesta forma el software pot ser enviat als dispositius finals amb tots els artefactes necessaris per la següent fase.

6.3.2.2 Instal·lació

És el procés de inserció del sistema preparat en la fase de la release al sistema del consumidor. S'han de tenir en compte tots els sistemes necessaris per a que el codi pugui funcionar en el lloc desitjat, a part de dissenyar la forma en la que es faran arribar tots els arxius de la release.

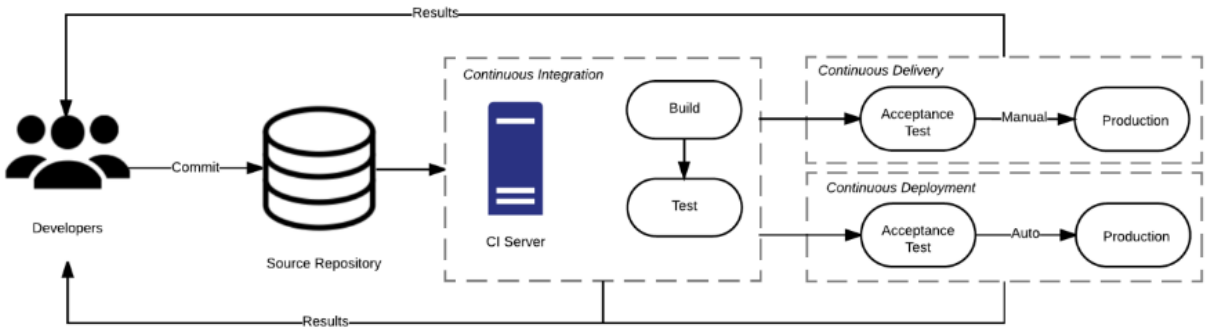
6.3.2.3 Activació

Posada en marxa del codi en el dispositiu objectiu amb totes les funcionalitats configurades. Aquest pot ser un procés simple com clicar a sobre d'un executable o complicat per a sistemes on hi ha la necessitat de processos externs com la connexió a una base de dades.

6.4 Continuous Software Engineering

La enginyeria de software continua és una pràctica que s'està estenent actualment. Comporta desenvolupar, desplegar i obtenir feedback tant del software com del client en cicles extremadament ràpids per aportar coneixement de forma constant [23]. L'objectiu final és bastant similar al procediment Lean, on en comptes de separar el procés en una seqüencia d'activitats discretes, la idea és obtenir un moviment continuat de feedback i aprenentatge [24].

Aquests resultats s'obtenen gràcies a tres activitats de desenvolupament i que conformen la base de les pràctiques de DevOps: Integració Continua, Entrega Continua i Desplegament Continu.



Imatge 11: Resum de les diferents activitats del desenvolupament continu. Extret de l'article *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*

6.4.1 Integració Continua

La idea darrere la integració continua és combinar el treball dels diferents desenvolupadors de forma constant, idealment després de cada canvi [23]. D'aquesta forma l'equip pot permetre's el luxe de publicar més "releases" d'una qualitat superior. Els filtres pels quals un canvi és acceptat o no pot variar segons el disseny dels processos, però generalment és obligatori haver passat tots els tests unitaris i assegurar-se de no haver trencat cap funcionalitat important.

El servidor de la integració continua és un dispositiu que s'encarrega de processar els canvis nous construint una build i executant els tests programats de forma automàtica. Sense aquest agent, seria impossible realitzar una pipeline amb diversos desenvolupadors o amb una freqüència d'interacció tan alta com ho demana la pròpia metodologia.

6.4.2 Entrega Continua

La principal missió de l'entrega continua és assegurar que sempre hi ha una aplicació llesta per ser enviada a producció [23]. L'obligació de confrontar els nous canvis amb una qualitat similar a una entrega final redueix els riscos alhora de desplegar finalment el software i permet obtenir feedback molt més ràpid del normal.

6.4.3 Desplegament Continu

És l'evolució de l'entrega continua on a part d'assegurar un bon estat de l'aplicació en tot moment el sistema també l'envia automàticament a l'entorn de producció [23].

Aquest canvi té una connotació important a l'hora de tractar el software final: L'adopció de pràctiques de desplegament continu es basen en actualitzar de forma constant el software que utilitza l'usuari. Això ocasiona que aquestes praxis no siguin compatibles amb alguns casos d'ús on calgui controlar quan i com s'ha de fer una actualització.

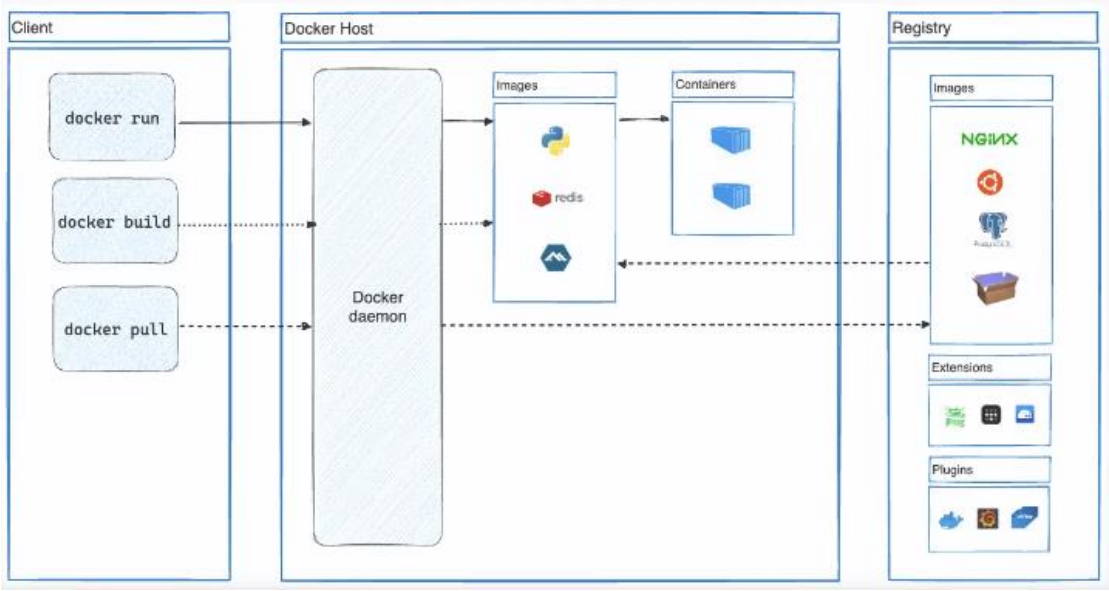
6.5 Aplicacions utilitzades

Durant el desenvolupament s'han fet servir un conjunt d'aplicacions que ajuden en el procés de l'enginyeria de software continu. Aquest apartat final té com objectiu explicar de forma concisa la lògica d'aquestes aplicacions i el seu vocabulari essencial. Amb això es vol aconseguir que els lectors tinguin una millor comprensió quan s'expliqui el desenvolupament del treball.

6.5.1 Docker i virtualització

Docker és una plataforma per a poder separar les aplicacions de la infraestructura que les acompanya i de forma totalment aïllada [25]. Per al desenvolupament en C++ aquesta tecnologia és clau ja que es pot compilar per a les diferents infraestructures evitant problemes d'instal·lació o de dependències.

Els principals agents que participen per a fer ús d'aquesta tecnologia son el client, el Docker Daemon, les imatges i els containers.



Imatge 12: Arquitectura de Docker (imatge extreta de la documentació oficial).

El Docker daemon (dockerd) és el procés que s'executa en la màquina del host i que controla totes les accions i objectes de Docker. És qui té la informació per a iniciar nous ambients o de mantenir la base de dades de les imatges que es necessiten. Per a poder interaccionar amb el daemon els usuaris poden fer servir el Docker client: Una sèrie de comandes com "docker run" o "docker image ls". Un client es pot comunicar amb un o més daemons en dispositius diferents.

Dins del context de Docker es poden trobar diferents objectes que realitzen funcionalitats i que son controlats pel daemon. Una imatge és un llibre d'instruccions que determina com crear un ambient de treball. Sovint son formades a partir d'altres imatges ja creades, com ara la imatge d'un ubuntu, d'un Windows o d'un servei de base de dades. D'aquesta forma els usuaris poden crear les seves pròpies imatges amb la mínima configuració. Un container és l'objecte que es crea quan una imatge s'executa i es crea una instància. Es pot configurar de diverses formes, com ara definint una xarxa, un volum de dades dinàmic o un punt d'entrada d'execució. La norma principal d'un container és que no reté els canvis que s'han efectuat en ell un cop s'elimina del dispositiu.

Tots els containers tenen el mateix punt de partida: la imatge que els forma. Això permet poder realitzar proves que son reproduïbles i deterministes.

6.5.2 Servidors d'integració continua (Jenkins, TeamCity)

Els servidors d'integració continua son serveis que donen control als enginyers de DevOps per configurar diferents processos i automatitzar-los. El treball s'ha efectuat finalment amb el software de JetBrains anomenat TeamCity. TeamCity és un software de gestió d'integració continua i de desplegament continu que permet la flexibilitat en la definició de pràctiques de treball i en la de pipelines de desenvolupament [26]. Com a tot sistema de CI/CD, ofereix l'estructuració de builds separades per diferents projectes de forma que es facilita el coneixement del seu estat i les accions que cal realitzar a continuació.

Generalment els softwares que gestionen aquestes tasques estan orientats a treballar en un servidor ja que controlen l'execució de diverses màquines (físiques o virtuals) en funció de les accions que els desenvolupadors necessiten. Això significa que el servidor o controlador fa la funció d'un director d'orquestra (anomenat orchestration):

relleva la feina a aquells agents que son compatibles i que estan lliures en aquell moment. Per posar un exemple específic, si és necessari executar un conjunt de proves en una màquina Windows amb arquitectura ARM de 32 bits, el servidor és qui revisa la llista d'agents disponibles cercant-ne un que compleixi amb aquest requeriments.

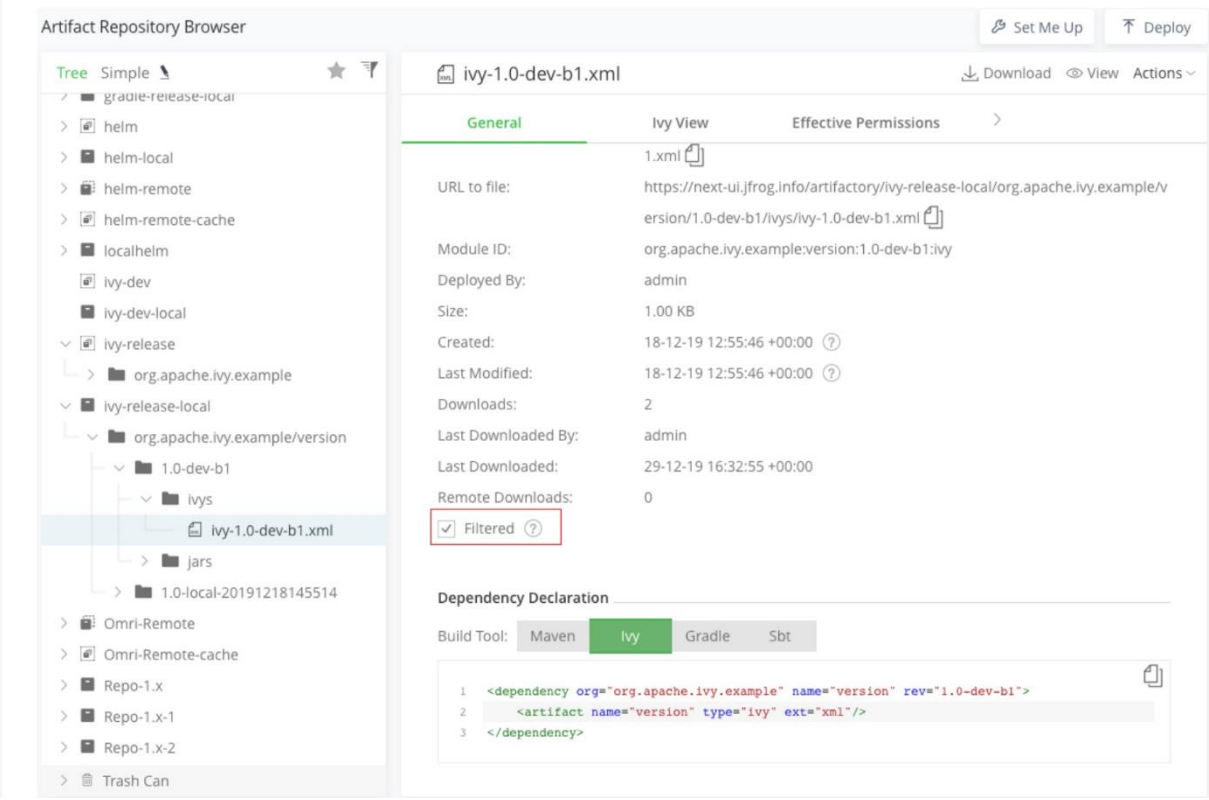
Imatge 13: Interfície del servidor de TeamCity

És important destacar que aquests agents (el nom oficial que molts servidors de CI utilitzen) no poden treballar per si mateixos i estan sempre a l'espera del controlador.

6.5.3 Artifactory

Artifactory és una solució creada per Jfrog i que serveix per a emmagatzemar de forma centralitzada artefactes, binaris i arxius que s'utilitzen durant la creació de software [27]. Així doncs la idea bàsica darrere Artifactory és la creació de servidors privats per a l'abastiment ordenat de dependències. D'aquesta forma els desenvolupadors o pipelines de CI poden descarregar el que necessiten amb la versió pertinent sense dificultats.

Hi ha diverses opcions per a començar a utilitzar Artifactory. La versió de pagament conté totes les característiques desbloquejades com ara la capacitat de poder enllaçar projectes via Github de forma automatitzada o la creació de servidors d'abastiment de paquets com ara npm. Per sort, la versió gratuïta és suficient pel que fa l'abast del treball, ja que permet crear repositoris simples. S'entén per repositori un espai virtual d'emmagatzematge d'un recurs o informació. La comunicació amb aquests respositoris i la seva estructura interna és molt similar a la d'un servidor FTP.



Imatge 14: Exemple d'un conjunt de repositoris en la versió de pagament de Artifactory.

Més enllà de guardar arxius, Artifactory també juga un paper molt important a l'hora de guardar metadades: Informació addicional que ajuda a organitzar el contingut del projecte. Un exemple molt útil és el registre de versió de l'artefacte.

7 Desenvolupament

El cas on s'aplica tota la teoria esmentada en el marc teòric és Discland, un bot desenvolupat plenament en C++. És un bon exemple amb el que experimentar ja que a part d'estar programat amb el llenguatge que s'està estudiant té algunes dependències que necessiten ser gestionades. A més a més, també té un petit servei de base de dades pel que acaba de completar els casos de desplegament i d'activació on hi ha més d'un procés a tenir en compte.

Val la pena recordar que la intenció del treball no és plantejar una millora de processos escaient als requeriments del desenvolupament. Per a nombres molt reduïts de treballadors algunes d'aquestes mesures podrien considerar-se excessives. El principal objectiu és investigar el procés de creació d'una pipeline d'integració i desplegament del C++ per així observar quins problemes arriben a sorgir. En un ambient controlat com aquest on l'aplicació ja està "tancada" es veu factible poder fer front als possibles contratemps i així poder aportar solucions de cara una futura investigació.

7.1 Sprint 1: Estudi inicial i cerca d'alternatives

El primer punt del desenvolupament passa per saber quins requeriments té el projecte per així poder començar a plantejar les primeres alternatives. L'objectiu es saber com atacar les diferents dependències i quins canals de distribució val la pena construir.

