Universidad Internacional de La Rioja

Escuela Superior de Ingeniería y Tecnología

Máster Universitario en Internet De Las Cosas

Master in Internet of Things (IoT)

Visualización de datos e integración de vertical para plataforma *fog-cloud* en Telescopio Robótico de 4 metros

Trabajo fin de estudio presentado por:	Josué Barrera Martín
Tipo de trabajo:	Planteamiento de proyecto IoT
Director/a:	Sergio Márquez Sánchez
Fecha:	Julio 2024

Resumen

El principal objetivo del presente trabajo fin de Máster es el desarrollar una plataforma de control en el borde desplegada sobre *Kubernetes* para un telescopio robótico de 4 metros de diámetro en su espejo primario.

Hoy en día, proyectos que están todavía en su fase de desarrollo, como es el caso del *New Robotic Telescope*, plantean soluciones de este tipo para su sistema de control. En el caso que nos compete, se usarán ideas o conceptos de dicho proyecto, como pueden ser el despliegue de la aplicación en un clúster de *Kubernetes.*

Sin embargo, a diferencia con la propuesta realizada por el equipo de desarrollo del *New Robotic Telescope,* se usará *Tango-Controls* en lugar de **GCS** *(GranTeCan Control System)* como *framework* de desarrollo para el sistema de control, integrando *MQTT* y *Kafka* para la publicación y subscripción de métricas del sistema en tiempo de ejecución.

Finalmente, se esbozará un modelo de mantenimiento predictivo del sistema, haciendo hincapié en el estado de los segmentos del espejo primario.

Palabras clave: Telescopio, Astronomía, *Kubernetes, Docker, Kafka, MQTT, Tango-Controls, Cassandra*

Abstract

The main target of the current thesis is to develop a control system platform on the edge to be deployed using Kubernetes for a 4 meters diameter primary mirror robotic telescope.

Nowadays, projects in the field, which are still in their development stage, such as the *New Robotic Telescope,* are meant to be defined using this kind of solution for their control system. In our case, it has been taken some ideas or concepts from this project, such as the deployment of the application using a *Kubernetes* cluster.

Nevertheless, our based-on *Tango-Controls* proposal, instead of **GCS** *(GranTeCan Control System)* as development framework, integrating *MQTT* and *Kafka* as Pub/Sub mechanism.

Finally, it is sketched a predictive maintenance solution for the facility, paying special attention to the segments of the primary mirror.

Keywords: Telescope, Astronomy, *Kubernetes, Docker, Kafka, MQTT, Tango-Controls, Cassandra*

Contenido

Índice de ilustraciones

Índices de Tablas

INTRODUCCIÓN

1.1. MOTIVACIÓN

En la isla de La Palma (Islas Canarias, España) existe uno de los observatorios astronómicos más importantes del mundo (Observatorio del Roque de los Muchachos, por sus siglas, ORM). En él se puede visitar el ***Gran Telescopio de Canarias*** **(GTC)**, el mayor telescopio del mundo en producción actualmente. A su vez, se espera que en los años venideros vean su primera luz el ***European Solar Telescope (EST)***[1] y el ***New Robotic Telescope (NRT)***[2], aumentando aún más si cabe la importancia científica de estas instalaciones.

Ilustración 1 Localización del Observatorio del Roque de los Muchachos. Fuente: Elaboración propia

Tal y cómo se expone en la página web del Instituto Astrofísico de Canarias, "(…) un telescopio robótico…es una poderosa herramienta para el estudio de las supernovas, ya que la flexibilidad de la programación robótica permite optimizar las secuencias de observación para

[1] (*European Solar Telescope Home.* 2024)

[2] (New Robotic Telescope, 2024)

cada objeto individual. Además de las supernovas, también estamos interesados en clases más exóticas de eventos transitorios. La experiencia ha demostrado que el tiempo de respuesta de un telescopio robótico lo convierte en una herramienta particularmente poderosa para el estudio de transitorios que se desvanecen rápidamente, como los resplandores de las explosiones de rayos gamma. El tiempo de respuesta típico del Telescopio Liverpool, incluyendo el tiempo de giro, el tiempo de asentamiento del espejo, la sobrecarga instrumental, etc., es de entre uno y dos minutos. El objetivo del NRT es reducir este tiempo de respuesta a 30 segundos, con lo que se podrá observar el objeto transitorio antes de que decaiga significativamente su brillo. En este caso, el tiempo de respuesta adquiere relevancia por encima de la apertura de la superficie colectora" (Instituto Astrofísico de Canarias, (., 2024)

1.2. PLANTEAMIENTO DEL TRABAJO

Por las condiciones atmosféricas, al hablar de grandes telescopios, hablaremos de instalaciones en zonas de difícil acceso o remotas, típicamente en zonas montañosas donde no siempre el acceso a pie es fácil de realizar. Habría que pensar, por ejemplo, en los tres observatorios más importantes a nivel mundial: Mauna Kea en Hawái (EE. UU), La Silla en el desierto de Atacama (Chile) y finalmente el Roque de Los Muchachos, en la isla de La Palma (Canarias). El primero se encuentra a una altitud de unos 4200 m. sobre el nivel del mar, mientras que La Silla y el Roque de Los Muchachos se encuentran a unos 2400 m. sobre el nivel del mar aproximadamente.

Dicha altitud, hace que sea poco recomendable para los operarios pasar largas temporadas en las instalaciones. A su vez, cualquier tipo de operación de mantenimiento estará supedita a las condiciones meteorológicas, haciendo especialmente compleja su planificación.

Adicionalmente, la climatología afecta negativamente a la calidad de las observaciones. Por ejemplo, la nubosidad del cielo durante la observación, o la estabilidad de la atmósfera condicionan de manera significativa a la calidad de estas. En este sentido poco se puede hacer para actuar sobre dicho agente, salvo el seleccionar la mejor ubicación geográfica para el telescopio.

En cuanto a la calidad y mantenimiento de los espejos, existe un gran margen de mejora. Por ejemplo, se sabe del efecto pernicioso de la calima en los espejos, ya que provoca que se acumule polvo y suciedad en los mismos (el procedimiento normal ante estas eventualidades es **directamente** cerrar la cúpula y evitar en lo posible exponer el telescopio al ambiente), por lo que un método que evalúe y pueda predecir a futuro el grado de limpieza de estos sería de gran ayuda.

Como se puede suponer, la ubicación remota de dichas instalaciones, así como las condiciones meteorológicas, hacen que este tipo de proyectos, a priori, sean muy susceptibles de integrar soluciones *IoT* en su diseño, para de esta manera poder operar de manera totalmente autónoma (robóticamente, incluso tomando decisiones sobre las observaciones a realizar mediante algoritmos), y al mismo tiempo, facilitar el mantenimiento del telescopio.

1.3. ESTRUCTURA DE CAPÍTULOS

2. Concepto y estado del arte. Descripción del estado actual del entorno tecnológico, haciendo alusión a proyectos que están en producción y a otros aún en fase de diseño y construcción.

3. Contribución del TFE. Descripción de objetivos a cubrir con el presente trabajo. También se explica las diferentes partes que componen la solución.

4. Desarrollo de la solución. Definición de una prueba de concepto donde la idea sea plasmada con un desarrollo software de esta.

5. Análisis y Conclusiones. Exposición de las contribuciones principales del Trabajo Fin de Estudio.

2. Contexto y estado del arte

2.1. Introducción

Desde los orígenes del ser humano, éste ha mostrado fascinación por la bóveda celeste y su influencia en La Tierra. Durante la transición de ser nómada a sedentario, el estudio de los astros por parte del hombre buscaba determinar los periodos óptimos para la caza o la recolección o el poder predecir la llegada de las estaciones.

Ya en la Edad Antigua, los babilonios dejaron constancia en sus escritos cuneiformes los hallazgos que realizaron en materia de astronomía - por ejemplo, véase las *Tablilla de Venus de Ammisaduqa* -, siendo los griegos, según es mencionado en la obra *Las Nubes* de Aristófanes[3], los primeros en dejar constancia de la utilización de un instrumento óptico.

Entrada la Edad Moderna (justamente en 1609), Galileo Galilei construyó uno de los primeros telescopios de tipo refractor con lentes ópticas como tal, descubriendo con su uso, los cráteres de la Luna, las lunas de Júpiter, las manchas solares y las fases de Venus. Este conjunto de descubrimientos fue expuesto en su obra *Sideros Nuncios*, obra que se considera un hito fundamental en la historia de la astronomía.

Ya en la década de los años 70 del siglo XX., se empieza a pensar en lo que se denominan telescopios de espejos segmentados, siendo el telescopio Keck[4], en el observatorio de *Mauna Kea* de Hawaii, el proyecto pionero que marcó el comienzo de esta nueva categoría de telescopios.

Esta idea busca conseguir mayores superficies colectoras de fotones, a base de combinar espejos que trabajen coordinadamente aplicando técnicas de óptica adaptativa, logrando que se comporten por tanto como un espejo único y continuo[5]. Esto ha significado un aumento

[3] (*Aristofanes: Nubes*2005)

[4] (W. M. Keck Observatory, 2024)

[5] Tal y como se puede apreciar en la ilustración el aumento significativo del tamaño de los espejos primarios gracias a la utilización de espejos de segmentos.

significativo de la capacidad de observación, siendo hoy en día lo que se entiende como estado del arte en cuanto a diseño y fabricación de ópticas para observación astronómica.

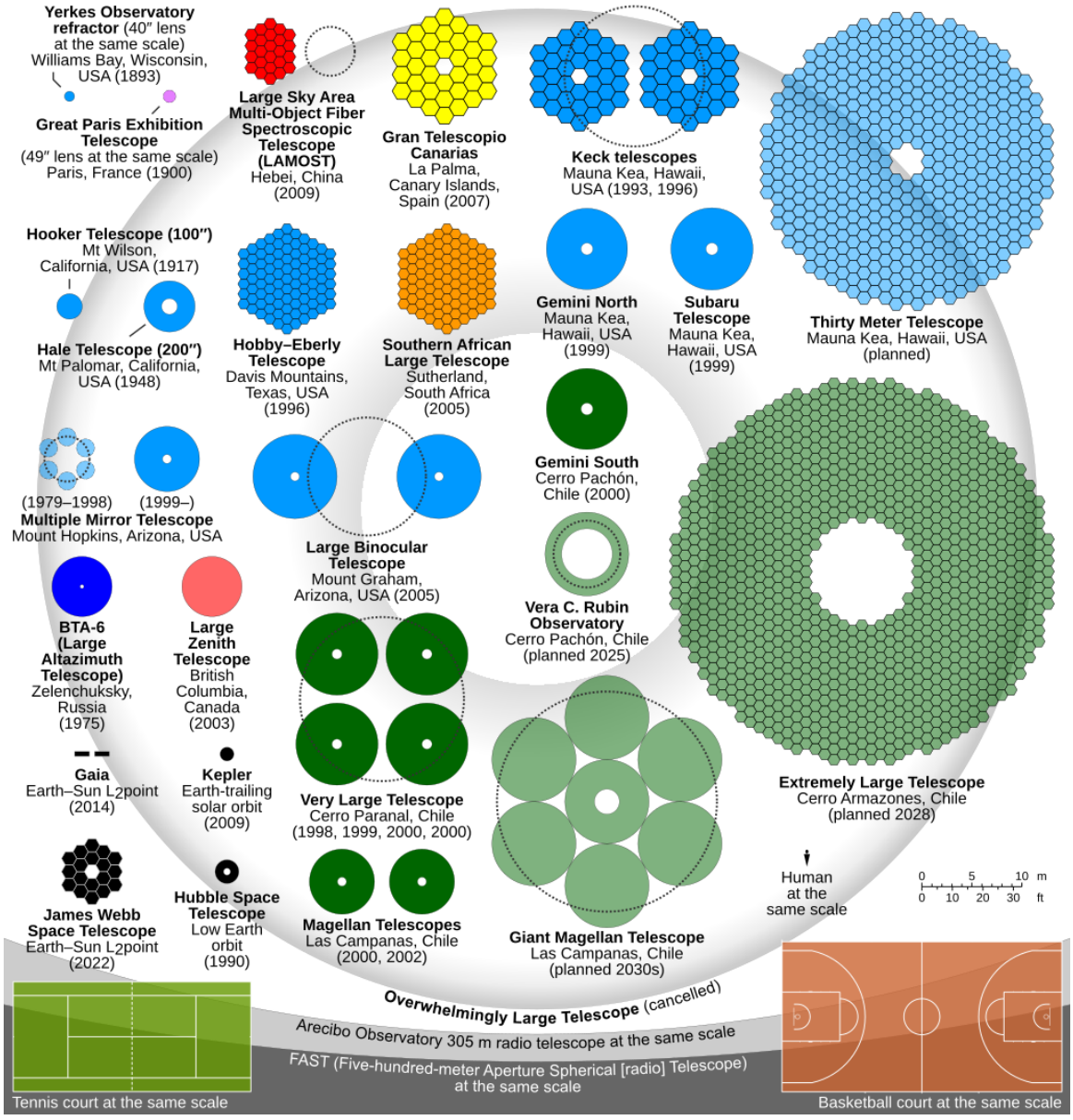

Ilustración 2 Comparativa del tamaño nominal de los espejos primarios de los principales telescopios ópticos reflectantes. Fuente: (anonymous user from IP & 71.41.210.146, 2024)