7.1.1 Plataforma objectiu

En l'apartat inicial sobre C++ ha quedat estipulat que aquest llenguatge és sensible a l'arquitectura i el dispositiu objectiu. D'aquesta forma el primer pas és definir aquesta variable. Donat que no es preveu que els usuaris finals executin el codi sinó que el servidor proveeixi les respostes pertinents, l'aplicació només serà compatible amb el sistema operatiu Ubuntu 24.04.

7.1.2 Dependències

Per a realitzar les tasques principals de l'aplicació es necessiten una sèrie de dependències externes. S'ha definit que qualsevol d'aquestes llibreries necessita ser compilada amb la versió mínima de C++ 20 per assegurar la homogeneïtat entre elles.

A part, s'han de mantenir actualitzades tant les versions de Debug com les de Release per així poder realitzar les comprovacions necessàries del codi o efectuar el desplegament quan sigui necessari. Una build en Debug és més lenta però s'inclouen variables de depuració. Una build en Release està altament optimitzada per córrer ràpid o ocupar poc espai.

A continuació s'explica breument els requeriments de les dependències de l'aplicació:

7.1.2.1 DPP

Discord Plus Plus és una llibreria open source que permet la connexió automàtica amb els serveis de Discord, implementant la seva API i gestionant la comunicació d'esdeveniments [28]. Està molt ben mantinguda i ofereix una documentació excel·lent per a poder automatitzar-ne la seva compilació.

7.1.2.2 Protobuf

Google Protobuf permet la creació de codi automàtic a partir d'unes plantilles generades pel propi programador [29]. D'aquesta forma es poden actualitzar els requeriments o la forma de representació de les dades internes en molt poc temps. També proveeix de diferents convertidors que transformen objectes JSON en objectes de C++ i viceversa.

Aquesta llibreria és un cas una mica especial ja que donat que genera codi per a compilar-lo dins el propi projecte, no cal mantenir un seguiment tant rigorós com la resta. Per aquest motiu es distribuirà la llibreria mitjançant el propi cercador de paquets de Ubuntu *apt-get* ja que permet especificar la versió que es vol fer servir.

7.1.2.3 LibGD

LibGD és una llibreria escrita principalment en C que permet l'edició d'imatges en format PNG, JPEG i TIFF a través del codi [30]. És una de les llibreries més antigues que es fan servir en el conjunt del programa, pel que es preveu que doni més problemes a l'hora de passar pels processos d'integració i automatització. A part, conté diverses dependències externes que cal revisar ja que no estan incloses dins la solució, com la renderització de text o l'exportació a PNGs.

7.1.2.4 Catch2

Catch2 és la llibreria per excel·lència per implementar testing automàtic unitari en un programa amb C++ [31]. No té cap dependència externa i està totalment integrada en l'entorn de CMake, pel que el seu tractament dins de l'entorn d'integració serà bastant directe.

7.1.2.5 Mongocxx

Mongocxx és el driver necessari per a poder interactuar amb un servidor de MongoDB a través de codi escrit en C++. El principal problema és la seva forta dependència amb mongoc, ja que està construït a sobre d'ell. Això significa que la plataforma de CI haurà de mantenir actualitzats els dos projectes i utilitzar els binaris de mongoc per compilar mongocxx.

7.1.3 Anàlisi de programari

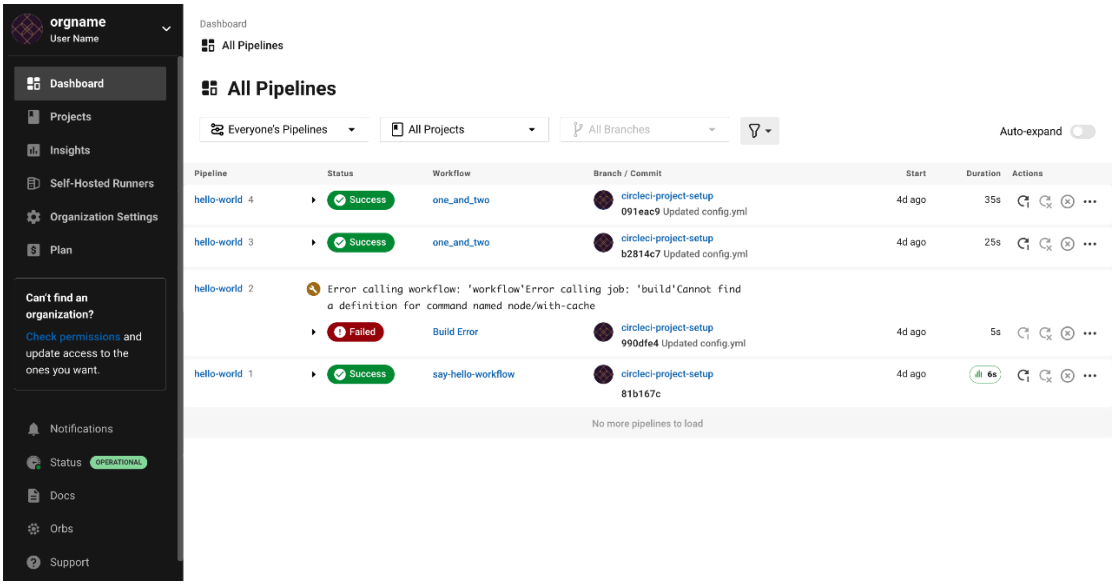
S'ha fet una llista d'algunes solucions per a la creació de servidors d'integració continua.

7.1.3.1 Jenkins

Jenkins és un programa bastant famós i totalment gratuït. Porta utilitzant-se molts anys i és una de les primers opcions de cara al desenvolupament del treball. No només sembla ser que compleix amb els requeriments principals que s'han marcat sinó que el procés d'instal·lació es pot realitzar a través de Docker. La clau principal de Jenkins és que l'usuari que l'utilitza crea el seu propi servidor en local o en una instància al núvol. La gestió en aquest cas és responsabilitat total del desenvolupador o empresa.

7.1.3.2 CircleCI

CircleCI ofereix una plataforma web per a poder interactuar amb el conjunt de servidors i agents. A diferencia de Jenkins, no hi h cap procés d'instal·lació: Tot passa través de la seva interfície. És una opció factible ja que es permet la creació d'un usuari gratuït amb accés a totes les característiques necessàries per a la realització del treball. No obstant, ja que el seu ús és totalment via web pot amagar processos d'implementació valuosos per a la investigació sobre el tema.



Imatge 15: Portal web de Circle CI (dashboard) per on es poden gestionar les pipelines del projecte.

7.1.3.3 Artifactory

És un dels gestors de paquets més famosos. Molts servidors de CI ofereixen sistemes d'integració automàtica per a la lògica d'Artifactory. No només això sinó que sembla ser que la documentació està ben actualitzada i hi ha bastants recursos per poder informar-se adequadament.

7.1.3.4 TeamCity

Creat per l'empresa JetBrains, és el competidor directe de Jenkins ja que la seva proposta de valor és molt similar: creació dels teus propis servidors de CI on premise. Sembla ser que a part utilitzen el mateix vocabulari cosa que pot significar certa facilitat a l'hora de migrar el coneixement entre ambdues solucions. La documentació sembla ser més completa en el cas de TeamCity. Finalment, la versió on premise és gratuïta amb certes limitacions tot i que no es preveu que afecti a l'objectiu final del treball.

7.2 Sprint 2: Preparació de l'entorn amb Docker

Amb tot el procés d'investigació realitzat, el desenvolupament comença tractant amb un dels fonaments més importants del treball: Docker. Donat que es vol compilar

diferents dependències que tenen requeriments molt versemblants, s'ha contemplat la necessitat d'utilitzar aquesta tecnologia en la futura pipeline d'integració.

El primer pas ha sigut instal·lar Docker en la màquina Ubuntu principal. Això es pot fer anant a la pàgina principal de Docker i revisant la guia per a sistemes operatius Linux. A partir d'aquí es pot començar a dissenyar l'arquitectura que ha de tenir el sistema:

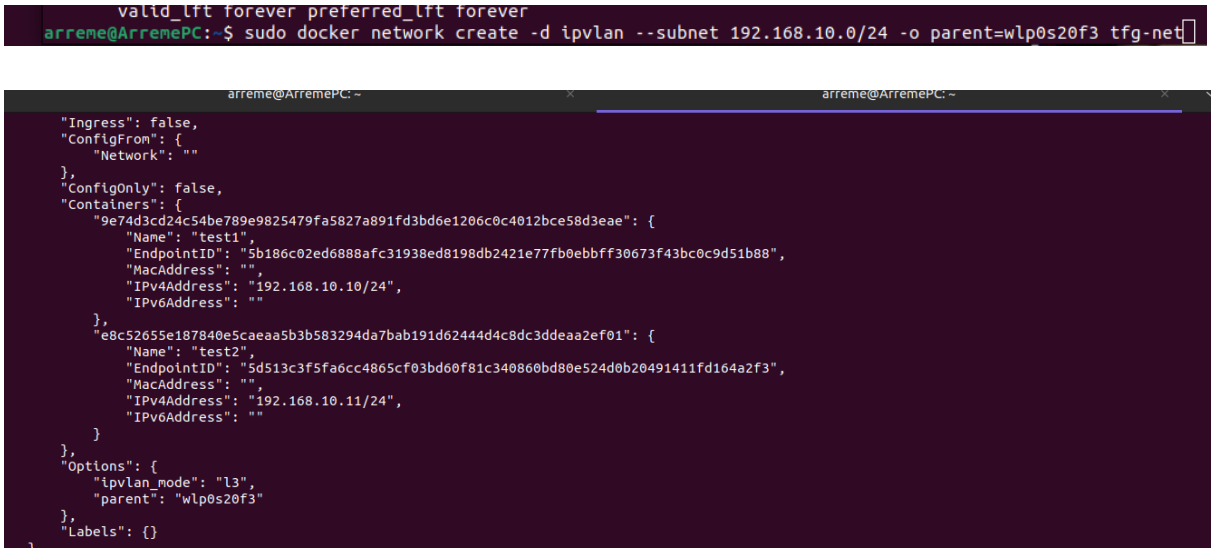
- El servidor d'integració continua ha de tenir accés als diversos agents per a poder controlar el seu estat i la llista de requeriments. Tant el servidor com els agents seran containers creats per Docker ja que no es disposa de hardware físic en aquest moment.
- El servidor d'integració continua i els agents han de poder comunicar-se amb el servidor encarregat de gestionar els artefactes que es generen.
- Els containers han de tenir accés a internet per a poder descarregar les dependències i mantenir-les al dia.
- Es vol gestionar els diferents docker com si fossin dispositius físics en l'entorn de desenvolupament. Això es fa per donar la sensació que son servidors totalment separats de la resta.

Aquests requeriments indiquen la necessitat d'un cert plantejament més profund a com es faran les comunicacions entre els diferents containers. Per sort Docker permet la gestió d'aquestes comunicacions.

7.2.1 Docker Networks

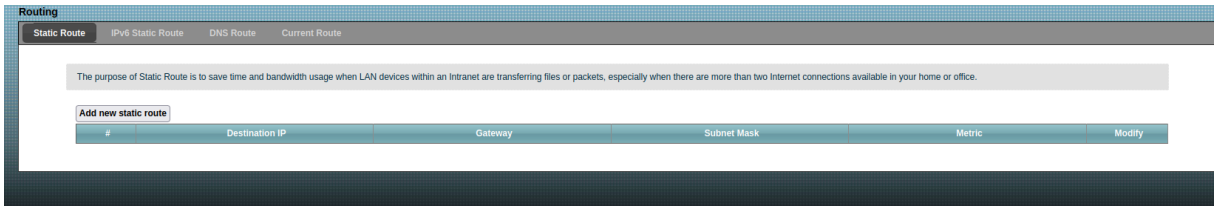
Per a cada container, Docker permet l'elecció de diferents drivers que modifiquen com es realitzen les connexions internes de la instància. Per exemple, el mètode més comú anomenat "bridge" o pont permet la creació de subxarxes a través d'una interfície virtual en el propi host. D'aquesta forma es poden connectar les diferents instàncies a la mateixa subxarxa per a que puguin parlar entre elles. El principal problema a treballar amb la gran majoria de drivers és que estan centrats en la capa 2 del model OSI, cosa que complica la interconnexió entre els dispositius ja que cal tenir en compte els possibles bucles que es poden originar. A part això també significa l'enviament constant de senyals de tipus broadcast a la xarxa.

El driver IPVLAN de capa 3 s'ha establert com la nova forma recomanada a llarg termini per a realitzar aquest tipus de comunicacions i soluciona tots els problemes que s'han exposat anteriorment. La idea principal resideix en passar totes aquestes connexions entre les subxarxes de la capa 2 a la capa 3.



Imatge 16: A dalt: comanda que genera la configuració de la subxarxa. A baix: Mostra de prova on els Dockers han rebut la seva pròpia IP a partir de la subxarxa virtual creada.

L'únic problema que sorgeix al realitzar tot aquest procés és que el router que dona internet als Dockers es incapaç d'accedir-hi sense configurar una ruta estàtica. En el cas del treball s'ha realitzat justament això per al router domèstic.

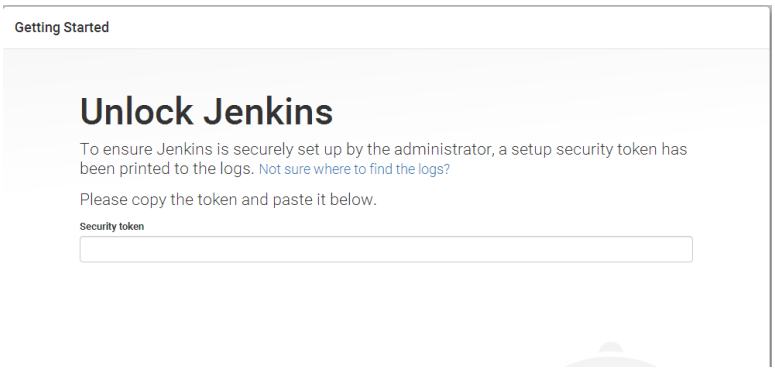


Imatge 17: Terminal de configuració de les rutes estàtiques del router

Un cop establerta la comunicació entre els dockers ja es pot començar a pensar sobre les aplicacions i serveis que poden tenir a dintre.

7.3 Sprint 3 i 4: Integració continua: Jenkins i errors

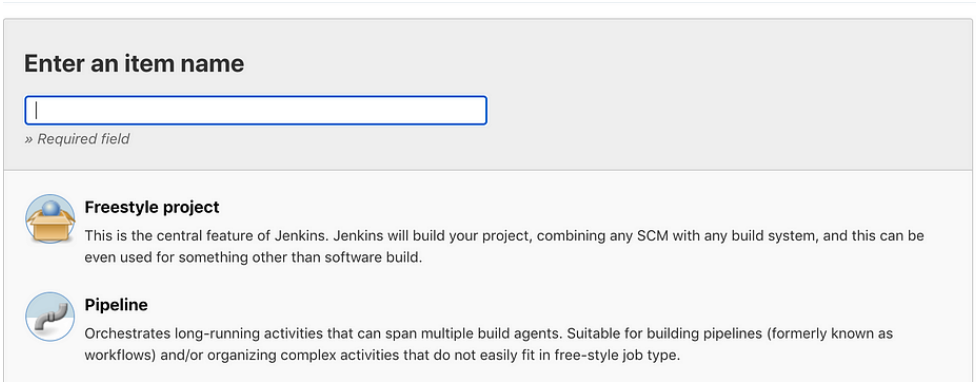
La primera aplicació que s'ha intentat integrar ha sigut Jenkins, un controlador de pipelines de CI i CD molt coneguda en el mercat i que té força renom. Per provar-la s'ha agafat la imatge del servidor ja construïda a través del repositori d'imatges oficials de Docker, anomenat Dockerhub. Aquestes imatges més complexes sovint demanen la configuració de volums: unes carpetes especials on guardar la informació en cas que el container s'esborri o deixi de funcionar.