2.2. Descripción general del contexto del proyecto

El autor del presente proyecto ha estado ligado profesionalmente en el pasado en el desarrollo del sistema de control del NRT. Este telescopio, que, a fecha de redacción de este trabajo, aún está en fase de diseño, se plantea como una plataforma totalmente robotizada,

donde las observaciones se realizarían de una manera autónoma. Su espejo primario, de 4 metros de diámetro, consistirá en 18 segmentos hexagonales, donde cada uno tiene una longitud de 960 mm. A su vez, la plataforma podrá alojar en su *folded Cassegrein* hasta un máximo de 7 instrumentos.

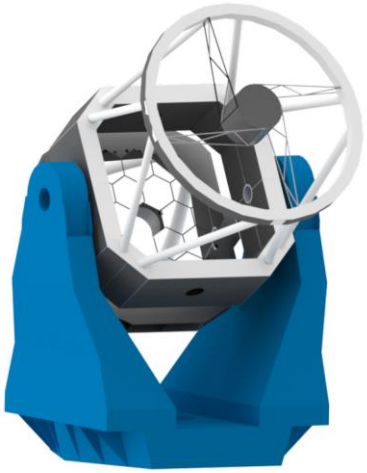

Ilustración 3 Renderizado preliminar del New Robotic Telescope, donde se aprecia el espejo primario de 4 metros de diámetro aproximadamente, con sus 18 segmentos hexagonales. Fuente (Jermak et al., 2020)

La particularidad del sistema de control del NRT consiste en su despliegue basado en herramientas de orquestación de contenedores *Docker.* La solución propuesta por el equipo de ingenieros de software del NRT está basada en la adaptación del sistema de control de GTC[6] a un entorno de *Kubernetes.* Dicho sistema de control, el cual lleva en producción varios lustros en GTC, se basa en un sistema distribuido de control desplegado sobre entorno Linux, donde se usa CORBA[7] como *middleware.* En el caso de la solución planteada por el equipo de software del NRT, se ha conseguido demostrar la factibilidad, no solo de tener un despliegue distribuido sobre *Kubernetes,* sino también la integración de un secuenciador basado en *Apache Airflow* como herramienta de dispensador de procedimientos.

[6] También conocido como **GranTeCan Control System**, por sus siglas, **GCS**

[7] (Sharma, 2022) (Parashar, 2022)

En el caso del presente Trabajo Fin de Estudios, se procederá a plantear una **solución basada en *Tango-Controls,*** en lugar de GCS como *framework* del sistema de control***.*** Esta decisión está supeditada al carácter propietario de GCS*,* mientras que, por el contrario**, *Tango-Controls* es un framework *open source,*** con un uso contrastado en proyectos del ámbito de la radioastronomía como es *SKAO – Square Kilometer Array Observatory –* (SKA Observatory, 2024)*.*

La idea, por tanto, para el presente Trabajo de Fin de Máster es levantar una **plataforma de control para un telescopio robótico desplegada sobre *Kubernetes***. Se tratará de definir una solución que sea válida para cualquier tipo de futura plataforma de observación, utilizando para ello **herramientas *open source*** que sean de uso reconocido en la industria.

Por experiencia del autor en astronomía, en el pasado desarrollador del sistema de control del NRT, para cada proyecto nuevo es usual plantear una solución particularizada, realizando todo el desarrollo prácticamente desde cero. Para el caso particular del NRT no se siguió esta filosofía, sino lo contrario, se hizo un planteamiento más pragmático, poniendo el esfuerzo y el trabajo en adaptar y reutilizar un sistema de control existente y funcional como el GCS. Sin embargo, y como inconveniente insalvable muchas veces, el **carácter propietario de GCS**, hace que la adopción de este por parte de nuevos actores sea una opción poco asequible.

Esto es en lo relativo a lo que entenderíamos como sistema de control. Sin embargo, hay que decir que para el presente proyecto se ha dispuesto un conjunto híbrido, donde se une una parte de computación en la niebla con otra parte de computación en la nube. Las implicaciones de esta división son reseñables:

- Se dispondría de un *datacenter* de baja latencia (computación en la niebla) para integrar el sistema de control del telescopio, basándose en *Tango-Controls* como *framework* de desarrollo*.*
- Se dispondría de un *datacenter* de alta disponibilidad (computación en la nube) para realizar el tratamiento de los datos generados, tanto por lotes como en tiempo real.

Con esta configuración, a efectos prácticos, al *datacenter* en la nube podrían volcar datos ***N*** *datacenters* en la niebla. Es decir, múltiples telescopios generarían datos sobre el estado real

en cada momento de sus espejos, para tratarse posteriormente de manera global para extraer conocimiento.

Esto último es interesante, ya que se podría monitorizar la evolución de la calidad de imagen de cada espejo producido por el **Centro de Sistemas Ópticos Avanzados**[8], que será el proveedor de óptica avanzada para los telescopios del Observatorio del Roque De Los Muchachos.

Cabe destacar que actualmente, no existe una integración de estos dos elementos:

- Manufacturación de espejos. Por parte del CSOA en este caso.
- Monitorización en producción de calidad de imagen de estos. Telescopios del Observatorio del Roque De Los Muchachos.

Lo cual da una importancia y singularidad única al planteamiento propuesto por el autor en este Trabajo de Fin de Máster. Para conseguir esta integración *fog vs cloud,* se plantea el uso de *Kafka* como plataforma para gestionar la telemetría que es enviada (vía *MQTT*) por cada uno de los componentes (en el argot también conocidos como *devices*) de *Tango-Controls*, y *Cassandra* como elemento de almacenamiento de datos.

Es por ello por lo que, en el presente Trabajo de Fin de Estudios se planteará una solución basada en ***Tango-Controls*** sobre ***Kubernetes***. Además, para la recogida de métricas se usarán herramientas consideradas estándares en la industria hoy en día, como son **Kafka** o **MQTT** y bases de datos **NoSQL** como ***Cassandra***.

2.3. Proyectos relacionados con el tema del TFE

A la hora de citar casos de telescopios robóticos, habría que empezar por citar al *Liverpool Telescope* (Liverpool Jonh Moores University, 2024). Se trata del proyecto que sirvió de punto de partida para el NRT. El equipo de la *Universidad de Liverpool John Moores*[9] ha sido uno de los

[8] Por sus siglas CSOA (Instituto de Astrofísica de Canarias, - IAC, 2024)

[9] Por sus siglas LJMU

padres fundadores del proyecto NRT, aportando su conocimiento y experiencia el desarrollo y gestión de un telescopio robótico.

A su vez, en todo lo relacionado con el diseño, desarrollo y gestión de un gran telescopio, el proyecto *GranTeCan* (Instituto Astrofísico de Canarias, 2024) es un referente mundial. Ha sido este proyecto el que ha desarrollado el sistema de control (Penataro et al., 2000) (Filgueira & Rodriguez, 1998), el cual es el punto de partida para el desarrollo y diseño de dos de los proyectos más importantes actualmente en la astronomía, como son NRT (New Robotic Telescope, 2024) y el EST (European Solar Telescope Home. 2024). En el caso particular de NRT, se ha adaptado el sistema de control de GTC a un entorno *dockerizado* (Bento et al., 2022a). En este proyecto, NRT, el autor de este TFE obtuvo la experiencia inicial al desarrollar y desplegar aplicaciones basadas en microservicios (Bento et al., 2022b).

Finalmente, cabe destacar el trabajo del equipo de SKAO. En el ámbito de la radioastronomía, el proyecto *Square Kilometre Array Observatory,* ha hecho un trabajo que se utilizará como base de partida de este TFE. En este caso se trata de uno de los más importantes colaboradores y desarrolladores del framework **Tango-Controls** (Tango Controls, 2024).

2.4. Tecnologías relacionadas con el tema del TFE

2.4.1. Docker

En la edición de 2013 de la conferencia *PyCon*, Solomon Hykes impartió una presentación sobre una tecnología llamada *Docker*. Desde entonces esta tecnología se ha convertido en una herramienta indispensable para desarrollar aplicaciones software. "Docker es una plataforma que permite crear, probar e implementar aplicaciones rápidamente, empaquetándolas en unidas de software estandarizadas llamadas contenedores, que incluyen todo lo necesario para que el software se ejecute, incluyendo bibliotecas, herramientas de sistema, código y tiempo de ejecución". (Amazon Web Services Inc., 2023).

2.4.2. Kubernetes

Kubernetes fue un proyecto interno de Google liberado en el año 2014. Es una plataforma de código abierto, utilizado en la orquestación de contenedores, que permite y facilita la automatización y el despliegue de una manera declarativa de una aplicación (Google, 2022).

En el caso que nos compete, la distribución a utilizar será *k0s* (k0s, 2024). Se trata de una distribución para despliegues bajo *Kubernetes* en *bare-metal.* Esto lo hace especialmente interesante a la hora de trabajar con despliegues en la niebla, para *IoT*, o para *clouds* privados. Hay que citar que, para el desarrollo en local, se ha usado una distribución de *Kubernetes* basada en *Kind* (Kind, 2024) ligera y sencilla, pero que a su vez su uso no recomendable para un entorno de producción.

En lo que respecta a la gestión de distintos paquetes de aplicaciones, se usará *Helm* (Helm, 2024). Entiéndase que, a fecha de la redacción del presente proyecto, ***Helm*** se considera el estándar de facto en la industria para gestionar paquetes en *Kubernetes.*

2.4.3. Tango-Controls

Se trata de una *framework* utilizado para crear sistemas SCADA de manera distribuida. Permite la definición de *devices* (Tango Controls, 2024), que vienen a ser unidades funcionales de código que gestionan una abstracción en particular. Agnóstico en cuanto a lenguajes de programación, lo cual hace que sea bastante interesante a la hora de poder desarrollar cualquier nueva funcionalidad. Ya ha sido probado con éxito en sistemas *clusterizados* por el equipo de *SKAO* (SKAO, 2024) en un proyecto de gran envergadura para radioastronomía.

2.4.4. Kafka

Como se explica su website, "Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications" (Apache Kafka, 2024).