Imatge 18: La primera pantalla al iniciar una nova instància de Jenkins

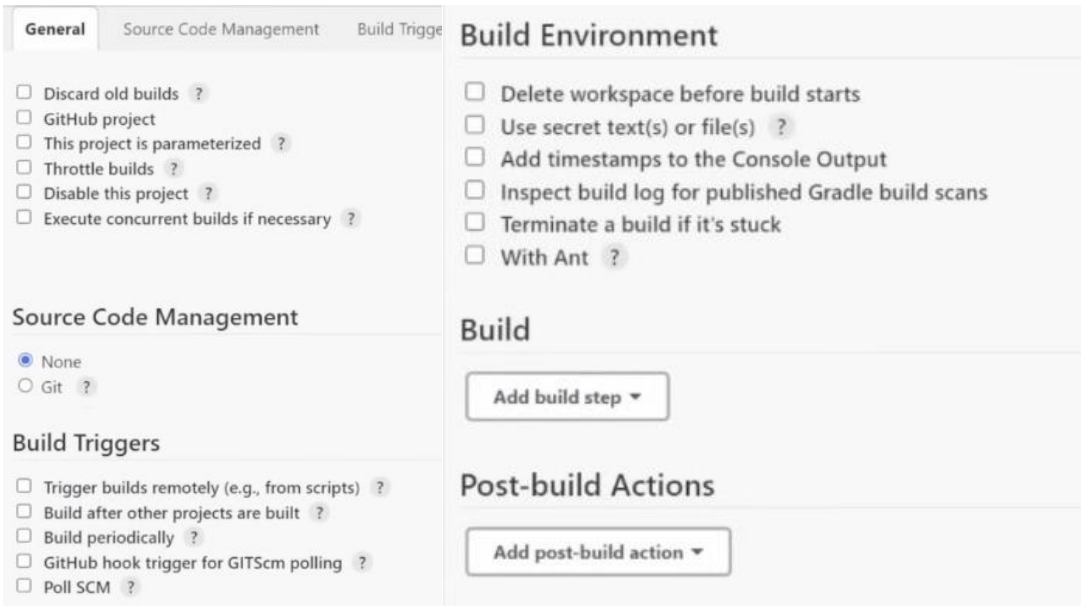
Un cop s'ha inicialitzat Jenkins es demana la inserció d'un codi secret (Imatge 18) que es pot trobar escrit en la terminal del procés. Després de varies pantalles més de configuració finalment es demana la informació per iniciar sessió d'un usuari administrador que serà qui s'encarregarà de configurar les "*pipelines*" per a la resta de l'equip.

El següent procediment que cal realitzar és la configuració d'un projecte on poder crear les diferents automatitzacions per a cada una de les dependències. La idea principal de Jenkins és la definició de diverses tasques que es centren a completar una funció específica. Per exemple, els projectes lliures o "*freestyles*" són tasques simples acotades a una sola acció, com executar els tests d'una aplicació. Amb una pipeline, d'altra banda, es poden programar diferents fases com ara compilar, testejar i realitzar un deployment per a una sola branca del repositori.



Imatge 19: Creació d'objectes per a treballar amb Jenkins

Al decidir que es vol crear un projecte "*freestyle*" Jenkins obre una pantalla amb un cúmul de configuracions necessàries per a la correcta automatització de la tasca:

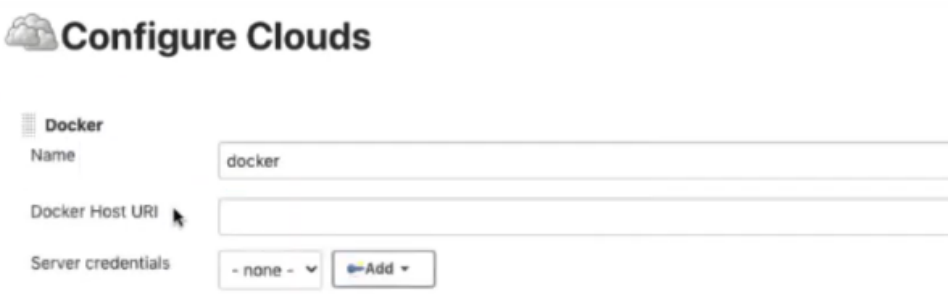


Imatge 20: Diferents opcions d'automatització

Una de los opcions més importants que s'ha de configurar és l'obtenció del codi font per a realitzar les accions. Com ja s'ha especificat en el marc teòric, la necessitat de tenir un projecte amb un controlador de versions és molt important al treballar amb servidors de CI. També es poden definir "*triggers*" o actuadors per a poder executar proves o builds de forma automàtica al detectar un push (un canvi en el codi) d'un desenvolupador en el projecte. També es pot definir les diferents fases que l'automatització ha d'executar, com una comanda per terminal o una acció de Maven (si es treballa amb Java, per exemple).

El problema principal que s'ha trobat amb Jenkins a l'hora de realitzar el treball ha sigut amb la generació d'agents capacitats per a l'execució de les configuracions definides. Jenkins ofereix una opció en la documentació bastant còmoda per a treballar amb Docker per a generar de forma automàtica contenidors compatibles a partir d'imatges ja definides. El resultat de l'experiment és que no s'ha pogut establir aquest servei de forma dinàmica tal i com es volia. A part, tant els missatges d'error com la documentació disponible no han ajudat gaire en tot aquest procés. A continuació s'explica el procediment que s'ha realitzat i els errors que s'han trobat.

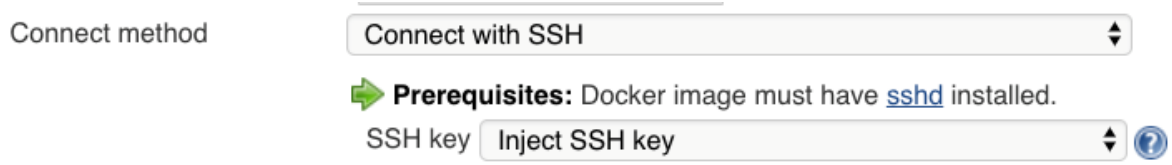
El primer pas a tenir en compte és instal·lar l'extensió de Docker dins del servidor de Jenkins. Després de reiniciar el servidor les noves funcionalitats ja es poden trobar disponibles. Ara la configuració d'agents en remot té una opció per definir la direcció IP del Docker daemon per on Jenkins actuarà com a client.



Imatge 21: Configuració dels agents en remot a través del plugin de Docker

A pesar de tota aquesta configuració creada, Jenkins no pot connectar-se a un ordinador o container de forma automàtica. Els agents que han d'esperar les ordres del controlador han de tenir un JDK instal·lat i un "agent.jar" en execució. A part, la connexió del servidor al agent es pot realitzar de diverses formes:

- Via SSH: Significa "Secure Shell". És un protocol que permet la comunicació entre dos ordinadors a partir d'un usuari i contrasenya. És important destacar que això implica que el container executant-se ha de tenir el servei sshd (ssh daemon) en execució.



Imatge 22: Connexió via SSH

- Via JNLP: Les sigles venen de Java Network Launch Protocol. Segons la pàgina oficial d'Oracle [32] aquesta funcionalitat permet l'execució d'una aplicació en un client utilitzant els recursos que estan en un servidor en remot.
- Via adjunció: El menys comú. El controlador té accés als fluxos d'entrada, sortida i d'error. Per a realitzar això s'utilitza la comanda "attach" que proporciona el client de Docker.

S'ha intentat aconseguir la connexió esperada provant de forma exhaustiva aquestes tres configuracions, però al final no s'ha arribat al resultat satisfactori: el servidor de Jenkins és incapaç de trobar i vincular el Docker tot i trobar-se en el dispositiu. També s'han provat plugins (funcionalitat extraordinària creada per la comunitat) que en teoria faciliten el procés però tampoc han acabat de funcionar.

Després de bastantes hores sense aconseguir avançar, s'ha decidit passar a la segona opció en la llista dels servidors d'integració continua: TeamCity

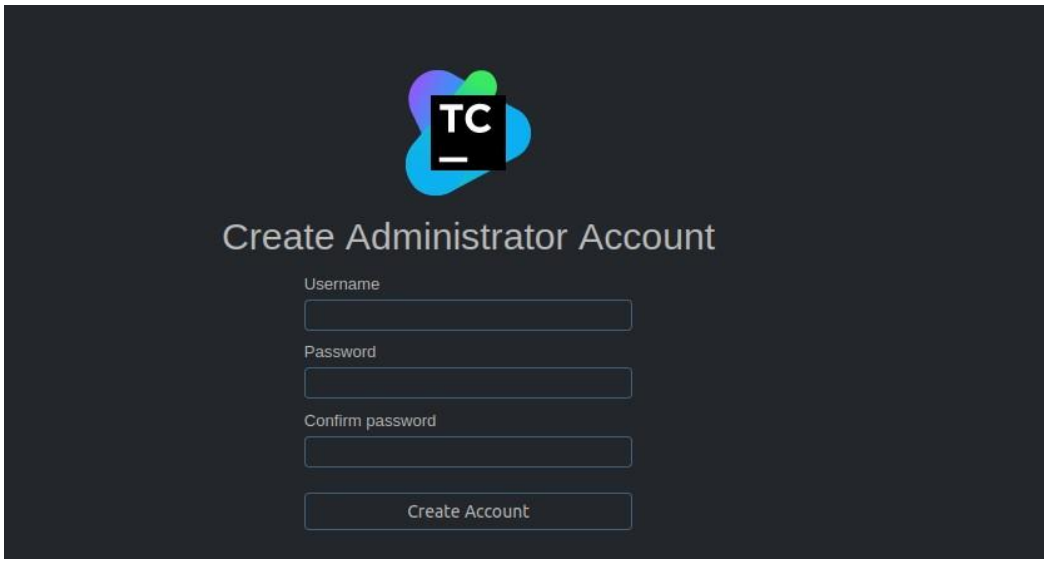
7.4 Sprint 5: TeamCity i primers Artefactes

TeamCity està disponible directament per Docker a través de la plataforma DockerHub. Allà mateix es pot obtenir la versió que es desitja i el sistema operatiu del container. En el cas del treball s'ha descarregat de la següent manera:

```
"sudo docker run --name teamcity-server-instance -v
/home/arreme/Documents/TFG/Teamcity/data:/data/teamcity_server/datad
ir -v /home/arreme/Documents/TFG/Teamcity/logs:/opt/teamcity/logs -p
8111:8111 --platform liux/amd64 --network tfg-net
jetbrains/teamcity-server"
```

Aquesta comanda utilitza el client de docker per crear un nou container amb nom "teamcity-server-instance". Després defineix dos volums compartits on guardar les dades del servidor i els registres de logs. També s'exposa el port 8111 que serà per on s'haurà de connectar l'administrador més endevenant. Finalment és defineix la

plataforma, el grup de la xarxa que es va configurar en sprints anteriors i el nom de la imatge amb el que es vol construir el container. Un cop s'ha descarregat i executat, es pot accedir al servidor a partir de la IP 192.168.10.2:8111.

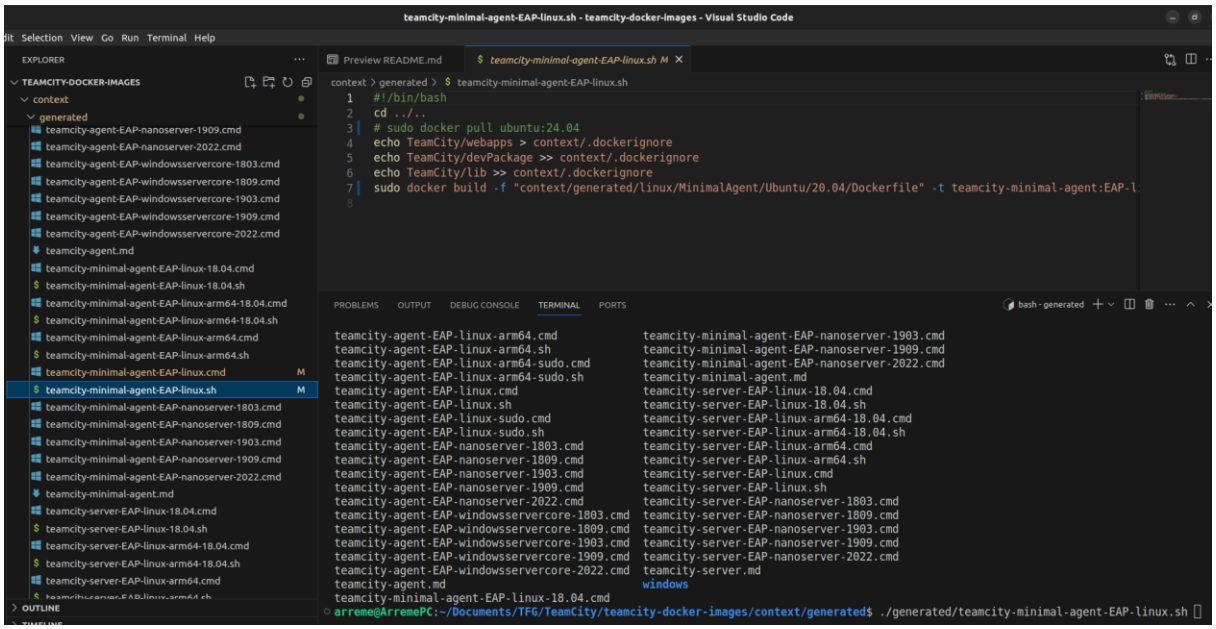


Imatge 23: Pàgina inicial del servidor de CI de TeamCity

Després de crear un usuari administrador, TeamCity obre la pàgina principal de l'equip. En aquesta part ja es poden trobar les primeres diferencies amb Jenkins: Cada projecte té una llista de solucions o builds que actuen sobre ell. D'aquesta forma amb una sola referència al codi font es poden executar diferents configuracions sense repetir la informació. La primera configuració que cal realitzar però es donar accés a TeamCity als projectes privats que estan guardats al núvol a partir d'un token secret generat pel propi Github.

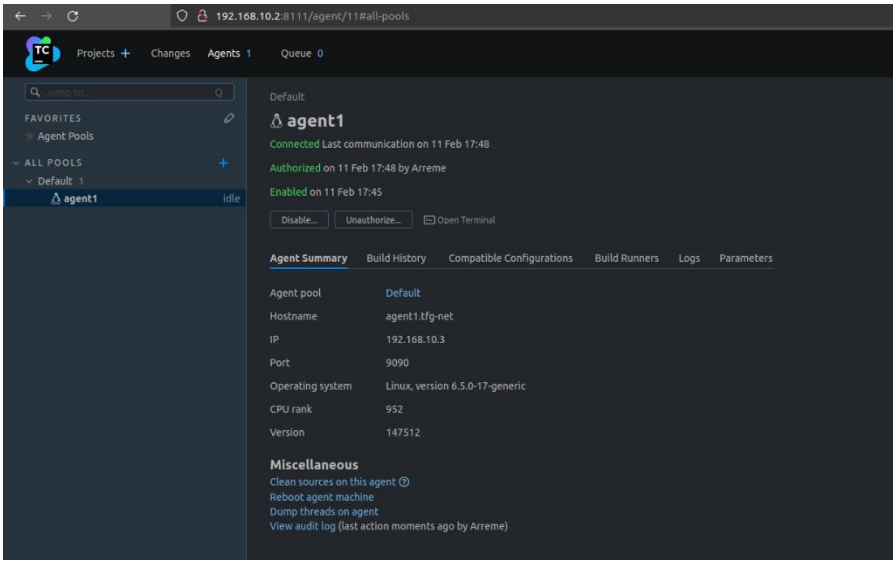
Amb el projecte principal ja integrat, ara es pot començar a realitzar el mateix procediment que s'ha intentat fer a Jenkins. El pla aquest cop és agafar una de les dependències més simples del projecte (D++) i intentar muntar un agent que sigui compatible. Per sort aquest cop el projecte que manté els arxius de creació d'imatges de Docker (Dockerfiles) està molt ben mantingut i això permet crear un agent llest per

a la connexió amb el controlador principal. El resultat és una imatge personalitzada a les necessitats dels projectes que depenen de l'aplicació principal.