Lo que se persigue con la integración de esta tecnología en el presente proyecto es el disponer de un mecanismo de alta escalabilidad a la hora de gestionar la telemetría generada por el telescopio. Esta alta escalabilidad permitiría en un futuro, si así fuese necesario, expandir la plataforma a otros telescopios ya presentes en el observatorio, o proyectados a futuro en el mismo.

2.4.5. Cassandra

A la hora de plantear qué tipo de base de datos utilizar para el presente proyecto, se ha decidido optar por ***Cassandra*** (Singh, 2024) (Bekker, 2018) (Vinci, 2024). Este tipo de base de datos está especialmente indicado para situaciones donde haya que persistir datos generados por sensores, con su correspondiente marca de tiempo, o también para plataformas con una alta carga de trabajo de escritura.

Cassandra es una base de datos NoSQL, distribuida, *open source*, y orientada a columnas. Tiene la particularidad de no seguir un patrón *masterless.*

2.4.6. MQTT

Tal y como se expone en los apuntes del Tema 3 de la asignatura *Plataforma de Internet de las Cosas de Google,* impartida en el *Máster Universitario en Internet de las Cosas / Master in Internet of Things (IoT) - PER 8325 octubre 2023* (Universidad Internacional de La Rioja, 2024), se trata de un estándar OASIS (Organización para el Avance de Estándares de Información Estructurada) para la conectividad de dispositivos *IoT*. Protocolo de comunicación basado en mecanismos de publicación y suscripción, simple y liviano, centrado en dispositivos con pocos recursos y redes de bajo ancho de banda.

2.4.7. GitHub

En la fecha de redacción del presente proyecto, GitHub se considera una plataforma de facto estándar en la industria para el desarrollo de proyectos de software. Según se indica en la propia web de GitHub, "(…) GitHub is a cloud-based platform where you can store, share and work together with others to write code (…)" (GitHub, 2024).

2.4.8. Argo CD

La integración de *Argo CD* como herramienta en un proyecto implica la utilización de una tecnología que permite desplegar una aplicación de manera declarativa. Esto significa de facto estar utilizando una metodología de *Continuos Deployment* en nuestra solución.

Tal y como se muestra en la web de Argo CD, "(…) Argo CD follows the GitOps pattern of using Git as the source of truth for defining the desired application state (…)". (Argo Project, 2024).

2.4.9. FastAPI

Se trata de una librería de Python utilizada para crear de manera rápida APIs. Vienen a ser utilizada de manera similar a otras librerías de Python como Flask, aunque FastAPI destaca por su velocidad de respuesta, a la par con *NodeJS* o *Go* (Ramírez, 2024). La utilizaremos para desplegar el modelo de Machine Learning, y hacerlo accesible a través de petición HTTP.

2.5. Conclusiones sobre el estado del arte

A través de lo expuesto del estado actual del arte, se aprecia una tendencia clara al desarrollo de aplicaciones basadas en microservicios. Esto significa que un despliegue sobre *Kubernetes* es una opción viable y recomendable, si bien, es cierto que hay una curva de aprendizaje bastante pronunciada de partida. Al mismo tiempo, del estudio del *framework Tango-Controls* y GCS se deduce que la utilización de *CORBA* (Penataro et al., 2000) (Filgueira & Rodriguez, 1998) como *middleware* para ejecutar operaciones sobre un *device* en concreto es una manera estandariza de proceder. Ambos, tanto *GCS* como *Tango-Controls* cuentan con una integración a base de plantillas que facilitan al desarrollador crear dichos métodos *CORBA*, si bien, la solución propuesta en *Tango-Controls* cuenta con una integración más elaborada en Python y Java.

También hay que citar la utilización del binomio *Kafka-MQTT.* A priori, podría parecer redundante incluir ambos protocolos en el diseño de la solución, pero son dos tecnologías que se complementan entre ellas.

Como es expuesto en la web de *EMQX* (EMQX Team, 2024), la experiencia de uso con clientes de *Kafka* suele mostrar una cierta complejidad y una alta demanda de recursos. Además, también existen limitaciones en cuanto a la escalabilidad de *topics* por parte de *Kafka*. Finalmente, hay que citar que los clientes de *Kafka* suelen requerir una conexión a la red estable, algo que a veces es bastante complicado conseguir durante todo el tiempo con dispositivos *IoT*. Es por ello, que al combinar *MQTT* con *Kafka* como pareja de trabajo para la telemetría, se pueden paliar los problemas relativos a:

- **Clientes con pocos recursos disponibles**, ya los clientes *MQTT* está pensados para consumir pocos recursos.

- **Problemas de conectividad**. *MQTT* está pensado para trabajar con poco ancho de banda, así como gracias a la definición de un *QoS* y del *Last Will Message,* hacen de MQTT una opción bastante adecuada para redes que pueden presentar problemas de conectividad.
- **Escalabilidad de tópicos.** *MQTT* encaja bastante bien a la hora de gestionar un gran número de tópicos.

Por eso, según lo expuesto en el párrafo anterior, el uso de ambos en la solución parece adecuado. El nexo entre ambos se realizará mediante una pasarela que permitirá un mapeo basado en tópicos entre el bróker de *MQTT* y el clúster de *Kafka*.

Es relevante citar que, a diferencia de GTC y NRT, el presente Trabajo Fin de Máster pretende la integración de técnicas de inteligencia artificial para disponer de una solución de mantenimiento predictivo. Actualmente, en *GranTeCan* no existe algo de este estilo en producción, y en el proyecto *NRT* (a fecha de redacción del presente trabajo, aún en fase de diseño), no se ha incluido de momento algo similar en la vertical. Esto significa una mejora cualitativa significante con respecto a ambos proyectos, ya que las operaciones de limpieza y mantenimiento son claves a la hora de disponer de una calidad aceptable en las observaciones.

3. DESCRIPCIÓN GENERAL DE LA CONTRIBUCIÓN DEL TFE

3.1. Objetivos

Objetivo General:

- Definición de una vertical *IoT* que permita la operación de un telescopio, así como la recogida, análisis y visualización de la telemetría generada durante la operación de las instalaciones.

Objetivos Específicos:

- Disminuir el tiempo empleado en las fases de definición y desarrollo del sistema de control de un telescopio
- Estandarizar las tecnologías utilizadas en el desarrollo y despliegue del sistema de control de un telescopio
- Aumentar la escalabilidad e integración de cualquier telescopio en un observatorio, al disponer de una plataforma *cloud* capacitada para añadir nueva funcionalidad.
- Mejorar el proceso de mantenimiento y fabricación de espejos para telescopios.

3.2. Metodología de trabajo

Para el presente proyecto se planteará una metodología *DevOps* para el desarrollo de la solución. Para ello se hará uso de *GitHub* como plataforma para desarrollo. Desde *GitHub*, definiremos una hoja de ruta donde estarán disgregados los diferentes hitos a conseguir durante la etapa de desarrollo y despliegue de la solución

Al utilizar una metodología *DevOps*, se persigue un ciclo de vida de desarrollo (en sus siglas en inglés SDLC - *Software Development Life Cycle* -) basado en los siguientes pasos:

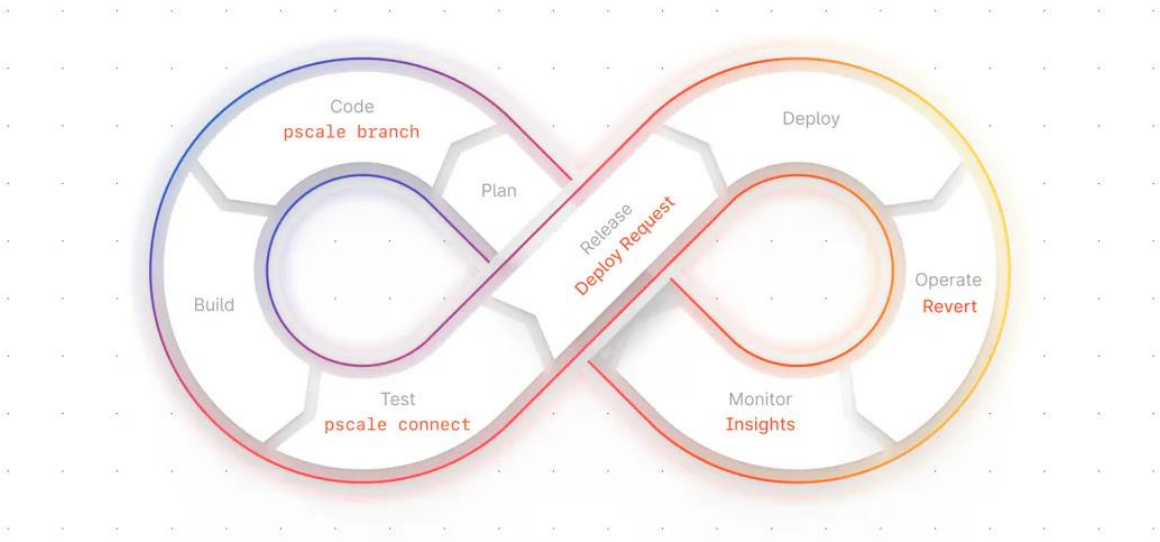

Ilustración 4 Pasos metodología DevOps. Fuente (PlanetScale Inc, 2023)

- ***Planning***. Se generará un conjunto de tareas o *sprints* con el conjunto de tareas a realizar por cada desarrollador
- ***Development***. Cada ingeniero de software desarrollará su código
- ***Build***. De manera automatizada, mediante la ejecución de *pipelines* se generará un conjunto de binarios a partir del código desarrollado por el ingeniero. En este paso se realizarían las operaciones de compilación de aplicaciones, librerías, generación de imágenes *Docker* o paquetes de *Helm*, etc.
- ***Testing***. Para cada binario obtenido en el proceso anterior, se ejecutarán una serie de pruebas unitarios que permitan tener cierto grado de validación de funcionalidades
- ***Release***. Una vez han sido pasadas con éxitos las pruebas unitarias, se procederá a almacenar en un registro privado los binarios generados a nivel de *pipeline.* Esto significa que estos archivos estarán versionados, proceso que se hará también de manera automatizada a nivel de *pipeline,* para de esta manera poder identificar cada una de las versiones generadas durante todo el ciclo de vida de la aplicación.
- ***Deploy***. El proceso de despliegue de la aplicación se hará mediante el uso de herramientas que permitan la orquestación y/o automatización del proceso. En nuestro caso, al tratarse de una aplicación basada en *microservicios*, será planteada la utilización *ArgoCD* para su despliegue.

- ***Operate***. El paso de operación es gestionado por el equipo de operaciones, que es el encargado de asegurar la disponibilidad, mantenimiento y escalabilidad de los equipos de producción
- ***Monitor***. En este paso se monitoriza lo que está sucediendo en la aplicación, tanto a nivel de *performance,* como de *UX (user experience)* y gestión de incidentes.

Desde un punto de vista de planificación en el tiempo, se planteará todo el proyecto utilizando para ello las herramientas que vienen ya integradas en *GitHub.*

3.3. Descripción general de las partes o componentes de la propuesta

La solución planteada estará basada en una arquitectura lambda (Vinci, 2023). En nuestro caso, se diferenciarán las capas que realizan la lógica de procesamiento de datos según si el usuario final pertenece al personal que opera el telescopio, o a la comunidad científica.

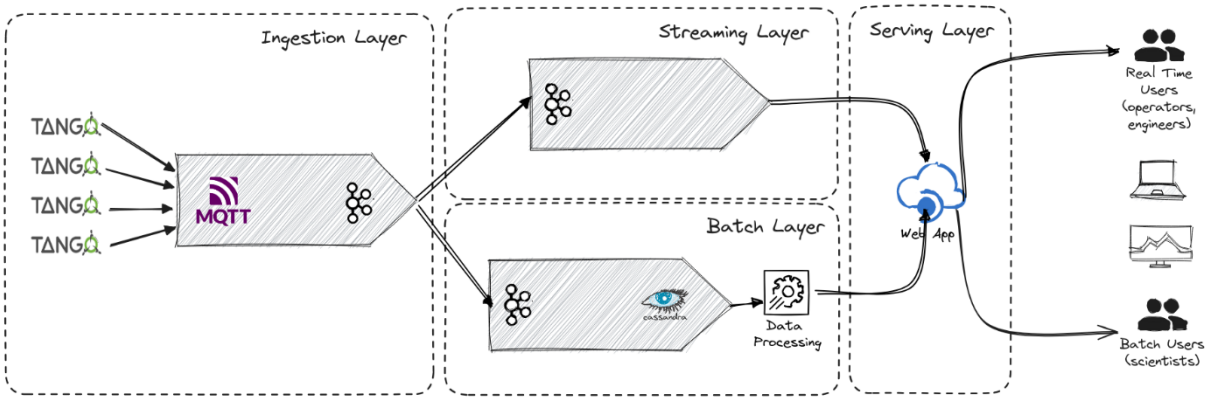

Ilustración 5 Arquitectura propuesta para la solución. Fuente: elaboración propia

La primera capa o ***Ingestion Layer***, define los generadores de datos. En este caso, se tratará de cualquier abstracción o *Device* integrado con *Tango-Controls,* que publicará datos vía *MQTT* hacia las capas siguientes.

A continuación, lo que denominamos como ***Streaming Layer*** (también se puede encontrar como *Hot Path* en literatura relativa) corresponde a la visualización y operación del telescopio en tiempo real. En esta capa se recogen los datos del protocolo *MQTT*, y mediante un traductor de protocolos, se realiza la pasarela hacia *Kafka*. Luego, los clientes de *Kafka* visualizarán los datos en tiempo real.

Por el contrario, la *Batch Layer* o C*old Path* es el encargado de procesar los datos persistidos en *Cassandra* u otro mecanismo de almacenamiento por lotes. Estos datos pueden ser desde la reducción de datos de observaciones científicas (archivos *fits,* principalmente), hasta el entrenamiento de modelos predictivos para el mantenimiento de los espejos del telescopio. Este tipo de datos procesados están orientados principalmente hacia la comunidad científica, para que de esta manera puedan trabajar en proyectos de investigación

3.3.1. Alcance y limitaciones

El presente Trabajo de Fin de Máster se limitará a la creación de una prueba de concepto donde se despliegue de manera simulada la capa en la niebla. Para ello, se utilizará *Kind* para crear un clúster de *Kubernetes* de manera local, donde se ejecutarán los siguientes elementos:

- *Tango-Controls Framework*. Desarrollo de *devices*:
    - Axis X Driver
    - Axis Y Driver
    - Dome
    - Simulated Instrument.
    - Edge Sensor
- Despliegue de bróker *MQTT* utilizando la plataforma EMQx.
- Despliegue de *Apache Kafka*. Integración de pasarela *MQTT ~ Kafka*
- Despliegue de *Cassandra* para persistencia
- Despliegue de pasarela *Apache Kafka ~ Cassandra*
- Despliegue de *Grafana*
- Despliegue de planteamiento de modelo *ML* usando *FastAPI*
- Visualización de datos en tiempo real usando un conector para *Cassandra* en *Grafana*

3.3.2. Listado de participantes

- Josué Barrera Martín

  Estudiante del Máster Universitario en Internet de las Cosas de la Universidad Internacional de La Rioja (UNIR).

3.3.3. Tecnologías implicadas

En lo relativo a las tecnologías a utilizar, han sido ya definidas previamente a lo largo del presente texto:

- *MQTT*
- *Kubernetes*
- *Docker*
- *Helm*
- *Cassandra*
- *Grafana*
- *Tango-Controls*
- *Apache Kafka*

Sobre los lenguajes de programación, debido a la naturaleza agnóstica de un despliegue basado en microservicios, los *devices* de *Tango-Controls* podrían haber sido desarrollados en **C++, Python, Java** etc., siendo **Python**, por *expertise* propio del autor, la opción finalmente utilizada. Para la aplicación web desplegada en el *cloud,* las opciones serían diferentes. Este paquete de trabajo queda fuera del alcance del presente trabajo.

3.3.4. Arquitectura, componentes e integración de tecnologías

La vertical planteada consiste en la definición y despliegue de dos capas funcionales separadas:

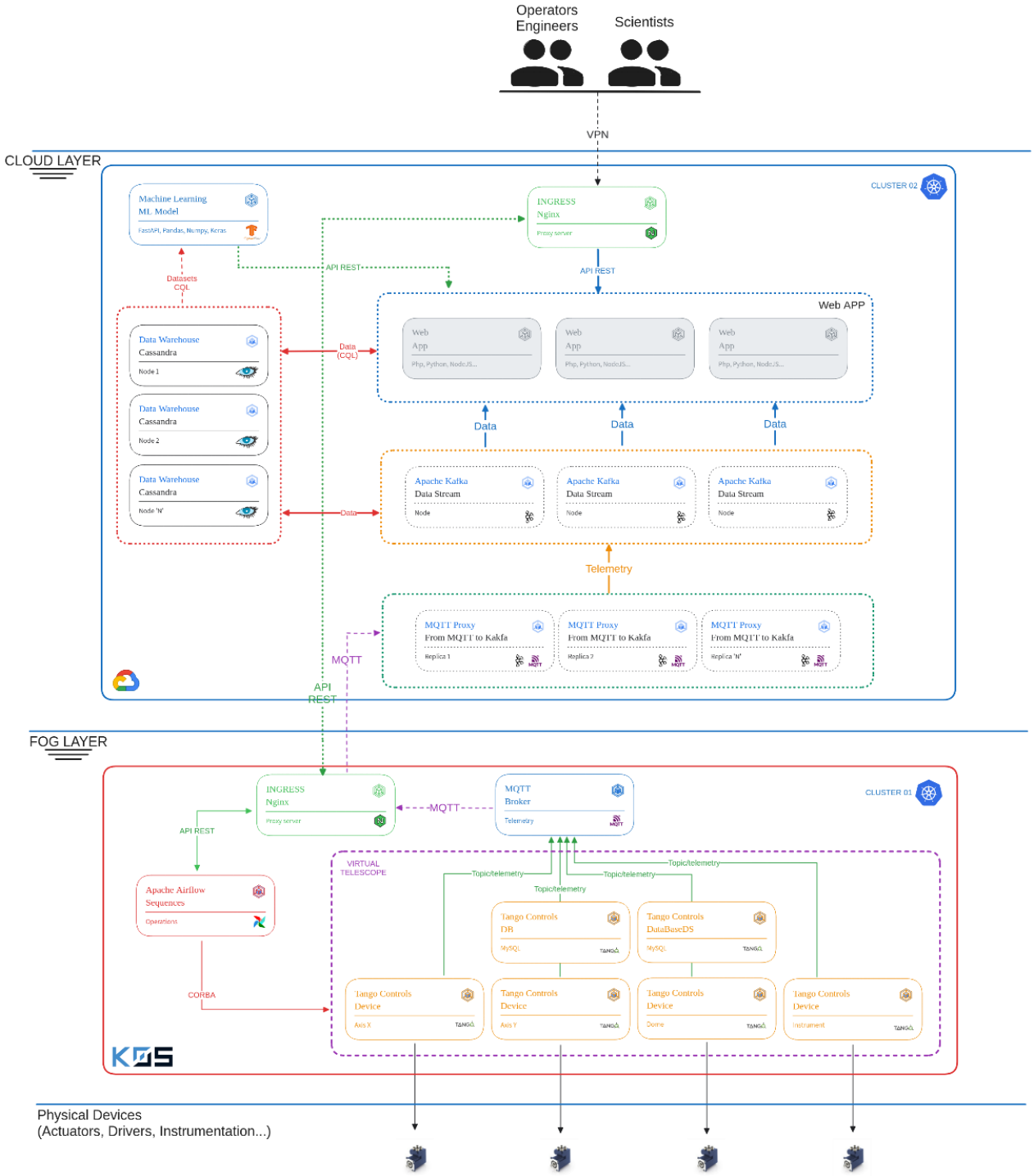

Ilustración 6 Diagrama por capas de la aplicación. Fuente: elaboración propia

Por la (usualmente), remota ubicación de los observatorios, en caso de condiciones meteorológicas adversas prolongadas durante varios días, se plantea el despliegue en una capa separada y funcional en sí misma para la lógica de control y operación del telescopio.

Esta primera capa o ***fog layer*** se trata de un *Datacenter* aprovisionado *in situ* en el propio observatorio del Roque de los Muchachos. Así se conseguiría **tener una red de baja latencia** que permita controlar de manera automatizada la plataforma de observación. Estamos hablando de la dotación en *bare-metal* de un clúster de *Kubernetes* desplegado con *k0s.* Este clúster dispondría de tantos nodos como se necesiten, dependiendo tanto de la capacidad computacional como el acceso físico al hardware necesario para operar el telescopio: motores, *drivers,* instrumentación científica, actuadores para óptica adaptativa etc.

En este centro de datos se deberá desplegar toda la lógica de control en tiempo real del telescopio: *devices* de *Tango-Controls,* bróker *MQTT* y cuantos *plugins* sean necesarios para hacer funcional un clúster *bare-metal* de *Kubernetes.* **Por simplicidad y limitaciones de alcance, se definirán únicamente los *Devices* más representativos: cúpula, altitud, azimut, sensores de borde (cada segmento del espejo primario dispone de un sensor para conseguir el cofaseado[10] y alineamiento en tiempo real de los espejos) e instrumento.** Así se conseguirá cierta resiliencia en caso de problemas de conexión con la capa superior.

Esta capa superior se denominada ***cloud layer*** y es donde se desplegará la lógica de aplicación científica de la solución. Aquí se persistirán los datos en una base de datos *Cassandra,* siendo dichos datos de telemetría recogidas gracias a una instancia *de Apache Kafka.* Existirá una pasarela entre la capa en la nube y la capa en la niebla *(cloud vs fog).* Esta pasarela hará de **traductor de protocolo** *MQTT* a *Kafka*, evitando mediante **replicación** minimizar el riesgo planteado por una situación de **SPOF** *(SINGLE POINT OF FAILURE).*

Al tratarse de un telescopio robótico, de alguna manera, tendrá que poder ejecutarse de manera programática las operaciones procedimentales típicas de este tipo de instalaciones:

[10](Guerra Ramos, 2020)

aiming, slewing, homing, calibration, etc. Para proveer con dicha funcionalidad, una posibilidad sería integrar *Apache Airflow* como servicio de secuenciación.

Adicionalmente, en esta capa se alojará la aplicación web que será la interfaz de usuario del sistema. En dicha web, habrá funcionalidades diferenciadas y accesibles dependiendo del rol del usuario, *RBAC* por sus siglas en inglés - *Role Base Access Control* -

Entre otras funciones propias de esta aplicación web, se podrían citar:

- Gestión de usuarios *(Login/Logout)* dependiendo de los roles asignados
- Visualización de telemetría
- Gestión de alertas y monitorización de la plataforma
- Acceso a los datos de observación realizados (datos científicos)
- Registro de propuestas científicas (planteamiento de posibles trabajos de observación astronómica).

La usará el personal del telescopio y científicos autorizados que trabajarán con los datos generados por la plataforma.

3.3.5. Resultados esperados

Con el presente Trabajo de Fin de Máster se pretende demostrar cómo se puede realizar un planteamiento y despliegue de una arquitectura *fog-cloud,* con visualización y modelización de datos con fines predictivos de un observatorio astronómico.