Imatge 24: Creació de l'agent personalitzat per a la connexió amb el controlador. S'ha creat un perfil nou amb Ubuntu 24.04

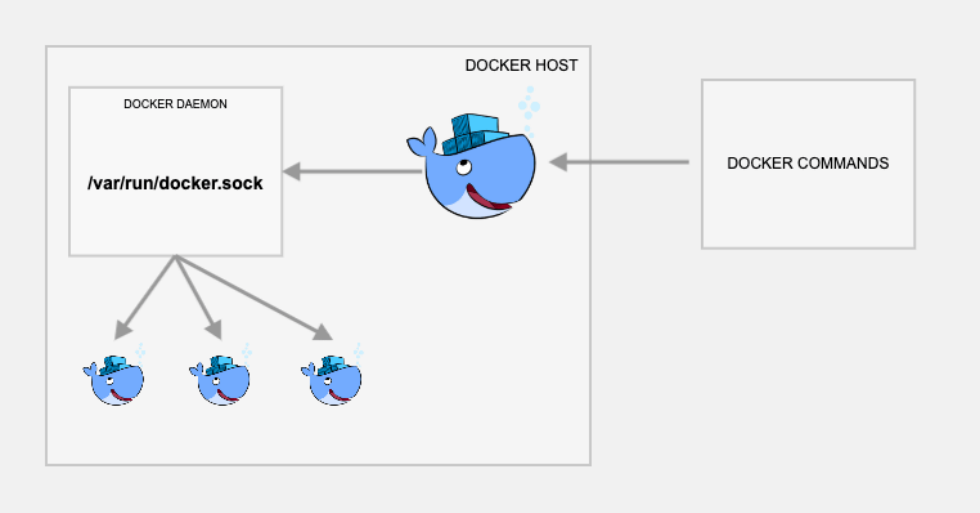
Per executar de forma correcta l'agent, cal definir primer la IP del servidor principal. Un cop creat el container, TeamCity detecta de forma automàtica que hi ha un agent no autoritzat intentant connectant-se al servidor de control. Al donar-li accés, l'agent ja està completament llest per a poder executar les primeres builds de forma automàtica.



Imatge 25: Agent configurat i connectat satisfactòriament

La primera prova és molt senzilla: Utilitzar l'agent configurat amb un Ubuntu per descarregar el codi del projecte DPP i realitzar una build. Per sort l'agent funciona correctament i respon a l'ordre del controlador. Malauradament però la build falla ja que falten dependències per instal·lar. En aquest punt es podria agafar la imatge que s'ha construït i instal·lar tot allò que faci falta, però s'ha de recordar que amb el paquet gratuït de TeamCity només es poden tenir fins a un màxim de 3 agents. A part també s'estaria violant la norma d'aïllament que es guanya amb els Dockers, ja que diferents projectes compartirien el mateix Ubuntu. Així doncs el següent pas és clar: que l'agent (un container de Docker) pugui utilitzar Docker per a crear noves imatges i containers. En un entorn normal amb diferents hardwares, cada equip disposaria del seu propi Docker daemon.

La millor forma d'aconseguir l'objectiu que s'ha proposat és mitjançant els propis volums de Docker. De la mateixa forma que el host és capaç de compartir fitxers amb el container per a guardar la informació, es pot compartir la pròpia instal·lació de Docker del host al container.



Imatge 26: Exemple gràfic sobre el procés de Docker In Docker. El Docker creat pel host té accés a /var/run/docker.sock, habilitant-lo de poder crear i eliminar objectes. Imatge extreta de devopscube.

```
"docker run -e SERVER_URL="http://192.168.10.2:8111" -v
/home/arreme/Documents/TFG/TeamCity/agents/agent1-
data:/data/teamcity_agent/conf -v
/var/run/docker.sock:/var/run/docker.sock -v
/usr/local/bin/docker:/usr/bin/docker -d jetbrains/teamcity-agent"
```

Comanda que executa un container amb la configuració pertinent del agent i ara amb la capacitat de poder executar Docker compartir pel host. S'ha emfatitzat en negreta l'addició en la comanda que habilita aquesta funcionalitat.

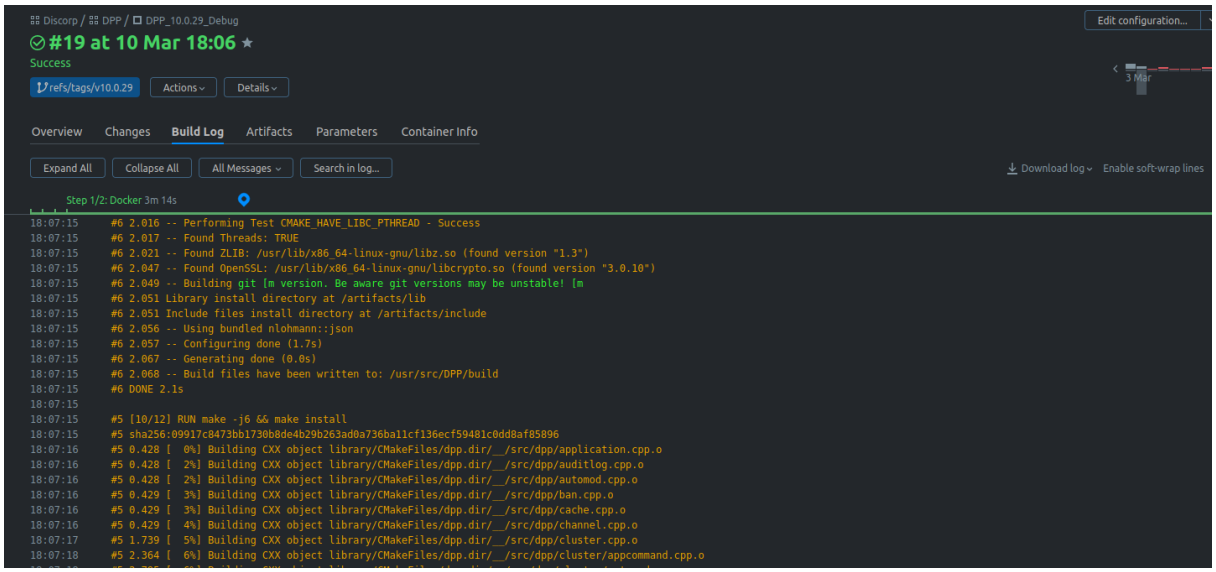
Finalment s'actualitza la configuració de la build per a poder crear i utilitzar la nova funcionalitat dels Dockers. Només cal afegir una descripció en format Dockerfile sobre com generar la imatge:

```
FROM ubuntu:24.04

RUN apt-get update && apt-get install -y libssl-dev zlib1g-dev ca-
certificates libsodium-dev libopus-dev libfreetype6-dev libpng-dev
cmake pkg-config g++ gcc git make
```

```
RUN cmake -DCMAKE_BUILD_TYPE=Debug -DCMAKE_CXX_STANDARD=20 -
DDPP_BUILD_TEST=OFF -DBUILD_VOICE_SUPPORT=OFF -
DCMAKE_INSTALL_PREFIX=/artifact .. && make && make install
```

Aquest arxiu conté dos comandes principals, prefixades per la paraula “RUN”. La primera s’utilitza per actualitzar les referencies a les llibreries enregistrades en la biblioteca oficial d’Ubuntu per posteriorment instal·lar-ne un seguit d’elles. Algunes donen accés a utilitzar certificats digitals que algunes pàgines necessiten per a poder descarregar dades, com és el cas de git a través de HTTPS. Altres son eines bàsiques per a poder compilar i executar programes en C++ com ara cmake, g++ o gcc.

La segona comanda és més interessant ja que serveix per compilar el projecte D++ amb una configuració específica. En aquest cas es vol fer servir una build de debug (sense optimitzacions del compilador) i amb una versió de C++20. També s’està explicitant el lloc on es vol instal·lar tots els artefactes generats. La comanda “make” compila el codi font i la comanda “make install” copia els arxius resultants al lloc d’instal·lació. Amb això i després d’alguns problemes menors s’ha aconseguit generar el primer artefacte a través d’un agent automatitzat per CI.



Imatge 27: Resultat dels logs de la primera build automatitzada

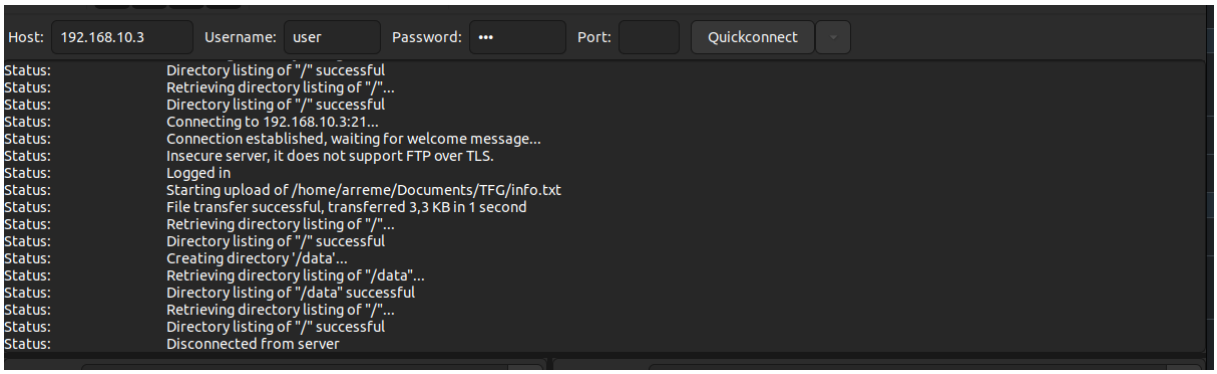
7.5 Sprint 6: Gestió d'artefactes: FTP i Artifactory

Amb tots els arxius resultants de la compilació correctament copiats a una carpeta del container, el següent pas es esbrinar com poder guardar-los per a que altres projectes puguin utilitzar-los.

7.5.1 Primers passos

La primera opció ha sigut utilitzar la pròpia base de dades de TeamCity. Ràpidament però ha sigut descartada ja que no s'ha vist de forma directa la capacitat de poder enumerar els artefactes resultants a partir de la seva versió. A part, sembla ser que cada build genera els seus propis arxius i no hi ha cap opció directa per poder "referenciar" els últims artefactes d'un projecte.

La segona opció que s'ha buscat ha sigut la implementació d'un servidor FTP simple ja que la idea és poder guardar arxius de forma estructurada en un lloc segur i sense gaires complicacions. Un FTP o File Transfer Protocol server és un software que ajuda a passar arxius d'un ordinador a un altre a través de TCP/IP. Gràcies a que es té Docker integrat en la solució, s'ha pogut descarregar una imatge amb un servei FTP ja desenvolupat. Després d'executar el container i instal·lar un client FTP, es poden començar transmetre arxius.

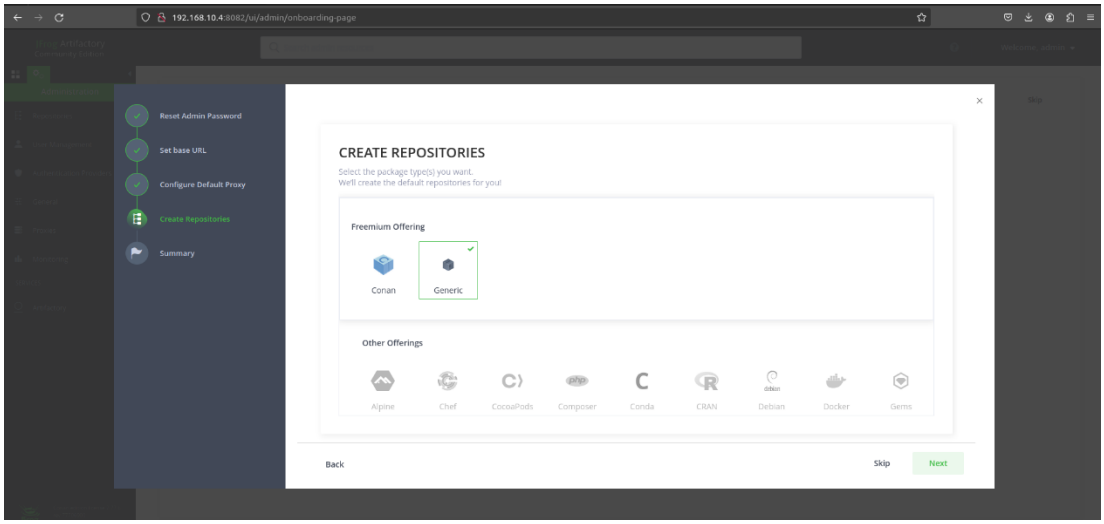


Imatge 28: Client FileZilla connectant-se al servidor FTP en la subxarxa local

El problema principal a utilitzar FTP és que, a pesar de ser una opció molt senzilla de mantenir, les accions alhora de pujar arxius són molt limitaded. S'hauria de crear tot un ecosistema de scripts i automatitzacions per a tenir sincronitzats els artefactes amb el servidor. A part, un FTP no s'encarrega de mantenir cap ordre ni generar metadades que puguin ser d'utilitat a l'hora de gestionar les builds de les dependències externes.

7.5.2 La solució final: Artifactory

La instal·lació d'Artifactory és bastant directe: Només cal fer servir tota l'arquitectura que s'ha dissenyat a través de Docker. Així doncs, el primer pas és descarregar-se a través de la pàgina web la imatge ja empaquetada que conté tota la lògica del servidor i integrar-la dins del nostre entorn. Després d'assignar una IP dins del clúster local ja es pot començar amb el procés de configuració:



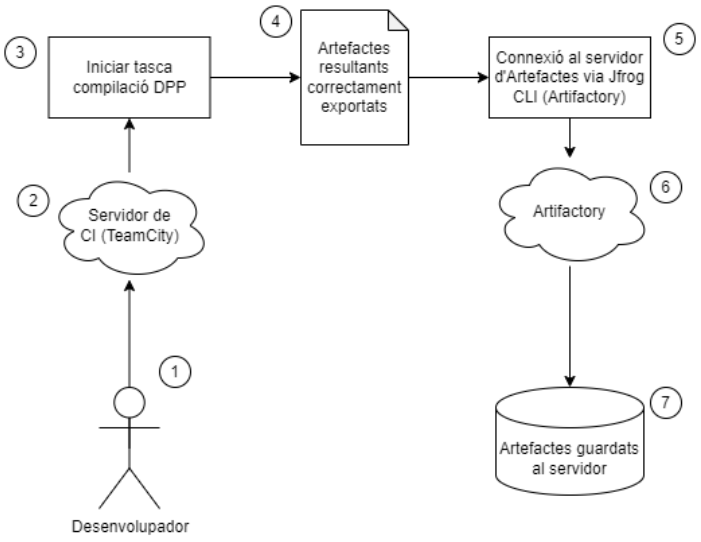
Imatge 29: Primers passos dins del servidor d'Artifactory

En aquest entorn, un repositori és un conjunt d'artefactes que conformen l'aplicació principal o bé formen part del seu arbre de dependències. Es poden separar aquests artefactes en dos grups principals: Desenvolupament i producció. De moment només es vol valorar si aquesta eina es compatible amb els requeriments de la pipeline així que es realitzarà una petita prova pujant uns arxius generats pels binaris de Discord Plus Plus.