1. Creación y despliegue de sistema de control de telescopio
2. Creación y despliegue de sistema de persistencia de datos, tanto de telemetría como de adquisición científica.
3. Creación y despliegue de sistema de visualización de datos
4. Simulación y despliegue de un modelo de *Machine Learning*. Se planteará la metodología a seguir para obtener un despliegue e integración del modelo a nivel de clúster en el *cloud.*

NOTA:

En este apartado de resultados, se da por hecho que al haber utilizado datos sintéticos para crear y exportar un modelo de *Machine Learning* en formato *pkl,* **las predicciones obtenidas del mismo no son reales**. La idea es demostrar cómo se **utilizarían los datos recopilados durante la vida útil del observatorio en la base de datos *Cassandra*, y cómo se harían accesibles a los usuarios finales dicha información a través de modelos de *Machine Learning.***

3.3.6. Planificación general

La planificación del proyecto será planteada desde la plataforma *GitHub*. Se definirá el desglose de los paquetes de trabajo, el tiempo estimado para cada uno y los equipos de desarrolladores asignados a ellos.

Se tendrán hasta 3 equipos diferentes trabajando al unísono en el proyecto, siendo el *kickoff* del proyecto el día **30 de septiembre de 2024**, y su *deadline* planificada para **el día 30 de septiembre de 2025.**

NOTA:

Para una mejor visualización de las distintas herramientas utilizadas (proyecto en GitHub, diagramas de *Gantt*, *tableros Kanban*, *backlogs…*), remítase al **Anexo A**

Tabla 1 Distribución de equipos de desarrollo de software. Fuente: elaboración propia

Equipo SRE	
Integrantes	- **1 SRE senior** - **3 SRE junior**
Tareas	**Descripción**
Creación de infraestructura	Despliegue de clústeres de *Kubernetes*, tanto en la nube como en la niebla *(Cloud y Fog layers*). Generación y desarrollo de las herramientas necesarias para automatizar

	el proceso, es decir, desarrollo de soluciones basadas en *Terraform* y *Ansible* principalmente. La idea sería poder disponer de diferentes entornos replicables para cada capa: entorno de desarrollo, entorno de pruebas o *testing* y entorno de producción.
Gestión de código fuente	Disposición y gestión de repositorios y registros privados de artefactos. Esto implica gestionar las cuentas de *GitHub*, el despliegue de una instancia de *Nexus* o similar como registro de artefactos, creación de pipelines, despliegue de *runners* para CI/CD...
Despliegue de aplicación	Creación y desarrollo de una estrategia de despliegue de solución: definición de arquitectura a nivel clúster, es decir, creación de los objetos de *Kubernetes* que permitan operar la plataforma. A su vez, esto significa la implantación de una estrategia de *Continuos Deployment,* donde se hará uso de *Argo CD* como herramienta para perseguir este hito
Equipo Niebla	
Integrantes	- **1 analista software** - **3 desarrolladores Python** - **2 desarrolladores C++** - **2 ingenieros de automatización industrial**
Tareas	**Descripción**
Desarrollo de Telescopio Virtual	Desarrollo del código de todos los *Devices* de *Tango-Controls* que definirán cada uno de los componentes del Telescopio Virtual

Integración de mensajería	Desarrollo de la solución que permita automatizar la generación de telemetría de cada uno de los *Devices*. Esto implica definir la configuración del bróker *MQTT* y su despliegue en el clúster mediante una *Helm Chart*.
Integración de secuenciador	Desarrollo y despliegue mediane *Helm Chart* de *Apache Airflow* como dispensador de tareas basados en *DAGs*. Cada *Directec Acyclic Graph* representará una abstracción de cada una de las operaciones o procedimientos que podrá realizar el Telescopio Virtual.
Equipo Nube	
Integrantes	- **1 analista software** - **5 desarrolladores software**
Tareas	**Descripción**
Integración de mecanismo de *streaming*	Integración en *cloud* de una instancia de *Kafka*. Esto implica además la integración de dicha instancia con *MQTT* mediante una pasarela *MQTT* vs *Kafka*
Integración de mecanismo de persistencia	Integración en *cloud* de un clúster de *Cassandra*. Esto implica la definición de la estructura de datos, la integración con *Kafka* como *Data Sink*
Desarrollo Aplicación Web	Desarrollo de una plataforma web que permita la gestión del sistema en su totalidad: - Separación de funcionalidades dependientes de RBACs - Gestión de usuarios - Visualización de telemetría

	- Visualización de alarmas y eventos - Gestión de observaciones: descarga de datos, propuestas científicas, tracking de observaciones...

3.3.7. Presupuesto y retorno esperado de la inversión

3.3.7.1. Recursos humanos

En cuanto a la gestión de recursos humanos para la ejecución del proyecto, se ha realizado una aproximación en función de las tareas propuestas en el proyecto de GitHub. De esta manera, se puede hacer una previsión del coste por contratación de personal, el cuál será mostrado a continuación:

Tabla 2 Distribución de paquetes de trabajo para cada equipo de desarrollo

EQUIPO NIEBLA										
WP	**Título Partida**	**Descripción de partida**	**Horas**	**Recursos**	**Precio Hora (€)**	**Total**	**Riesgo (%)**	**Beneficio (%)**	**Total**	**97460 €**
1.1	Integración de Tango Controls Framework	Creación de imágenes Docker: - Definición de dockerfiles, con stages para desarrollo y producción - Definición de devcontainer - definición de docker composes Creación de Helm Chart: - Securización del despliegue Creación de pipeline: - Docker Build - Helm Package - Unit Testing - Publish	120	Analista (x1)	45	5400	5	5	5940	
1.2	Desarrollo *Device* Instrumentación	Desarrollo de un instrumento para adquisición de imágenes: - Generación de archivos fits, con sus cabeceras incluidas - Publicación de telemetría con MQTT Creación de imágenes Docker: - Definición de dockerfiles, con stages para desarrollo y producción - Definición de devcontainer - definición de docker composes	320	Desarrollador C++ (x1)	40	12800	5	5	14080	

		Creación de Helm Chart: - Securización del despliegue Creación de pipeline: - Docker Build - Helm Package - Unit Testing - Publish								
1.3	Desarrollo *Device* Axis X Driver	Desarrollo de un device para controlar el movimiento en el eje X - Integración con EtherCAT vía OPC UA o ADS - Publicación de telemetría con MQTT Creación de imágenes Docker: - Definición de dockerfiles, con stages para desarrollo y producción - Definición de devcontainer - definición de docker composes Creación de Helm Chart: - Securización del despliegue Creación de pipeline: - Docker Build - Helm Package - Unit Testing - Publish	320	Desarrollador C++ (x1)	40	12800	5	5	14080	

1.4	Desarrollo *Device* Axis Y Driver	Desarrollo de un device para controlar el movimiento en el eje Y - Integración con EtherCAT vía OPC UA o ADS - Publicación de telemetría con MQTT Creación de imágenes Docker: - Definición de dockerfiles, con stages para desarrollo y producción - Definición de devcontainer - definición de docker composes Creación de Helm Chart: - Securización del despliegue Creación de pipeline: - Docker Build - Helm Package - Unit Testing - Publish	320	Desarrollador C++ (x1)	40	12800	5	5	14080	
1.5	Desarrollo *Device Dome Device*	Desarrollo de un device para controlar la apertura o clausura de la cúpula - Integración con EtherCAT vía OPC UA o ADS - Publicación de telemetría con MQTT Creación de imágenes Docker: - Definición de dockerfiles, con stages para desarrollo y producción - Definición de devcontainer - definición de docker composes Creación de Helm Chart: - Securización del despliegue Creación de pipeline:	320	Desarrollador Python (x1)	35	11200	5	5	12320	

		- Docker Build - Helm Package - Unit Testing - Publish								
1.6	Desarrollo integración con MQTT	Integración de todos los devices con bróker MQTT Creación de imágenes Docker: - Definición de dockerfiles, con stages para desarrollo y producción - Definición de devcontainer - definición de docker composes Creación de Helm Chart: - Securización del despliegue Creación de pipeline: - Docker Build - Helm Package - Unit Testing - Publish	320	Desarrollador Python (x1)	35	11200	5	5	12320	
1.7	Integración de secuenciador	Integración de Apache Airflow: - Creación de imagen docker customizada - Creación de Chart con sus dependencias, y su integración en ArgoCD - Creación de usuarios y configuraciones necesarias en Apache Airflow - Creación, depuración, testing e integración de secuencias propuestas por sistemas	640	Desarrollador Python (x1)	35	22400	5	5	24640	

EQUIPO NUBE

WP	Título Partida	Descripción de partida	Horas	Recursos	Precio Hora	Total (€)	Riesgo (%)	Beneficio (%)	Total	**228800 €**
2.1	Integración de Kafka	Integración en cloud de confluent: - Permisos. Usuarios, login/logout, securización - Persistencia - Replicación de workers - Despliegue en cloud - Registro de schemas	320	Analista (x1)	45	14400	5	5	15840	
2.2	Integración de conexión MQTT Kafka	Integración de conector MQTT para confluent: - Conectar el plugin de MQTT para confluent	320	Analista (x1)	45	14400	5	5	15840	
2.3	Integración de Cassandra	Despliegue en cloud de un clúster de Cassandra - Conectar una pasarela para volcar datos desde Kafka hacia Cassandra	320	Desarrollador Python (x1)	35	11200	5	5	12320	
2.4	Desarrollo Web	Creación de plataforma web: - Integración con kafka y cassandra - creación de sistema de gestión de usuarios - ... Integración y despliegue mediante: - creación de imágenes docker - creación de unit tests - creación de helm chart	960	Desarrollador web (x5)	35	168000	5	5	184800	

EQUIPO SRE

WP	Título Partida	Descripción de partida	Horas	Recursos	Precio Hora (€)	Total	Riesgo (%)	Beneficio (%)	Total	**214500 €**
2.1	Integración de Kafka	Integración en cloud de confluent: - Permisos. Usuarios, login/logout, securización - Persistencia - Replicación de workers - Despliegue en cloud - Registro de schemas	320	Analista (x1)	45	14400	5	5	15840	

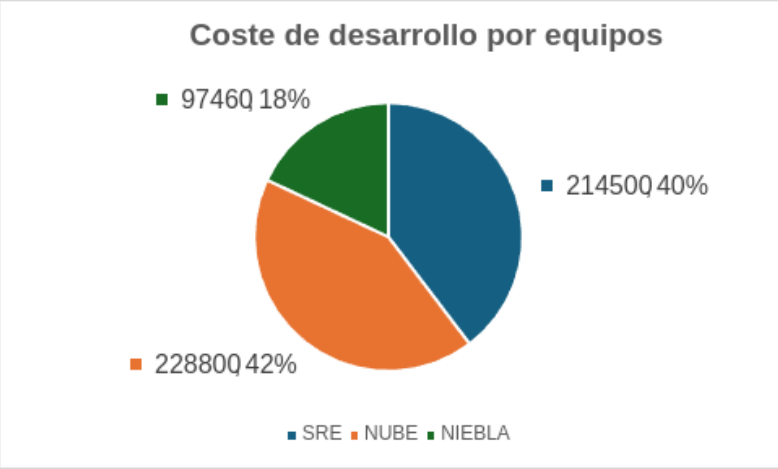

Ilustración 7 Distribución de coste de personal por equipos. Fuente: elaboración propia

En total, para contratación de recursos humanos:

$$TOTAL = 214500 + 228800 + 97460 = 540760\,€$$

3.3.7.2. Infraestructura

Desde un punto de vista de presupuesto, habría que tener en cuenta también el coste asociado al proveedor de *cloud*. En este caso en concreto, se hará una estimación inicial, para tener una ligera idea aproximada del coste fijo anual dispendiado por poder desplegar la solución en proveedores tradicionales:

Tabla 3 Coste aproximado de la infraestructura, distribuido según servicio. Fuente: elaboración propia

Servicio	*Kafka*
Comentarios	Tomando como referencia los precios de *Confluent*[11] tenemos que, para un despliegue tipo *standard,* el precio mensual se quedaría sobre los 550$/mes
Coste anual[12] (€/año)	$12 \cdot 550 \cdot 1.10 = 7260 \frac{€}{año}$
Servicio	*Cassandra*
Comentarios	Tomamos como referencia lo expuesto en GCP, con una configuración de 3 máquinas virtuales con 32 GB de memoria RAM y 20 GB de disco de arranque, con 3 discos estándar de 1000 GB para persistencia. Para esta configuración *GCP* ofrece un coste orientativo de 540,38$/mes
Coste anual (€/año)	$540{,}38 \cdot 1.10 \cdot 12 = 7133{,}02\ €/año$

[11] (Confluent Inc, 2024)

[12] Se considerará una razón de conversión monetaria EUR-USD de 1.10

Servicio	Clúster de *Kubernetes*
Comentarios	Para dimensionar el clúster en el *cloud*, lo haremos pensando que no tendremos nunca una cantidad superior a 300 *pods*. Además, utilizaremos como proveedor *GCP*, utilizando la instancia de *vCPU* más económica desde el punto de vista de despliegue por *pod*. Utilizando la calculadora que provee learnk8s, usando una instancia de vCPU *e2-highmem-16*, con un coste por *pod* de 4.49 $/mes. Esto hace que haya que usar al menos 3 instancias de *e2-highmem-16*, con una capacidad máxima de pod de 116 (por encima de los 110 por defecto de *Kubernetes)*[13]
Coste anual (€/año)	$300 \cdot 4{,}49 \cdot 12 \cdot 1{,}10 \cong 17780{,}4\ €/año$

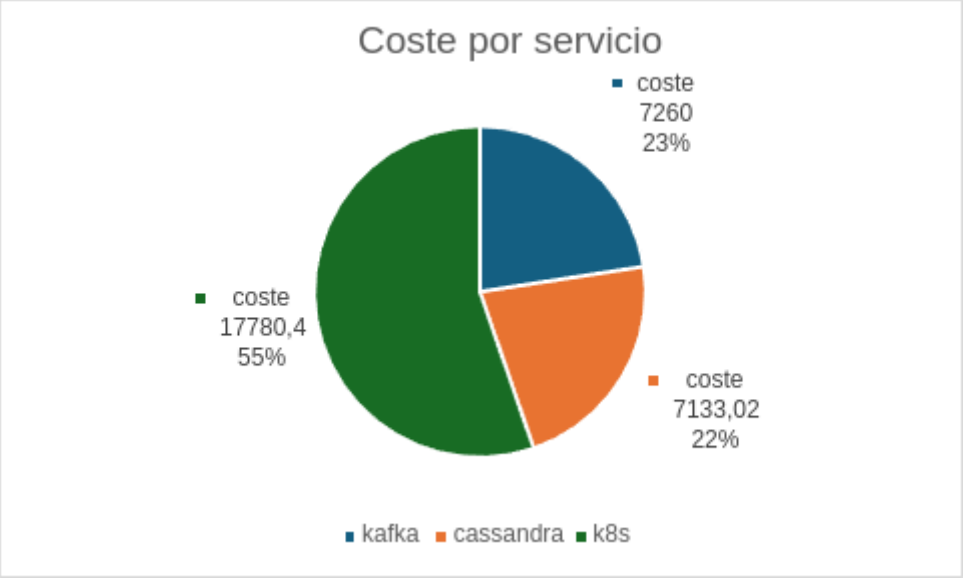

Ilustración 8 Distribución de coste por servicio. Fuente: elaboración propia

[13] (Google, 2024)

Como coste fijo, para la infraestructura, anualmente será:

$$TOTAL = 7260 + 7133{,}02 + 17780{,}4 = 32173{,}42\ €/año$$

4. DESARROLLO ESPECÍFICO DE LA CONTRIBUCIÓN

4.1. DESARROLLO DE *DEVICES*

4.1.1. Desarrollo de código Python

Para llevar a cabo este primer punto, haremos uso de la herramienta Pogo (Tango Community, 2023) para generar el esqueleto de código necesario para tener desarrollado cada *device.*

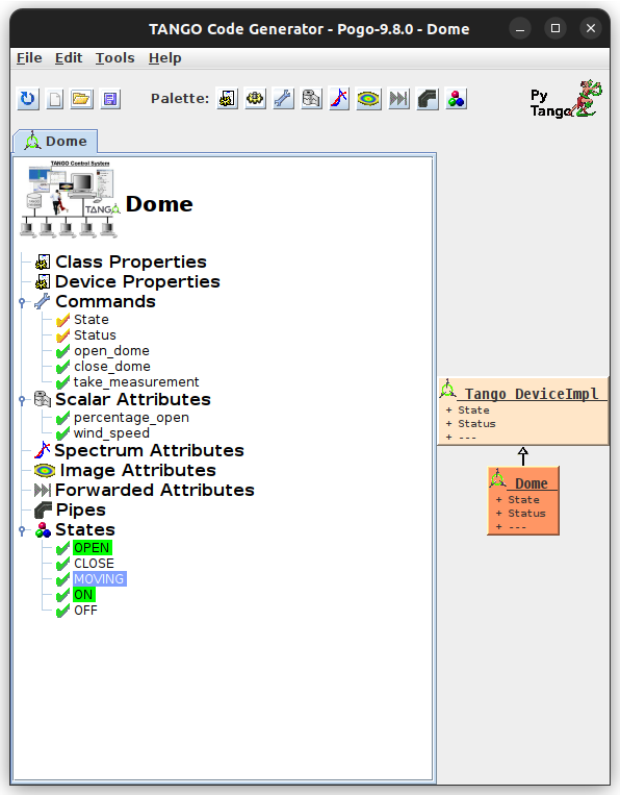

Ilustración 9 Captura de pantalla de la herramienta *Pogo* para el *device* Dome. Fuente: elaboración propia

De esta manera se define de manera sencilla los elementos que componen cada *device:*

- *Comandos*. Conjunto de operaciones que puede realizar el *device*
- *Scalar Attributes.* Propiedades del *device* que proporcionan información de este. Por ejemplo, temperatura, presión, velocidad...
- *States.* Como su nombre indica, el estado en que se encuentra el *device*. Es decir, ON, OFF, RUNNING, IDLE...

Para el caso representado en la imagen anterior, donde se esboza el *tfm-dome device,* se han definido los siguientes componentes:

Comandos	
open_dome	Abrir cúpula. Emula que se está produciendo una operación de apertura de cúpula
close_dome	Cerrar cúpula. Emula que se está llevando a cabo una operación de cierre de cúpula
take_measurement	Realiza una medición puntual del estado de la velocidad del viento en la cima de la cúpula
Scalar Attributes	
percentage_open	Porcentaje de apertura de la cúpula
wind_speed	Velocidad del vienta medida por el anemómetro en la cima de la cúpula
States	
ON, OFF, MOVING, CLOSE, OPEN	Cada estado representa una situación de la cúpula. Por ejemplo, una vez finalizado con éxito la operación *open_dome,* el estado del *device* será OPEN