Artifactory simplifica l'automatització en la descàrrega i pujada d'arxius amb una petita eina anomenada Jfrog CLI. A través de comandes especificades en la documentació es permet enviar paquets d'arxius i carpetes amb una sola comanda, sense necessitat d'escriure cap iteració. A més a més està integrat automàticament per a servidors com Artifactory. Un exemple de la seva sintaxis seria: '`jf rt dl my-local-repo/cool-froggy.zip`' on "jf" és el binari principal, "rt dl" és una abreviació de "Artifactory download" i "my-local-repo/cool-froggy.zip" és el fitxer que volem descarregar del

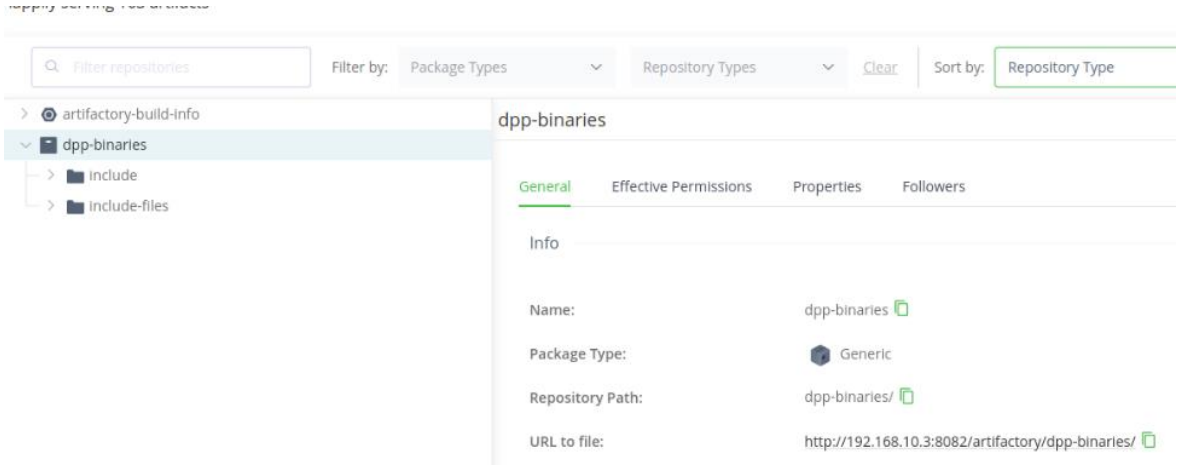
servidor. Com no s'especifica carpeta destí, la comanda posarà els continguts en la carpeta actual.

Amb tot això el pla inicial per fer la prova amb Artifactory és el següent:



Imatge 30: Esquema per a la prova de la primera pipeline amb Artifactory

Per fer la proposta esquematitzada en la Imatge 30 cal primer modificar el Dockerfile per afegir els passos extres 4 i 5. En aquest moment només es comprovava que els arxius es compilessin correctament sense cap tractament posterior dels resultats. En la nova iteració s'ha afegit un pas més que guarda tots els artefactes de la build en una carpeta i a posteriori el client de jfrog els envia en un sol paquet al servidor. Després de resoldre alguns problemes sobre la sintaxis de la comanda (cal ser molt rigorós amb els noms de les carpetes), el resultat és el següent:



Imatge 31: Dockerfile modificat i resultat final dins del servidor d'artefactes

Destacar que cal tenir en compte el nom de les carpetes on es troben els diferents artefactes tal i com marca el conveni assignat per Ubuntu. Per exemple en la carpeta "include" es troben tots els .h públics, en la carpeta "lib" es troba el codi compilat de les llibreries en format estàtic o compartit. D'aquesta forma els propers desenvolupadors que desitgin crear automatitzacions amb la llibreria en qüestió ho tindran molt més fàcil per entendre la seva organització.

7.6 Sprint 7 i 8: Integració total al sistema de CI

Ara que ja s'ha completat tota la pipeline per a un sol binari i s'ha comprovat que funciona, es pot començar amb la producció del contingut final.

Primer, s'han integrat totes les dependències necessàries pel projecte (explicades i enumerades en el Sprint 1). Això implica automatitzar cada un dels processos d'instal·lació i descàrrega del codi a més de fer front a possibles requeriments extres que sorgeixin. Tant Catch2 com DPP no han presentat problemes inesperats i la seva integració ha sigut bastant directe. En el cas de MongoCxx s'ha hagut també d'automatitzar el procés de build per a la seva dependència: MongoC. D'aquesta forma es té un control total sobre la compilació del driver. El problema més destacable ha sorgit a través del tractament d'una de les llibreries més antigues: LibGD.

LibGD està principalment pensada per a ser utilitzada en un entorn purament escrit en C. Això significa que la API espera que el desenvolupador tingui un control extern de la càrrega i descàrrega de memòria a més de treballar amb punters natius del

llenguatge. C++ introdueix diferents utilitats que embolcallen aquesta complexitat, com els destructors o els punters únics. Els desenvolupadors de libGD per sort han creat una API en C++ que conté aquestes facilitats. Malauradament al activar aquesta funcionalitat a través del CMake s'ha generat el següent error:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

main.cpp:(.text._ZN12ostreamIOCtx4initEPSo[_ZN12ostreamIOCtx4initEPSo]+0x17): undefined reference to `ostreamIOCtx::Getchar
(gdIOCtx*)'
/usr/bin/ld: main.cpp:(.text._ZN12ostreamIOCtx4initEPSo[_ZN12ostreamIOCtx4initEPSo]+0x25): undefined reference to `ostreamI
OCtx::Putchar(gdIOCtx*, int)'
/usr/bin/ld: main.cpp:(.text._ZN12ostreamIOCtx4initEPSo[_ZN12ostreamIOCtx4initEPSo]+0x34): undefined reference to `ostreamI
OCtx::Getbuf(gdIOCtx*, void*, int)'
/usr/bin/ld: main.cpp:(.text._ZN12ostreamIOCtx4initEPSo[_ZN12ostreamIOCtx4initEPSo]+0x43): undefined reference to `ostreamI
OCtx::Putbuf(gdIOCtx*, void const*, int)'
/usr/bin/ld: main.cpp:(.text._ZN12ostreamIOCtx4initEPSo[_ZN12ostreamIOCtx4initEPSo]+0x52): undefined reference to `ostreamI
OCtx::Tell(gdIOCtx*)'
/usr/bin/ld: main.cpp:(.text._ZN12ostreamIOCtx4initEPSo[_ZN12ostreamIOCtx4initEPSo]+0x61): undefined reference to `ostreamI
OCtx::Seek(gdIOCtx*, int)'
/usr/bin/ld: main.cpp:(.text._ZN12ostreamIOCtx4initEPSo[_ZN12ostreamIOCtx4initEPSo]+0x70): undefined reference to `ostreamI
OCtx::FreeCtx(gdIOCtx*)'
collect2: error: ld returned 1 exit status
make[2]: *** [CMakeFiles/gd_test.dir/build.make:98: gd_test] Error 1
make[1]: *** [CMakeFiles/Makefile2:83: CMakeFiles/gd_test.dir/all] Error 2
make: *** [Makefile:91: all] Error 2
```

Imatge 32: Error al generar els artefactes per a la llibreria de libGD

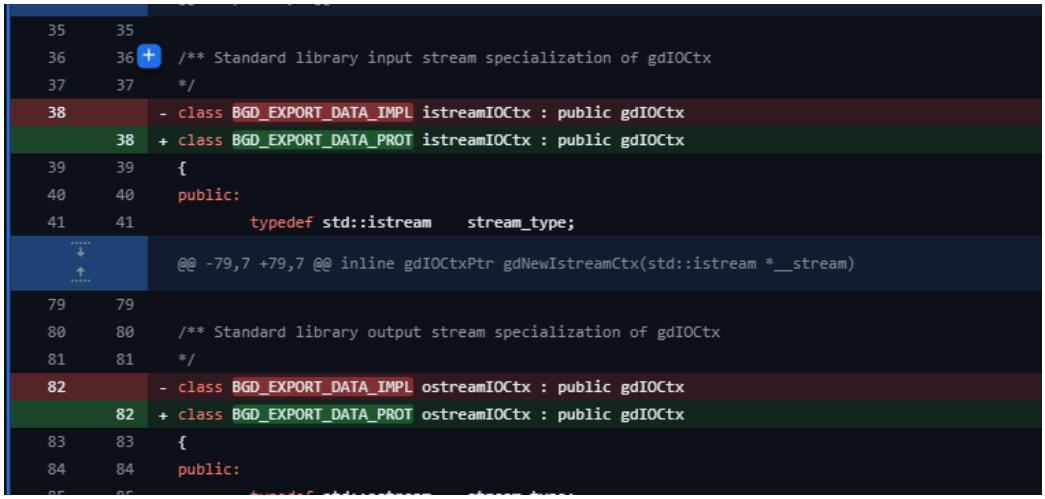
En aquest punt és important recordar la teoria que s'ha esmentat en el marc teòric sobre el procés de compilació d'un artefacte en C++. Com es pot veure en la imatge, l'error està generat per la comanda "ld" els quals ha retornat un valor de "1". Això dona a entendre que el problema és específicament d'enllaçament: el linker no és capaç de trobar les implementacions d'un conjunt de funcions. Després d'investigar una estona sobre la possible causa de l'error, en el CMake de la llibreria es troba el següent:

```
64  if (BUILD_SHARED_LIBS)
65      add_library(${GD_LIB} ${LIBGD_SRC_FILES})
66      set_target_properties(${GD_LIB} PROPERTIES
67          SOVERSION ${GDLIB_LIB_SOVERSION}
68          VERSION ${GDLIB_LIB_VERSION}
69          C_VISIBILITY_PRESET hidden
70          CXX_VISIBILITY_PRESET hidden
71      )
72  endif()
```

Imatge 33: CMake de la llibreria LibGD, en aquest cas s'està afegint un target nou i aplicant una sèrie de propietats.

Com es pot veure en la línia seleccionada, per defecte s'està ofuscant les classes i implementacions del llenguatge. Això significa que el linker no pot enllaçar res que no sigui expressament marcat com a públic. Sabent això, la solució passa a ser directa:

Canviant la definició de la classe i marcant explícitament que es vol considerar pública, el linker ja pot realitzar les operacions pertinents. Juntament amb altres canvis més petits, la llibreria libGD ja està completament preparada per a ser automatitzada.

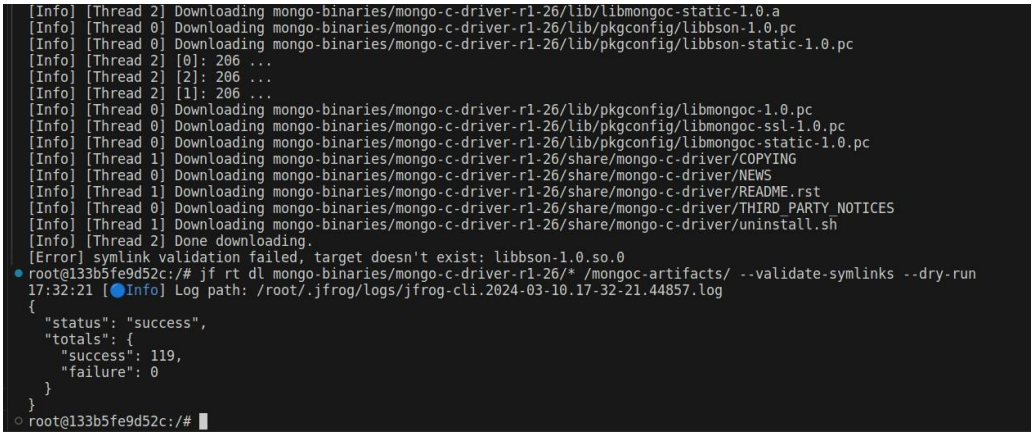


Imatge 34: Canvi realitzat a la llibreria per solucionar el problema d'enllaçament

Un altre problema que s'ha hagut de resoldre ha sigut amb la pujada d'arxius de tipus enllaç simbòlic. Bàsicament son arxius de Linux que apunten a un altre arxiu en el sistema. Quan s'està desenvolupant una llibreria amb un sistema de versions, l'arxiu final sovint se li concatena el número de la versió (llibreria-1.2.3.so). Aquests canvis de noms poden causar problemes als sistemes que s'encarreguen de validar dependències. Per aquest motiu, es crea un punter (symlink) a la llibreria amb un nom constant (llibreria-1.2.3.so -> llibreria.so).

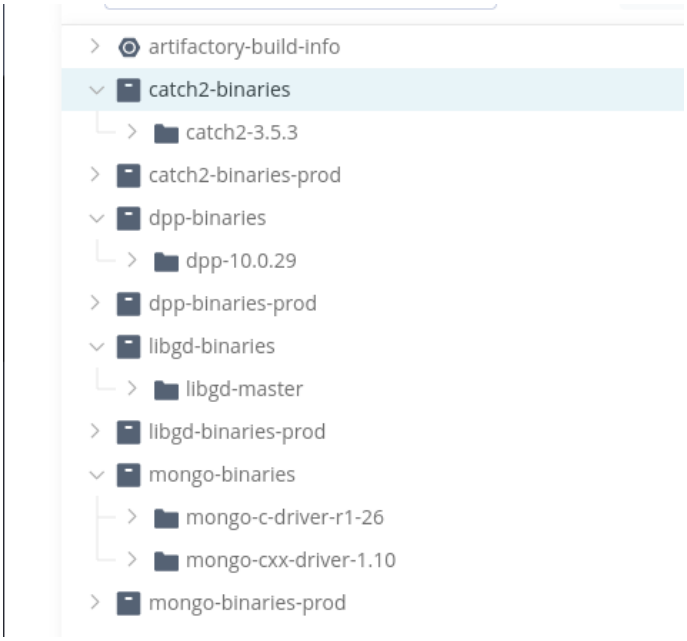
Artifactory no valida de forma automàtica si s'està pujant un arxiu de tipus symlink. Això és una conseqüència de que l'entorn en què treballa ha de ser agnòstic del sistema operatiu. Aquesta propietat s'aconsegueix afegint metadades als arxius pujats, fent que al demanar una descàrrega només s'hagi de reconstruir el fitxer amb el format adient.

Donat que Artifactory no activa la propietat de guardar symlinks de forma automàtica, al pujar-ne un s'ha d'especificar el seu ús afegint la propietat "--symlinks". A continuació es pot comprovar que els punters estan ben configurats si no salta cap error al descarregar els binaris amb la propietat "--validate-symlinks"



Imatge 35: Llibreria mongo-c-driver enviada de forma satisfactòria al servidor

Amb aquest problema resolt finalment ja es tenen totes les dependències enregistrades i automatitzades al servidor de TeamCity on, de forma automatitzada, es sincronitzen els artefactes a Artifactory. A continuació es mostren algunes imatges del procés acabat:

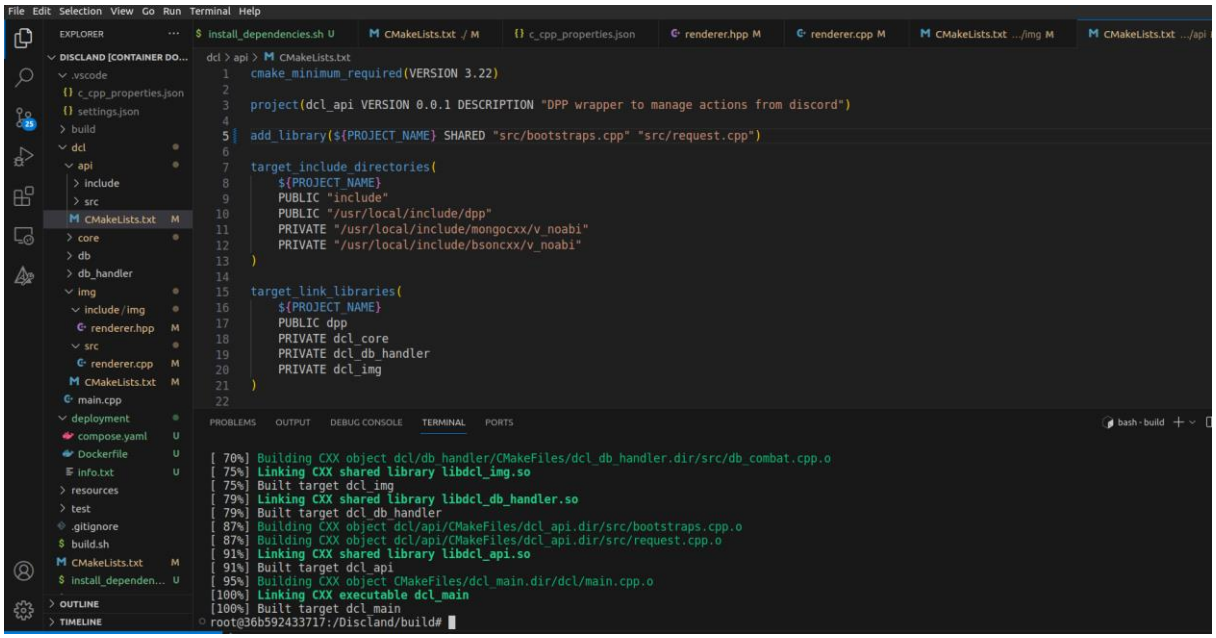


Imatge 36: Repositori dins d'artifactory amb totes les dependències separades per producció i desenvolupament pròpiament versionades i classificades.

Cal destacar que s'han generat dos processos diferents per a cada dependència. En un es compila el codi amb una configuració de debug, pensat principalment pels desenvolupadors per a que puguin veure millor el seguiment del codi sense les

optimitzacions pertinents. En l'altre, la configuració és de producció i es compila el codi amb la configuració de "Release" per a que estigui més optimitzat i vagi més ràpid.

Per acabar el sistema de CI només cal integrar-ho amb l'aplicació final: l'executable que presenta totes aquestes dependències. Per fer-ho cal realitzar el procediment invers. En comptes de pujar els artefactes al servidor, el nou fitxer de Dockerfile els ha de descarregar al seu lloc pertinent. En aquest cas s'ha decidit utilitzar el lloc per defecte d'Ubuntu "usr/local/". En el CMake de la imatge a continuació es pot veure com s'agafen les dependències a través de la carpeta "include", ja que s'ha respectat el conveni que s'ha explicat anteriorment.



Imatge 37: El programa compila satisfactòriament amb totes les dependències adquirides a través del servidor d'artefactes de forma automàtica.

Per acabar amb l'explicació d'aquest sprint es veu convenient adjuntar alguns fitxers importants que s'han desenvolupat. En els annexos es pot trobar tot el conjunt de scripts i Dockerfiles del projecte.

Per començar, el cas concret que es presenta a continuació automatitza el procediment per a la llibreria de MongoCXX que presenta una dependència amb MongoC.