A continuación, se mostrará una parte del código ya implementado para este *device:*

```
"""
Dome Device
"""


# !/usr/bin/env python
# -*- coding:utf-8 -*-
```

```
# ############################################################################
#  license :
# ============================================================================
#
#  File :        Dome.py
#
#  Project :     Dome
#
# This file is part of Tango device class.
#
# Tango is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# Tango is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with Tango.  If not, see <http://www.gnu.org/licenses/>.
#
#
#  $Author :      josue.barrera$
#
#  $Revision :    $
#
#  $Date :        $
#
#  $HeadUrl :     $
# ============================================================================
#            This file is generated by POGO
#     (Program Obviously used to Generate tango Object)
# ############################################################################

__all__ = ["Dome", "DomeClass", "main"]

__docformat__ = 'restructuredtext'

# system libraries
```

```
import sys
import os
import time
import random

# 3rd party libraries import
import tango
import PyTango

# ----- PROTECTED REGION ID(Dome.additionnal_import) ENABLED START -----#
# ----- PROTECTED REGION END -----#     //      Dome.additionnal_import

# Device States Description
# OPEN : Dome Opened
# CLOSE : Dome Closed
# MOVING : Dome going from OPEN -> CLOSE, or from CLOSE -> OPEN
# ON : Device ON
# OFF : Device ON


class Dome (PyTango.LatestDeviceImpl):
    """Dome Tango Control Device"""

    # -------- Add you global variables here --------------------------
    try:
        DEVICE_INSTANCE_CLASS = os.environ['DEVICE_INSTANCE_CLASS']
        DEVICE_INSTANCE_SERVER = os.environ['DEVICE_INSTANCE_SERVER']
        DEVICE_INSTANCE_FULLPATH = os.environ['DEVICE_INSTANCE_FULLPATH']
    except Exception as err:
        print('[%s] - %s' % type(err).__name__, err)
        sys.exit(1)

    # ----- PROTECTED REGION ID(Dome.global_variables) ENABLED START -----#

    def __init__(self, cl, name):
        PyTango.LatestDeviceImpl.__init__(self, cl, name)
        self.debug_stream("In __init__()")
        Dome.init_device(self)
        # ----- PROTECTED REGION ID(Dome.__init__) ENABLED START -----#
        # ----- PROTECTED REGION END -----#     //      Dome.__init__

    def delete_device(self):
```

```
        self.debug_stream("In delete_device()")
        # ----- PROTECTED REGION ID(Dome.delete_device) ENABLED START -----#
        # ----- PROTECTED REGION END -----#     //      Dome.delete_device

    def init_device(self):
        self.debug_stream("In init_device()")
        self.get_device_properties(self.get_device_class())
        self.attr_percentage_open_read = 0.0
        self.attr_wind_speed_read = 0.0
        # ----- PROTECTED REGION ID(Dome.init_device) ENABLED START -----#
        # ----- PROTECTED REGION END -----#     //      Dome.init_device

    def always_executed_hook(self):
        self.debug_stream("In always_excuted_hook()")
        # ----- PROTECTED REGION ID(Dome.always_executed_hook) ENABLED START -----
        # ----- PROTECTED REGION END -----#     //      Dome.always_executed_hook

    # --------------------------------------------------------------------------
    #    Dome read/write attribute methods
    # --------------------------------------------------------------------------

    def read_percentage_open(self, attr):
        self.debug_stream("In read_percentage_open()")
        # ----- PROTECTED REGION ID(Dome.percentage_open_read) ENABLED START -----
        attr.set_value(self.attr_percentage_open_read)
        return self.attr_percentage_open_read
        # ----- PROTECTED REGION END -----#     //      Dome.percentage_open_read

    def write_percentage_open(self, attr):
        self.debug_stream("In write_percentage_open()")

        # ----- PROTECTED REGION ID(Dome.percentage_open_write) ENABLED START -----#
        self.attr_percentage_open_read = attr.get_write_value()
        # ----- PROTECTED REGION END -----#     //      Dome.percentage_open_write

    def read_wind_speed(self, attr):
        self.debug_stream("In read_wind_speed()")
        # ----- PROTECTED REGION ID(Dome.wind_speed_read) ENABLED START -----
        attr.set_value(self.attr_wind_speed_read)
        return self.attr_wind_speed_read
        # ----- PROTECTED REGION END -----#     //      Dome.wind_speed_read
```

```
    def write_wind_speed(self, attr):
        self.debug_stream("In write_wind_speed()")

        # ----- PROTECTED REGION ID(Dome.wind_speed_write) ENABLED START -----#
        self.attr_wind_speed_read = attr.get_write_value()
        # ----- PROTECTED REGION END -----#     //      Dome.wind_speed_write

    def read_attr_hardware(self, data):
        self.debug_stream("In read_attr_hardware()")
        # ----- PROTECTED REGION ID(Dome.read_attr_hardware) ENABLED START -----#
        # ----- PROTECTED REGION END -----#     //      Dome.read_attr_hardware

    # --------------------------------------------------------------------
    #    Dome command methods
    # --------------------------------------------------------------------
    def open_dome(self):
        """
        Open dome command
        :rtype: None
        """
        self.debug_stream("In open_dome()")
        self.set_state(PyTango.DevState.CLOSE)

        # opening dome operation - simulation
        for i in range(100):
            time.sleep(0.5)
            self.attr_percentage_open_read = i
            self.info_stream(f"percenage open - {i}")

        self.set_state(PyTango.DevState.OPEN)
        # ----- PROTECTED REGION END -----#     //      Dome.open_dome
        self.attr_percentage_open_read = 100.0

    def close_dome(self):
        """
        close dome command
        :rtype: None
        """
        self.debug_stream("In close_dome()")
        # ----- PROTECTED REGION ID(Dome.close_dome) ENABLED START -----#
        self.set_state(PyTango.DevState.OPEN)
```

```
        # closing dome operation - simulation
        for i in reversed(range(100)):
            time.sleep(0.5)
            self.attr_percentage_open_read = i
            self.info_stream(f"percenage open - {i}")
        self.set_state(PyTango.DevState.CLOSE)
        self.attr_percentage_open_read = 0.0
        # ----- PROTECTED REGION END -----#     //      Dome.close_dome

    def take_measurement(self):
        """ take measurements from anemometer
        :return: success
        :rtype: PyTango.DevBoolean
        """
        self.debug_stream("In take_measurement()")
        argout = False

        # ----- PROTECTED REGION ID(Dome.take_measurements) ENABLED START -----#
        try:
            self.info_stream("Taking measurements. Generating mocked up data")
            self.attr_wind_speed_read = (random.randint(0.0, 150.0)
                                         + random.random())
            self.info_stream('[Wind Speed] -- %0.2f',
                             self.attr_wind_speed_read)
            argout = True
        except Exception as err:
            self.error_stream('[%s] - %s' % type(err).__name__, err)

        # ----- PROTECTED REGION END -----#     //      Dome.take_measurements
        return argout

    # ----- PROTECTED REGION ID(Dome.programmer_methods) ENABLED START -----#
    # ----- PROTECTED REGION END -----# //      Dome.programmer_methods


class DomeClass(PyTango.DeviceClass):
    # -------- Add you global class variables here --------------------------
    # ----- PROTECTED REGION ID(Dome.global_class_variables) ENABLED START -----
    # ----- PROTECTED REGION END -----# //      Dome.global_class_variables

    #  Class Properties
    class_property_list = {}
```

```
#  Device Properties
device_property_list = {}

#    Command definitions
cmd_list = {
    'open_dome':
        [
            [PyTango.DevVoid, "none"],
            [PyTango.DevVoid, "none"]
        ],
    'close_dome':
        [
            [PyTango.DevVoid, "none"],
            [PyTango.DevVoid, "none"]
        ],
    'take_measurement':
        [
            [PyTango.DevVoid, "none"],
            [PyTango.DevBoolean, "Success"],
            {
                'Polling period': "1000",
            }
        ],
    }

#    Attribute definitions
attr_list = {
    'percentage_open':
        [
            [
                PyTango.DevFloat,
                PyTango.SCALAR,
                PyTango.READ_WRITE
            ],
            {
                'label': "open percentaje",
                'unit': "%",
                'standard unit': "%",
                'display unit': "%",
                'format': "0.1f",
                'max value': "100",
```

```
                    'min value': "0",
                }
            ],
        'wind_speed':
            [
                [
                    PyTango.DevFloat,
                    PyTango.SCALAR,
                    PyTango.READ_WRITE
                ],
                {
                    'label': "Wind speed",
                    'unit': "m/seg",
                    'standard unit': "m/seg",
                    'display unit': "m/seg",
                    'format': "0.2f",
                    'min value': "0",
                }
            ],
        }


def main():
    try:
        py = PyTango.Util(sys.argv)
        py.add_class(DomeClass, Dome, 'Dome')
        U = PyTango.Util.instance()
        U.server_init()
        U.server_run()

    except PyTango.DevFailed as err:
        print('-------> Received a DevFailed exception:', err)
    except Exception as err:
        print('-------> An unforeseen exception occurred....', err)
        print('[%s] - %s' % type(err).__name__, err)
        sys.exit(1)


if __name__ == '__main__':
    try:
        dev_info = tango.DbDevInfo()
        dev_info._class = Dome.DEVICE_INSTANCE_CLASS
```

```
        dev_info.server = Dome.DEVICE_INSTANCE_SERVER
        dev_info.name = Dome.DEVICE_INSTANCE_FULLPATH

    except KeyError as err:
        print(f'-------> Received a KeyError: {err}')

    except Exception as err:
        print('[%s] - %s' % type(err).__name__, err)
        sys.exit(1)
     db = tango.Database()
    db.add_device(dev_info)
    main()
```

Con este código, se tendría desarrollado el *tfm-dome device*. Este mismo proceso ha sido replicado para cada uno de los diferentes *devices* desarrollados para este Trabajo Fin de Máster. Sin embargo, observando el código, se llega rápidamente a la conclusión de que no existe publicación de datos vía MQTT. En la práctica, **cada uno de los *devices* representa un servidor al cual se le realizan llamadas (o peticiones en realidad) CORBA sobre los comandos que han sido definidos previamente.**

Esto significa que hay que **desarrollar un cliente de cada uno de los diferentes *devices* que realice la publicación de telemetría vía protocolo MQTT**. Para ello se ha creado un módulo Python adicional que implemente esta funcionalidad. Esto se puede observar a continuación:

```
# Telemetry.py
"""
Telemetry Device Tango Controls
"""
# Standard library imports
from concurrent.futures import ThreadPoolExecutor

# 3rd party imports
from telemetry.telemetry_mqtt_connection import DeviceTelemetry

if __name__ == "__main__":

    tel = DeviceTelemetry()
    tel.connect_and_start()
```

```
    with ThreadPoolExecutor() as executor:
        tel_args = tel.load_telemetry_from_file(
            DeviceTelemetry.DEVICE_TELEMETRY_CFG_FILE
        )
        executor.map(lambda f: tel.broadcast_telemetry(**f), tel_args)

    tel.disconnect_and_stop()
```

```
# __init__.py
"""
Telemetry module
"""
# Standard libray imports



# 3rd party imports



# particular imports
from telemetry_logging import log
from telemetry_mqtt_connection import DeviceTelemetry

__all__ = [
    "log",
    "DeviceTelemetry"
]
```

```
# telemetry_mqtt_connection.py
"""
Tango Controls MQTT client connection definition
"""
# Standard libray imports
import sys
import time
import os
import json
from pathlib import Path
from datetime import datetime

# 3rd party imports
import paho.mqtt.client as mqtt
import tango
```

```
import yaml


# particular imports
from telemetry_logging import log



class DeviceTelemetry:
    """
    Telemetry class
    """


    # --------------- GLOBAL SECTION ---------------
    try:
        MQTT_KEEPALIVE_TIMEOUT = int(os.getenv(
            key="MQTT_KEEPALIVE_TIMEOUT",
            default="60"
        ))
        MQTT_ADDRESS = os.environ["MQTT_ADDRESS"]
        MQTT_PORT = int(os.environ["MQTT_PORT"])
        MQTT_USER = os.environ["MQTT_USER"]
        MQTT_PASS = os.environ["MQTT_PASS"]
        MQTT_TOPIC_ROOT = os.environ["DEVICE_INSTANCE_FULLPATH"]
        DEVICE_TELEMETRY_CFG_FILE = os.environ["DEVICE_TELEMETRY_CFG_FILE"]
    except KeyError as e:
        log.error(e)
        sys.exit(1)
    # ----------------------------------------------


    # Constructor
    def __init__(self) -> None:
        self.client_connector = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
        self.client_connector.enable_logger()


        # Setting up callbacks
        self.mqttc.on_publish = DeviceTelemetry.on_publish
        self.mqttc.on_log = DeviceTelemetry.on_log
        self.mqttc.on_connect = DeviceTelemetry.on_connect
        self.mqttc.on_disconnect = DeviceTelemetry.on_disconnect


        # Setting up the Device instance
        try:
            self.tango_dev = tango.DeviceProxy(DeviceTelemetry.MQTT_TOPIC_ROOT)
```

```
        except KeyError:
            sys.exit(1)

    # connect and start method
    def connect_and_start(self) -> None:
        """
        Set user credentials. Getting those parameters from env vars
        Then, connect to broker and the start the lopp
        """
        # Connecting to broker
        self.mqttc.username_pw_set(
            username=DeviceTelemetry.MQTT_USER,
            password=DeviceTelemetry.MQTT_PASS,
        )

        self.mqttc.connect(
            host=DeviceTelemetry.MQTT_ADDRESS,
            port=DeviceTelemetry.MQTT_PORT,
            keepalive=DeviceTelemetry.MQTT_KEEPALIVE_TIMEOUT
        )

        if self.mqttc.loop_start(): # not equal to zero means there is an error
            sys.exit(1)

    # disconnect and stop method
    def disconnect_and_stop(self) -> None:
        """
        Disconnect from broker and stop loop
        """

        if self.mqttc.is_connected():
            self.client_connector.disconnect()
            self.client_connector.loop_stop()

    # publish msg method
    def publish_msg(self,
                    attribute: str,
                    qos=2,
                    **value) -> None:
        """
        Publish a new message to telemetry topic
```

```
        : param str attribute The name of the Device Attribute
        : param int qos The quality of service
        : param dict value The value of the telemetry and its units
            See below
        : rtype None

        **value
        value : dict{
            "VALUE": int | str | float,
            "UNITS" : str
        }
        """
        # create topic
        _topic = self.get_topic(attribute)

        # build payload
        _payload = DeviceTelemetry.build_payload(
            data=value["VALUE"],
            units=value["UNITS"]
        )

        # send payload
        _msg_info = self.mqttc.publish(_topic, _payload, qos)
        _msg_info.wait_for_publish()

    # broadcast telemetry method
    def broadcast_telemetry(self,
                            attr: str,
                            units: str,
                            timeout: int | float
                            ) -> None:
        """
        publish message to broker mechanism.
        To be called through the pool executor
        : param attr - str
        : param units - str
        : param timeout - int | float
        :rtype None
        """
        while 1:
            parameter = self.tango_dev[attr]
```

```
            data = {
                "VALUE": parameter.value,
                "UNITS": units
            }
            self.publish_msg(attribute=attr, qos=1, **data)
            time.sleep(timeout)

    # load telemetry data method
    def load_telemetry_from_file(self, file: str | Path) -> tuple:
        """
        Gets the kwargs arguments to be used to
        : param file str | Path
        : rtype dict
        """
        with open(file, 'r', encoding="utf-8") as f:
            data = yaml.safe_load(f)

        t = []
        for element in data["telemetry"]:
            a = {
                "attr": element["attr"],
                "units": element["units"],
                "timeout": element["timeout"]
            }
            t.append(a)
        return tuple(t)

    # Getters and Setters
    @property
    def mqttc(self):
        """
        Getter for client connector
        """
        return self.client_connector

    # Staticmethods
    @staticmethod
    def on_connect(client, user_data, flags, rc, properties):
        """
        on connect callback method
        """
        log.debug("client %s is connected - %s", client, rc)
```

```
@staticmethod
def on_disconnect(client, user_data, flags, rc, properties):
    """
    on disconnect callback method
    """
    log.debug("client %s is disconnected - %s", client, rc)

@staticmethod
def build_payload(data: int | str | float,
                  units="NaN") -> str:
    """
    builds a json structure which will be sent
    as payload in a publish message operation
    """
    _payload = {
        "timestamp": str(datetime.now()),
        "value": data,
        "units": units
    }
    return json.dumps(_payload)

@staticmethod
def get_topic(attribute: str) -> str:
    """
    Returns the topic used to send messages
    """
    return f"{DeviceTelemetry.MQTT_TOPIC_ROOT}/telemetry/{attribute}"

@staticmethod
def on_publish(client, userdata, mid, reason_code, properties) -> None:
    """
    on publish callback function
    """
    try:
        log.debug("Client %s publishing message. mid: %d", client, mid)
    except KeyError:
        log.error(
            """on_publish() is called with a mid not present in \
                unacked_publish.""")

@staticmethod
```

```
    def on_log(client, userdata, paho_log_level, message) -> None:
        """
        on log callback function
        """
        # build message to be sent
        _msg = f"{client} - {userdata} - {message}"

        if paho_log_level in {mqtt.LogLevel.MQTT_LOG_ERR}:
            log.error(_msg)
            return None

        if paho_log_level in {mqtt.LogLevel.MQTT_LOG_DEBUG}:
            log.debug(_msg)
            return None

        if paho_log_level in {
            mqtt.LogLevel.MQTT_LOG_INFO,
            mqtt.LogLevel.MQTT_LOG_NOTICE
        }:
            log.info(_msg)
            return None

        if paho_log_level in {mqtt.LogLevel.MQTT_LOG_WARNING}:
            log.warning(_msg)
            return None
```

```
# telemetry_logging.py
"""
Telemetry logging definiton
"""
import logging
from rich.logging import RichHandler

FORMAT = "%(message)s"
logging.basicConfig(
    level="NOTSET",
    format=FORMAT,
    datefmt="[%Y %m %d %H:%M:%S]",
    handlers=[RichHandler()]
)

log = logging.getLogger("rich")
```