```
FROM ubuntu:24.04

ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install --no-install-recommends -y
libssl-dev ca-certificates cmake curl pkg-config g++ gcc git make &&
apt-get clean && rm -rf /var/lib/apt/lists/*

WORKDIR /usr/src/mongocxx

COPY . .

RUN curl -fL https://install-cli.jfrog.io | sh
RUN jf c add local --url=http://192.168.10.3:8081 --user=admin --
password=Jfrog_1234 --interactive=false

WORKDIR /usr/src/mongocxx/build

RUN jf rt dl mongo-binaries/mongo-c-driver-r1-26/*
/mongo_c_artifacts/

RUN cmake -DCMAKE_BUILD_TYPE=Release -
DCMAKE_INSTALL_PREFIX=/artifacts -DCMAKE_CXX_STANDARD=20 -
DCMAKE_PREFIX_PATH=/mongo_c_artifacts/mongo-c-driver-r1-26 -
DBUILD_VERSION="3.10.x" ..
RUN make && make install

RUN jf rt del mongo-binaries-prod/mongo-cxx-driver-1.10/ --quiet
WORKDIR /artifacts
RUN jf rt upload "./*" mongo-binaries-prod/mongo-cxx-driver-1.10/ --
symlinks
```

La primera part (les tres accions inicials) consisteix en la configuració del sistema operatiu: S'especifica quina versió d'Ubuntu es vol utilitzar i que no presenta terminal interactiva a l'hora de realitzar les comandes (ho fa un procés automàtic). Finalment s'especifiquen tots els paquets essencials per a poder realitzar la feina.

A continuació es descarreguen les dependències necessàries del servidor d'artefactes, en aquest cas mongoc.

Finalment es compila la llibreria especificant la localització de les dependències descarregades. També s'especifica la carpeta on es s'han de posar els fitxes resultants per a que es puguin ser pujats al servidor d'Artifactory de forma satisfactòria. Cal parar atenció que aquest Dockerfile s'utilitza per a la versió de producció, ja que la comanda de CMake esta configurada per a que utilitzi totes les optimitzacions possibles.

Un altre fitxer important que cal explicar conforma el procés d'instal·lació de les dependències en l'aplicació final:

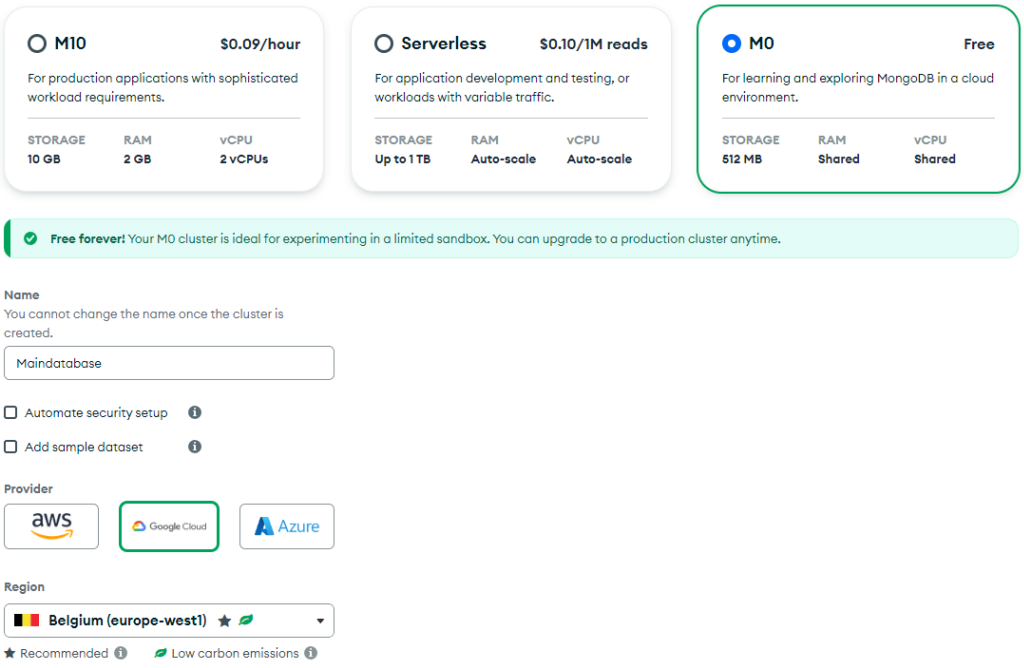
```
function navigate_and_rsync {
    local dir="$1"
    cd "$dir" || exit

    for subdir in */; do
        if [ -d "$subdir" ]; then
            echo "Entering $subdir"
            cd "$subdir" || exit
            # Execute rsync command
            rsync -av --ignore-existing ./* /usr/local/
            cd ..
            echo "Exited $subdir"
            rm -rd "$subdir"
        fi
    done
}

starting_path="/dependencies"

if [ -d "$starting_path" ]; then
    echo "Starting navigation from: $starting_path"
    navigate_and_rsync "$starting_path"
else
    echo "Invalid starting path: $starting_path"
fi
```

El seu principal objectiu és agafar tots aquells fitxers descarregats de forma automàtica en la carpeta /dependències i moure'ls al seu lloc corresponent per a que les eines de compilació i enllaçament puguin detectar-los.

7.7 Sprint 9: MongoDB i deployment

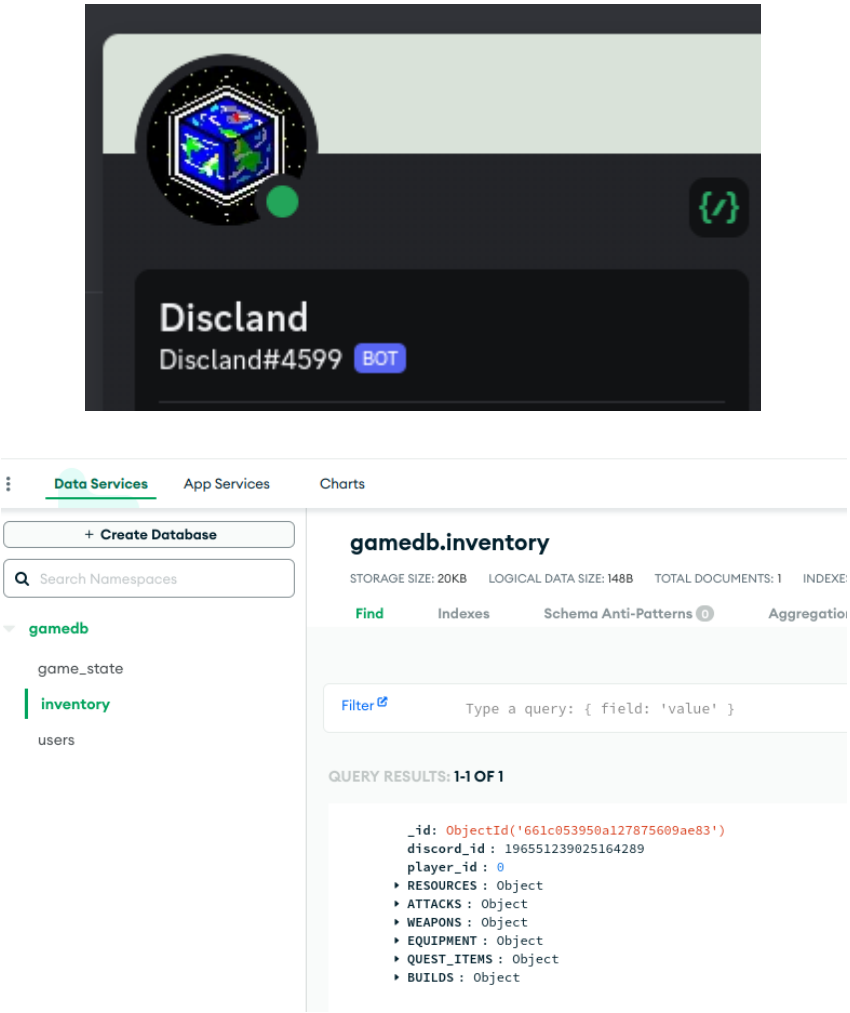
L'aplicació necessita un clúster d'una base de dades MongoDB a la que es pugui connectar per funcionar i guardar la informació dels jugadors. Anteriorment, aquesta instància s'executava de forma local a la màquina principal. Dins d'una visió on es vol automatitzar el procés de desplegament això pot portar problemes i configuracions extres que a la llarga es poden complicar.

Amb aquesta premissa en ment, s'ha desplaçat la instància de la base de dades a un servei al núvol que dona el propi MongoDB de forma gratuïta. D'aquesta forma només cal configurar i restringir l'accés al servidor de totes aquelles màquines que no tinguin la mateixa IP del servidor on s'està executant el joc.



Imatge 38: Procés de registre d'una instància al servidor de MongoDB

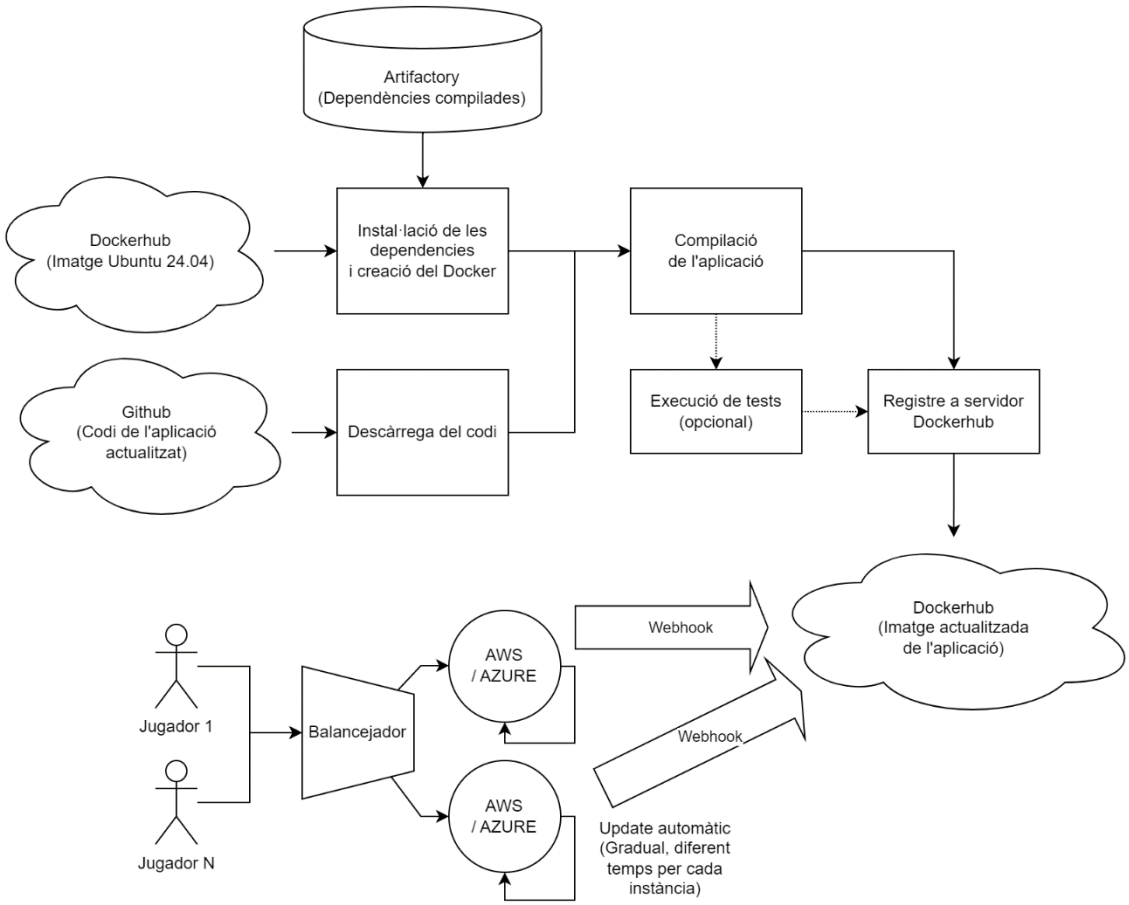
Amb un parell de configuracions extres i després d'executar l'aplicació compilada, es pot veure com es connecta satisfactòriament a la base de dades i el seu funcionament torna a ser el normal:



Imatge 39: (A dalt) Bot connectat al servei de Discord. (A baix) Enviament de dades del usuari al nou clúster de MongoDB

Ara que finalment està tot lligat en aquest primer apartat sobre la integració continua és el moment de seguir amb la pipeline i passar al desplegament continu. Per fer-ho primer s'ha de dissenyar què es vol donar a la màquina que executarà l'aplicació i com es pot automatitzar el procés. Amb això, es pot seguir la tendència del treball a utilitzar Docker per guardar un paquet compilat i llest pel servidor final en qualsevol moment. La idea és utilitzar un repositori privat dintre de DockerHub on el servidor (o servidors)

podrà descarregar la imatge i executar-la directament sense cap mena de dificultat. L'esquema d'aquesta interacció queda de la següent forma:

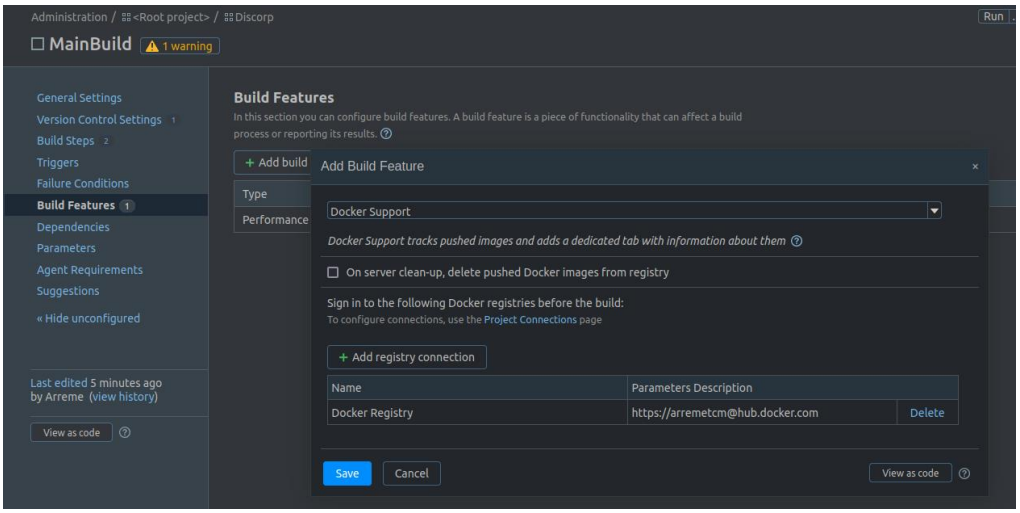


Imatge 40: Arquitectura de la integració continua i disseny de les seves fases.

Tot i que està dissenyada l'arquitectura de la infraestructura dels servidors (Imatge 40), aquesta fase està fora de l'abast del treball. La idea principal és poder assegurar en tot moment que hi ha una aplicació actualitzada i comprovada al Dockerhub.

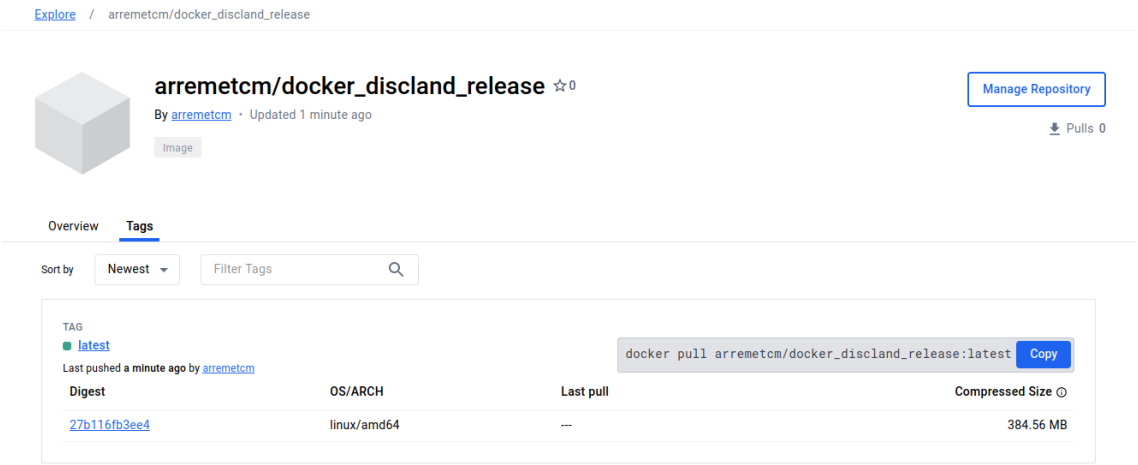
Així doncs, a través del que es veu en el pla de desplegament, el primer pas és configurar TeamCity per a que pugui pujar imatges al repositori privat de Dockerhub. Per fer-ho primer cal que els servidor tingui l'accés al usuari, ja que la imatge que es pujarà pot contenir informació sensible com ara codi font o donar traces de possibles vulnerabilitats. Després d'afegir el compte, s'ha d'afegir un pas extra a la pipeline d'integració continua per a que pugi de forma automàtica la imatge al repositori privat.

Això es pot fer a través de la comanda “docker push <nom:tag>”. També cal afegir dins d’aquest pas quin compte s’ha de fer servir per iniciar sessió.



Imatge 41: Configuració del compte per a que pugui pujar la imatge al repositori privat.

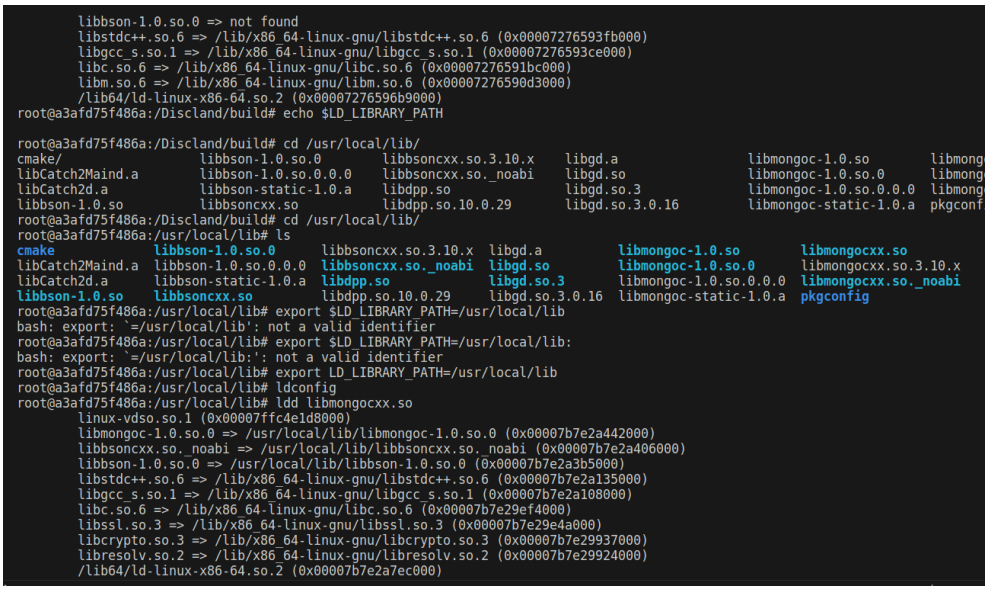
Amb un parell de retocs extres i acabar de solucionar algun problema sobre la pipeline finalment s’aconsegueix l’objectiu esperat:



Imatge 42: Imatge de l’aplicació llesta per a ser descarregada pels servidors

Finalment, s’ha hagut de fer front a un petit problema pel que fa les llibreries compartides a l’hora de fer el salt a Docker. Per algun motiu el sistema operatiu era

incapaç de trobar els arxius ".so" d'algunes dependències a pesar de trobar-se en el mateix lloc que abans. El problema s'ha resolt reindexant el buscador de les llibreries compartides a través de la comanda ldconfig.



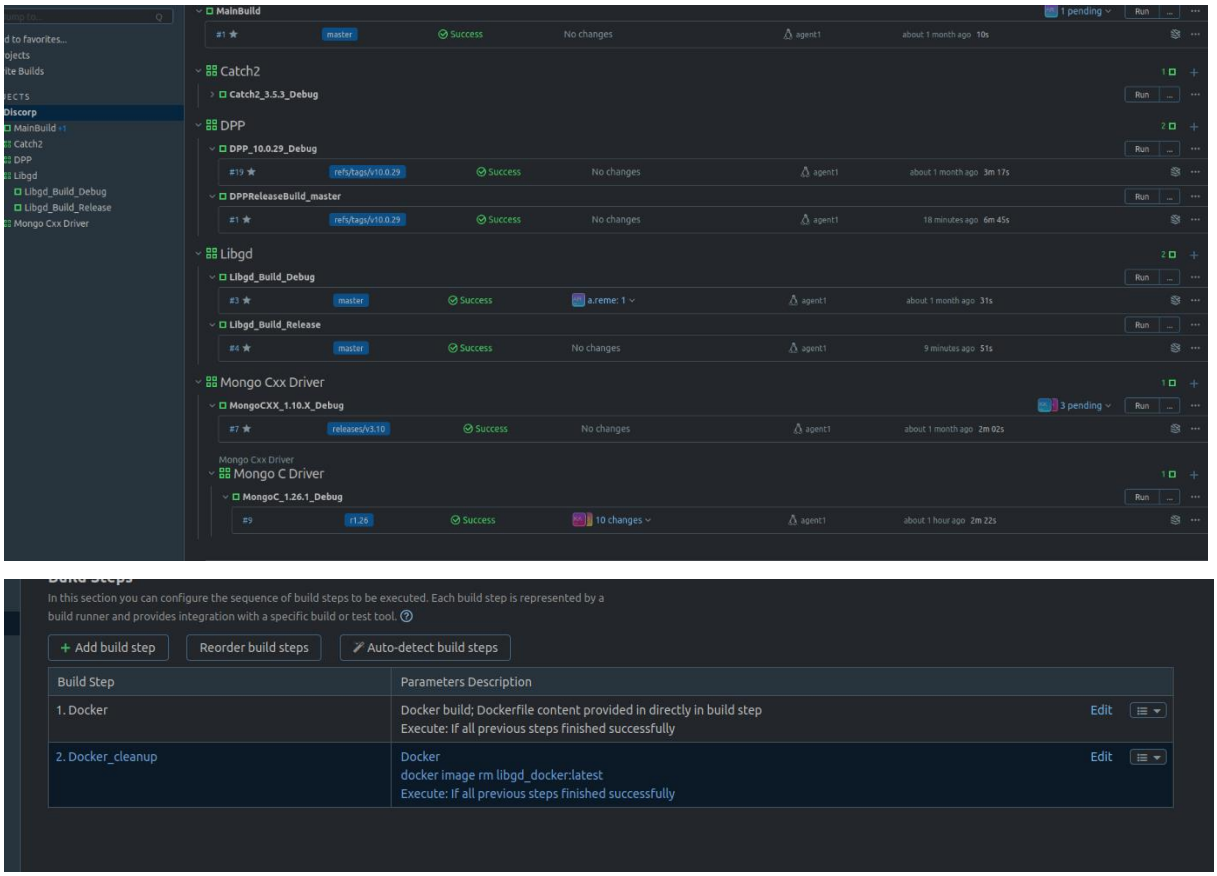
Imatge 43: En la primera línia no es troba la llibreria compartida. Després de reindexar-les el buscador ja és capaç de trobar-les sense cap problema.

7.8 Sprint 10: Refinament i testing

Aquest últim sprint es va deixar sense objectius per si alguna de les fases anteriors necessitaven més temps del previst. Com que la pipeline s'ha realitzat al complet aquestes dues setmanes es destinaran refinar el que ja es té i a implementar un petit sistema de testing unitari per a comprovar la lògica de l'aplicació.

En el cas del refinament s'han realitzat principalment dues tasques importants. La primera ha sigut ordenar els noms de totes les diferents pipelines dins del servidor de TeamCity i executar les seves últimes actualitzacions. També s'ha establert com han de ser els noms dels diferents artefactes. La segona tasca s'ha centrat principalment en la implementació d'un sistema de neteja automàtic dels artefactes que es creen. És una forma bastant eficient per estalviar espai. D'aquesta manera no cal realitzar rutinàriament una neteja dels Dockers que TeamCity va creant. Per fer-ho s'ha creat

un pas extra a cada una de les diferents pipelines d'integració continua i s'ha configurat que ho esborrin top un cop s'ha pujat el necessari al servidor d'Artifactory.

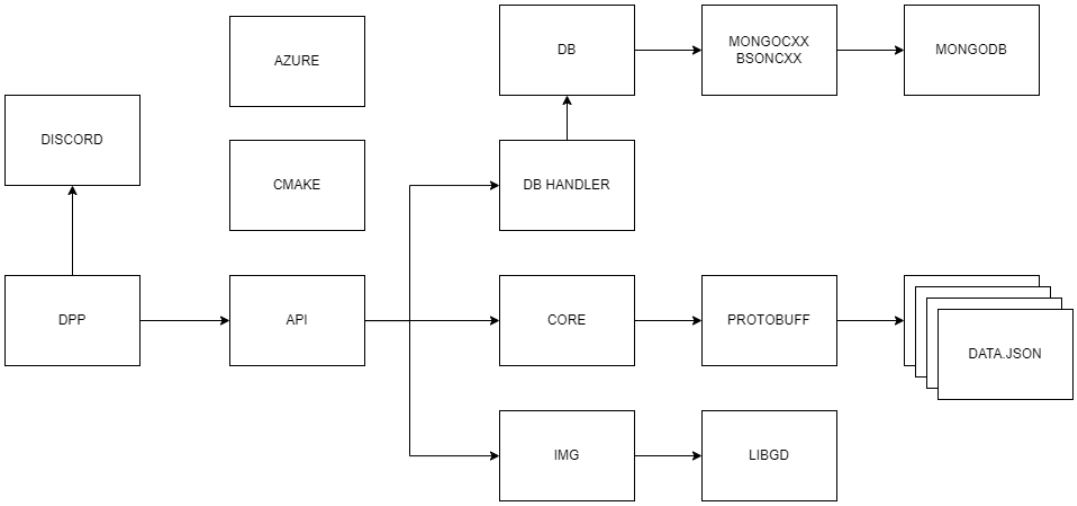


Imatge 44: (A dalt) Estructuració dels noms de les diferents pipelines, resultat de la primera tasca. (A baix) Neteja dels dockers i altres recursos, resultat de la segona tasca.

Per a la creació de tests s'ha intentat respectar l'arquitectura de l'aplicació ja existent on els diferents mòduls estan ben diferenciats i separats per llibreries internes, cada una amb la seva responsabilitat. Segons es pot veure en la Imatge 45, les diferents llibreries es categoritzen de la següent forma:

- Core: Part central de l'aplicació que a partir de protobuffers gestiona els models de dades i la lògica del joc. És la capa que té menys dependències ja que la idea central és que es pugui testejar de forma simple.
- DB: És la interfície que separa el driver de la base de dades amb la resta de l'aplicació. No hi ha lògica de joc, només gestiona la comunicació.

- DB Handler: Crea i gestiona els diferents DTOs (Data transfer objects) de l'aplicació amb la capa de comunicació que s'ha creat. També funciona com a DAO (Data access object) ja que adapta i transforma les diferents accions de l'aplicació amb crides a la base de dades.
- Img: Utilitza la llibreria libgd per a crear imatges a partir de les dades de l'usuari. És la capa de "visualització".
- Api: Implementa les crides i la comunicació que venen a través de Discord. De la mateixa forma que la capa més externa d'una aplicació REST, només defineix els endpoints i què és el que han d'executar en quant a lògica.



Imatge 45: Arquitectura interna de DiscLand, l'aplicació que s'està automatitzant

Aquesta separació permet a aquells que hagin de crear tests unitaris poder enviar dades falses (en anglès "mocks") per a posar a prova la lògica de l'aplicació.

En aquesta part és on entra en joc una dependència que fins ara no ha sortit: Catch2. Permet la creació de tests unitaris per a C++. La metodologia que es farà servir en el treball és la de triple A o (Arrange, Act, Assert). L'exemple de la següent imatge és una demostració d'aquest sistema: Primer s'assegura una comunicació amb la base de dades de proves i s'esborra tota la informació que pot malmetre la replicació del test. Després es comencen a crear les classes per a poder inserir la informació amb la que es vol fer el test. A continuació s'executa la lògica. En aquest cas donat que el test pertany a la llibreria db_handler, s'està comprovant si funciona la inserció d'un

nou usuari. Finalment les clàusules “REQUIRE” s’asseguren que la lògica s’ha completat com s’ha previst.

```cpp
TEST_CASE("User Database Handler Testing","[db_handler][db_handler_test_1]")
{
    //Arrange
    db::DeleteManyOperation del_op_usrs = db::DeleteManyOperation{"users", make_document()};
    del_op_usrs.ExecuteOperation();
    PBUser user;
    user.set_discord_id(0);
    user.set_user_name("Arreme");
    user.set_current_player_id(0);
    auto last_online = user.mutable_last_online();
    auto now = std::chrono::system_clock::now();
    auto seconds = std::chrono::duration_cast<std::chrono::seconds>(now.time_since_epoch()).count();
    last_online->CopyFrom(google::protobuf::util::TimeUtil::SecondsToTimestamp(seconds));
    auto player = user.add_players();
    player->set_version("0.0.1");
    player->set_guild_id(10);
    player->set_current_location(PBLocationID::MAIN_BASE);
    {
        auto equip = player->mutable_equipation();
        auto skills = player->mutable_skills();
        skills->set_foraging_lvl(1);
        equip->add_attacks(PBItemEnum::KICK);
    }
    db_handler::DBUserHandler db_user_handler{&user};
    //Act
    REQUIRE(db_user_handler.InsertUser());
    player->set_guild_id(230);
    //Assert
    REQUIRE(db_user_handler.FindUserCurrentPlayer());
    REQUIRE(player->guild_id()==10);
    REQUIRE(player->equipation().attacks_size() == 1);
    REQUIRE(player->equipation().attacks(0) == PBItemEnum::KICK);
    REQUIRE(player->version() == "0.0.1");
};
```

Imatge 46: Exemple d’un test unitari dins l’aplicació DiscLand

```
-------------------------------------------------------------------------------
Collecting a post
-------------------------------------------------------------------------------
/Discland/test/core_test.cpp:23
...............................................................................

/Discland/test/core_test.cpp:23: FAILED:
due to unexpected exception with message:
  filesystem error: directory iterator cannot open directory: No such file or
  directory [resources/data/locations]

-------------------------------------------------------------------------------
Scenario: Testing database basic operations
     Given: 2.- A collection with a document
      When: 2.1.- Inserting with the same id
      Then: Result should be duplicated id
-------------------------------------------------------------------------------
/Discland/test/db_test.cpp:78
...............................................................................

/Discland/test/db_test.cpp:80: FAILED:
  REQUIRE( ins_op_2.GetState() == db::OperationState::DUPLICATED_ID )
with expansion:
  1 == -4

===============================================================================
test cases: 14 |  7 passed | 7 failed
assertions: 49 | 42 passed | 7 failed
```

Imatge 47: Execució dels tests. Es pot veure que hi ha errors pel que el programador haurà de revisar el seu codi amb més atenció.

8 Conclusions

El treball ha acabat amb tots els objectius principals completats llevat del desenvolupament de tècniques de monitorització avançada. Tot i així, s'ha dedicat un esforç addicional a entendre les diferents eines actuals que donen solucions en la creació d'un servidor d'integració continua. També s'ha tractat amb molta profunditat tot el procés de creació d'una pipeline orientada a una aplicació en C++, des del moment de la seva ideació fins al seu desplegament.

Un dels aprenentatges més importants que es pot extrapolar a una gran quantitat de projectes ha estat la concepció de Docker com a eina principal per aïllar processos. Molts cops no tenir un control sobre l'ambient en que es treballa causa problemes de replicació i de funcionament. A més, tal i com s'ha estat revistant en la part teòrica, llenguatges que no presenten independència amb el sistema son molt susceptibles a canvis en els dispositiu on s'executen. Docker ha simplificat notablement aquestes operacions i ha estat una excel·lent decisió situar-lo com a eix central del projecte. No només permet aquesta replicació del codi sinó que obliga a l'usuari a ser més conscient de les dependències que es necessiten, ja que s'han d'escriure de forma explícita.

Lligat al punt anterior, és important remarcar la consistència com a qualitat central per a la creació i millora de processos. Agrupar diferents metodologies de funcionament a través d'un sol canal ha estat la part més difícil del treball, ja que les llibreries tenen la seva forma pròpia de compilar-se i d'automatitzar-se. Garantir una interfície consistent i determinista aporta un gran valor a tots els agents que l'utilitzen en el seu dia a dia, facilitant la seva feina i embolcallant la dificultat a través de simples scripts.

Però la consistència es pot aconseguir en un entorn on competeixen una gran infinitat d'eines que resolen el mateix problema de formes diferents? Aquesta era una de les principals preocupacions que el treball intentava solucionar en el seu inici. Com s'ha pogut veure en la part teòrica si ve és cert que les eines per a C++ són molt variades, amb la part pràctica la pregunta queda totalment resolta. La flexibilitat del llenguatge permet als desenvolupadors de processos adaptar-se als requeriments canviants de les eines. Cada projecte en aquest sentit és únic però a la vegada consistent en si

mateix. El fet que hi hagi tantes eines emfatitza la virtut del propi llenguatge a ser modificat segons els requeriments de cada projecte.

El que també s'ha vist durant la part pràctica és que l'ús del sistema operatiu Ubuntu dona molt de joc a la flexibilitat pròpia de C++. És bastant senzill poder modificar el funcionament bàsic del sistema afegint o modificant les peces requerides del programa que s'està instal·lant. A part simplifica molt el procés d'empaquetament de les aplicacions per poder exportar els binaris a altres dispositius. Amb això s'acaba concloent que ha estat una decisió encertada treballar amb Ubuntu i es recomana el seu ús per a feines similars.

Tot i així, a pesar que el treball ha estat finalitzat amb els temps preestablerts, és important remarcar alguns punts que podrien ser millorats o bé tractats en futures investigacions i que podrien aportar més coneixement sobre el tema:

- **Ús de Conan dins la pipeline d'integració continua:** Conan és un controlador de paquets per aplicacions basades en C i C++. A partir de scripts escrits amb Python es pot configurar quines dependències s'han de descarregar i com es pot instal·lar la llibreria que s'està desenvolupant en altres dispositius. D'aquesta forma cada desenvolupador es responsable d'escriure els passos adequats i mantenir el seu codi compatible amb altres paquets. Conan està poc a poc agafant força pel que seria un bon punt a investigar en futures anàlisis.
- **Eines de monitorització amb panells d'informació:** Estudiar com parametritzar una aplicació en C++, quines eines servirien i com mostrar la informació generada als actors pertinents. Per exemple com generar un informe d'eficiència on es mostrin els punts crítics de l'aplicació o tenir un panell amb els errors que han saltat per a que els desenvolupadors puguin solucionar-los. Un altre cas pot ser la monitorització de l'usuari per saber possibles punts de millora en el disseny del joc.
- **Estudi amb un altre servidor d'integració continua:** El treball s'ha centrat en estudiar l'ús de TeamCity com a eina principal d'integració continua. Jenkins ha estat descartat per problemes tècnics però futurs treballs en la mateixa línia d'investigació poden tornar a examinar-lo. D'altre forma, també es poden realitzar comparatives amb eines que no s'han pogut tocar en el treball com

CircleCI, Github Actions o Gitlab. Aquesta investigació pot ser interessant ja que es pot veure com diferents eines han resolt el mateix problema.

Amb tot això exposat, val la pena remarcar que el treball s'ha concebut des d'una perspectiva més d'investigació que no pas de producte. Si no hagués estat així moltes de les praxis que s'han estudiat no s'haurien acabat de programar ja que els objectius haurien sigut diferents. De fet, centrar-se en una perspectiva més teòrica ha resultat en una gran facilitat per a resoldre els problemes que han sorgit en les fases finals.

La investigació acaba manifestant que els responsables del disseny i la implementació d'aquestes estructures han de tenir un coneixement completament transversal per fer front a les subtileses del dia a dia. Ser enginyer implica resoldre problemes per a millorar la qualitat de vida de les persones que els envolten i això només es pot aconseguir harmonitzant el coneixement teòric amb la pràctica i la disciplina.

9 Bibliografía

[1] D. Fonović y T. Galinac Grbac, «A Quantitative Study of C/C++ FOSS Software Buildability,» *CEUR-WS,* vol. 3237, nº 3, pp. 3:1 - 3:10, 9 Oct 2022.

[2] Á. Kiss, «An explorative analysis of managed CI/CD usage among open-source C/C++ projects,» *Production Systems and Information Engineering,* vol. 10, nº 3, pp. 19-30, 27 Sep 2022.

[3] S. Shajarian, *The C++ programming language in modern computer science (Master's Thesis),* Tampere: Tampere University Press, 2020.

[4] A. Miranda y J. Pimentel, «On the Use of Package Managers by the C++ Open-Source Community,» de *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, Pau, Association for Computing Machinery, 2018, p. 1483–1491.

[5] A. Reig Méndez, «Tecnocampus Digital Repository,» 18 Jun 2023. [En línea]. Available: http://hdl.handle.net/20.500.12367/2353. [Último acceso: 26 Dec 2023].

[6] SFML, «SFML: Home,» 2023. [En línea]. Available: https://www.sfml-dev.org/. [Último acceso: 12 12 2023].

[7] C. Rossi, «Engineering at Meta,» Meta, 31 Aug 2017. [En línea]. Available: https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/. [Último acceso: 5 Jan 2024].

[8] C. Edwards, «D++: A C++ Discord API Library for Bots,» [En línea]. Available: https://dpp.dev/. [Último acceso: 16 Dec 2023].

[9] K. Schwaber y J. Sutherland, «The 2020 Scrum Guide,» Scrum Guides, 2020. [En línea]. Available: https://scrumguides.org/scrum-guide.html. [Último acceso: 16 Dec 2023].

[10] Indeed Editorial Team, «What Is a High-Level Project Plan? Definition and Importance,» Indeed, 27 Jan 2023. [En línea]. Available: https://www.indeed.com/career-advice/career-development/high-level-project-plan. [Último acceso: 12 Jan 2024].

[11] B. Stroustrup, The C++ programming language, Nova Jersey: Addison-Wesley, 2015.

[12] B. Stroustrup, The design and evolution of C++, Addison-Wesley Professional, 1994.

[13] IBM, «Function declarations and definitions,» IBM, 26 Jan 2024. [En línea]. Available: https://www.ibm.com/docs/en/xl-c-and-cpp-aix/16.1?topic=functions-function-declarations-definitions. [Último acceso: 31 Jan 2024].

[14] Microsoft, «Header files (C++),» Microsoft, 8 Mar 2021. [En línea]. Available: https://learn.microsoft.com/en-us/cpp/cpp/header-files-cpp?view=msvc-170. [Último acceso: 7 Feb 2024].

[15] Kitware, «CMake.org,» Kitware, 2023. [En línea]. Available: https://cmake.org/. [Último acceso: 8 Feb 2024].

[16] A. F. A. H. R. S. H. D. V. D. H. A. &. W. A. L. Carzaniga, «A characterization framework for software deployment technologies,» *Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado,* vol. 80, 1998.

[17] L. A. C. P. L. T. A. E. W. Daniel J. Barret, «A Framework for Event-Based Software,» *ACM Transactions on Software Engineering and Methodology,* vol. 5, nº 4, pp. 378-421, 19956.

[18] J. Humble y D. Farley, Continuous Delivery, Boston: Pearson Education, 2011.

[19] J. Palviainen, Introducing Continuous Integration for C and C++ Software Development Projects on Linux Platform, Lappeenranta: Lappeenranta University of Technology department of Information Technology, 2009.

[20] M. Fowler y M. F. , «Continuous Integration,» 1 May 2006. [En línea]. Available: https://www.academia.edu/download/33643236/ContinuousIntegration.pdf. [Último acceso: 11 Mar 2024].

[21] R. S. Pressman, Software Engineering: A Practitioner's Approach, New York: McGraw-Hill, 2005.

[22] R. B. S. L. Jean-Paul Arcangeli, «Automatic deployment of distributed software systems: Definitions and state of the art,» *Journal of Systems and Software,* vol. 103, pp. 198-218, 2015.

[23] M. Shahin, M. A. Babar y L. Zhu, «Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices,» *IEEE Access,* vol. 5, pp. 3909-3943, 2013.

[24] K.-J. S. Brian Fitzgerald, «Continuous software engineering: A roadmap and agenda,» *The Journal of Systems and Software,* vol. 123, pp. 176-189, 2017.

[25] Docker, «Docker Overview,» Docker, [En línea]. Available: https://docs.docker.com/get-started/overview/. [Último acceso: 15 May 2024].

[26] JetBrains, «TeamCity,» JetBrains, [En línea]. Available: https://www.jetbrains.com/teamcity/. [Último acceso: 15 May 2024].

[27] Jfrog, «JFROG Artifactory,» Jfrog, [En línea]. Available: https://jfrog.com/artifactory/. [Último acceso: 17 Apr 2024].

[28] DPP, «What is D++ (DPP)?,» DPP, [En línea]. Available: https://dpp.dev/. [Último acceso: 15 May 2024].

[29] Google, «Protocol Buffers,» Google, [En línea]. Available: https://protobuf.dev/. [Último acceso: 15 May 2024].

[30] P. Joye, «Libgd About,» LibGD, [En línea]. Available: https://libgd.github.io/. [Último acceso: 15 May 2024].

[31] Catch2, «What is Catch2,» [En línea]. Available: https://github.com/catchorg/Catch2. [Último acceso: 15 May 2024].

[32] Oracle, «Java Network Launch Protocol,» Oracle, [En línea]. Available: https://docs.oracle.com/javase%2Ftutorial%2F/deployment/deploymentInDepth/jnlp.html. [Último acceso: 16 May 2024].

TecnoCampus

Centres universitaris adscrits a la

upf. Universitat Pompeu Fabra *Barcelona*

Grau en Enginyeria Informàtica de gestió i sistemes d'informació

Desenvolupament d'una *pipeline* de CI/CD per a una aplicació de C++

ANNEXOS

Arnau Reig Méndez
Tutor: Dr. Enric Sesa Nogueras
2023-2024

TecnoCampus
Mataró-Maresme

Codi Font:

- Tests.zip: Codi dels tests desenvolupats.
- ArxiusTeamCity.zip: Arxius interns del servidor de TeamCity
- ArxiusJfrog.zip: Arxius interns del servidor de JFrog Artifactory

Vídeos:

- “CompilacioMesExecucioDiscland.mp4”: Exemple de la pipeline completa on es compila el joc utilitzant les llibreries en Artifactory.
- “ContinousDeployment.mp4”: Exemple de desplegament del joc actualitzat a un Docker en el núvol.
- “ContinousIntegration.mp4”: Exemeple de com s’ha configurat el servidor TeamCity.
- “Jfrog.mp4”: Ús del servidor d’Artifactory per emmagatzemar els binaris