4.1.2. Desarrollo de contenedores Docker

A continuación, se planteará la integración del código Python desarrollado previamente. Piénsese que para cada *device* habrá que crear una imagen *Docker,* siendo este container el que encapsule todas las dependencias, configuraciones, ficheros, variables de entorno necesarias para su ejecución. Esto no es solo para el servidor, sino lo mismo para el cliente que publicará su telemetría vía MQTT.

Esto nos lleva a pensar que habrá que reducir la cantidad de código a desarrollar, reutilizando los diferentes componentes. Esto se consigue al reusar para el publicador de telemetría vía MQTT el mismo contenedor, parametrizándolo luego desde un archivo *yaml.*

```
# syntax=docker/dockerfile:1.4

ARG _base_image_="python:3.11-slim"
ARG _app_="/app"
ARG _tango_port_="10000"
ARG _port_="25000"
ARG _group_="unir"
ARG _user_="student"
ARG _group_id_="11000"
ARG _user_id_="11001"
ARG _name_="dome"

#########################
#     builder stage     #
#########################
FROM ${_base_image_} as builder
ARG _base_image_
ARG _app_
ARG _tango_port_
ARG _port_
ARG _group_
ARG _user_
ARG _group_id_
ARG _user_id_
ARG _name_

ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

WORKDIR ${_app_}

RUN apt-get update &&\
    apt-get install -y --no-install-recommends gcc

COPY --chown=${_user_}:${_group_} ./Python/${_name_}/requirements.txt .

RUN pip wheel -vvv --no-cache-dir --no-deps --wheel-dir ${_app_}/wheels -r requirements.txt

###################################
```

```
#        development stage        #
###################################
FROM ${_base_image_} as development
ARG _base_image_
ARG _app_
ARG _tango_port_
ARG _port_
ARG _group_
ARG _user_
ARG _group_id_
ARG _user_id_
ARG _name_

WORKDIR ${_app_}

COPY --from=builder ${_app_}/wheels /wheels
RUN pip install -vvv --no-cache /wheels/*

WORKDIR ${_app_}/${_name_}
COPY ./Python/${_name_}/*py .

EXPOSE ${_port_}/tcp
ENV DEVICE_PORT ${_port_}
ENV TANGO_HOST ${_tango_port_}
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1
ENV DEVICE_ORB_ENDPOINT="giop:tcp:0.0.0.0:${_port_}"
ENV DEVICE_INSTANCE_NAME="Dome"
ENV PYTHONPATH=${_app_}/${_name_}

ENTRYPOINT [ "/bin/bash" ]

###################################
###       production stage      ###
###################################
FROM ${_base_image_} as production
ARG _base_image_
ARG _app_
ARG _tango_port_
ARG _port_
ARG _group_
ARG _user_
ARG _group_id_
ARG _user_id_
ARG _name_

RUN\
addgroup --gid ${_group_id_} --system ${_group_};\
adduser --no-create-home --shell /bin/false --disabled-password --uid ${_user_id_} --system --gid
${_group_id_} ${_user_};

WORKDIR ${_app_}

COPY --from=builder --chown=${_user_}:${_group_} ${_app_}/wheels /wheels
RUN pip install -vvv --no-cache /wheels/*

WORKDIR ${_app_}/${_name_}
COPY --chown=${_user_}:${_group_} ./Python/${_name_}/*.py .
```

```
EXPOSE ${_port_}/tcp

USER ${_user_}

ENV DEVICE_ORB_ENDPOINT="giop:tcp:0.0.0.0:${_port_}"

ENV DEVICE_INSTANCE_NAME="Dome"
ENV PYTHONPATH=${_app_}/${_name_}

ENTRYPOINT [ "/bin/bash","-c" ]
CMD [ "python Dome.py ${DEVICE_INSTANCE_NAME} -ORBendPoint ${DEVICE_ORB_ENDPOINT} -v5"]
```

```
# syntax=docker/dockerfile:1.4

ARG _base_image_="python:3.11-slim"
ARG _broker_image_="emqx:5.1"
ARG _app_="/app"
ARG _group_="unir"
ARG _user_="student"
ARG _group_id_="11000"
ARG _user_id_="11001"
ARG _name_="telemetry"

#########################
#     builder stage     #
#########################
FROM ${_base_image_} as builder
ARG _base_image_
ARG _app_
ARG _group_
ARG _user_
ARG _group_id_
ARG _user_id_
ARG _name_
ARG _broker_image_

ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

WORKDIR ${_app_}

RUN apt-get update &&\
    apt-get install -y --no-install-recommends gcc

COPY --chown=${_user_}:${_group_} ./Python/${_name_}/requirements.txt .

RUN pip wheel -vvv --no-cache-dir --no-deps --wheel-dir ${_app_}/wheels -r requirements.txt

###################################
#        development stage        #
###################################
FROM ${_base_image_} as development
ARG _base_image_
ARG _app_
ARG _group_
ARG _user_
ARG _group_id_
ARG _user_id_
```

```
ARG _name_
ARG _broker_image_

WORKDIR /tmp
COPY ./Python/telemetry.yaml .

ENV DEVICE_TELEMETRY_CFG_FILE "/tmp/telemetry.yaml"

WORKDIR ${_app_}

COPY ./Python/*.py .
COPY --from=builder ${_app_}/wheels /wheels
RUN pip install -vvv --no-cache /wheels/*

WORKDIR ${_app_}/${_name_}
COPY ./Python/${_name_}/*py .

ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1
ENV PYTHONPATH=${_app_}/${_name_}
WORKDIR ${_app_}/
ENTRYPOINT [ "/bin/bash" ]

###################################
###       production stage      ###
###################################
FROM ${_base_image_} as production
ARG _base_image_
ARG _app_
ARG _group_
ARG _user_
ARG _group_id_
ARG _user_id_
ARG _name_
ARG _broker_image_

RUN\
addgroup --gid ${_group_id_} --system ${_group_};\
adduser --no-create-home\
--shell /bin/false\
--disabled-password\
--uid ${_user_id_}\
--system\
--gid ${_group_id_} ${_user_};

WORKDIR /tmp
COPY ./Python/telemetry.yaml .

ENV DEVICE_TELEMETRY_CFG_FILE "/tmp/telemetry.yaml"

WORKDIR ${_app_}

COPY ./Python/*.py .
COPY --from=builder --chown=${_user_}:${_group_} ${_app_}/wheels /wheels
RUN pip install -vvv --no-cache /wheels/*

WORKDIR ${_app_}/${_name_}
COPY --chown=${_user_}:${_group_} ./Python/${_name_}/*.py .
```

```
WORKDIR ${_app_}
USER ${_user_}

ENV PYTHONPATH=${_app_}/${_name_}

ENTRYPOINT [ "/bin/bash","-c" ]
CMD [ "python Telemetry.py"]

#########################
#      broker stage     #
#########################
FROM ${_broker_image_} as broker
ARG _base_image_
ARG _app_
ARG _group_
ARG _user_
ARG _group_id_
ARG _user_id_
ARG _name_
ARG _broker_image_
```

Estos dos *Containerfiles* mostrados justo encima están pensados para ser construidos en un modo multietapa. De esta manera, conseguimos una optimización del espacio necesario para crear la imagen, siendo una práctica recomendada en la industria[14]. Realmente, también buscamos que en cierto modo la manera de trabajar en producción y desarrollo difieran en cuanto a lo desplegado en cada momento.

Esto puede parecer contraproducente, ya que se podría pensar que se estaría desarrollando por partida doble. En la práctica, esto no es así porque se trabaja con *devcontainers*[15]. De esta manera, se ha conseguido separar el desarrollo del código del despliegue de este en un entorno de producción. **Para ello, lo que hacemos es crear dos contenedores diferentes: uno para entorno de desarrollo y otro para entorno de producción,** lo cual se logra gracias a que hemos definido un *multistage build.*

```
// For format details, see https://aka.ms/devcontainer.json. For config options, see the
// README at: https://github.com/devcontainers/templates/tree/main/src/alpine
{
    "name": "development",
```

[14] (Docker Inc., 2024)

[15] (Microsoft Corporation, 2024)

```
    "dockerComposeFile" : [
        "../src/Docker/docker-compose.development.yaml"
    ],
    "service": "device",
    "shutdownAction": "stopCompose",
    "workspaceFolder": "/app/",
    "customizations": {
        "vscode": {
            "extensions": [
                "ms-python.python",
                "ms-python.pylint",
                "ms-python.isort",
                "ms-python.debugpy",
                "ms-python.flake8"
            ]
        }
    },

// Features to add to the dev container. More info: https://containers.dev/features.
// "features": {},

// Use 'forwardPorts' to make a list of ports inside the container available locally.
"forwardPorts": [ 25000 ],

// Use 'postCreateCommand' to run commands after the container is created.
// "postCreateCommand": "uname -a",

// Configure tool-specific properties.
// "customizations": {},

// Uncomment to connect as root instead. More info: https://aka.ms/dev-containers-non-root.
"remoteUser": "root"
```

```
}
```

4.1.3. Desarrollo de Helm Charts

Una vez desarrollados tanto el código Python necesario para integrar la funcionalidad que corresponde a cada *device,* como el *containerfile* que permite construir la imagen *Docker,* el siguiente punto es definir el despliegue a nivel de clúster de *Kubernetes.*

Para ello habrá que definir la *Helm Chart* que realizará dicha función. Esta Helm Chart construirá siempre, al menos:

- Un servicio tipo *ClusterIP.* De esta manera, el *device* será accesible vía llamadas *CORBA en el p.*
- Un *StatefulSet,* donde se encuentren los dos contenedores desarrollados previamente: el que corresponde al *device* en sí mismo y el que corresponde al cliente que publica telemetría vía MQTT.
- Algún elemento adicional, como un *ServiceAccount* o uno o varios *Configmaps/Secrets*, lo cual dependerá de las necesidades particulares de cada *device.*

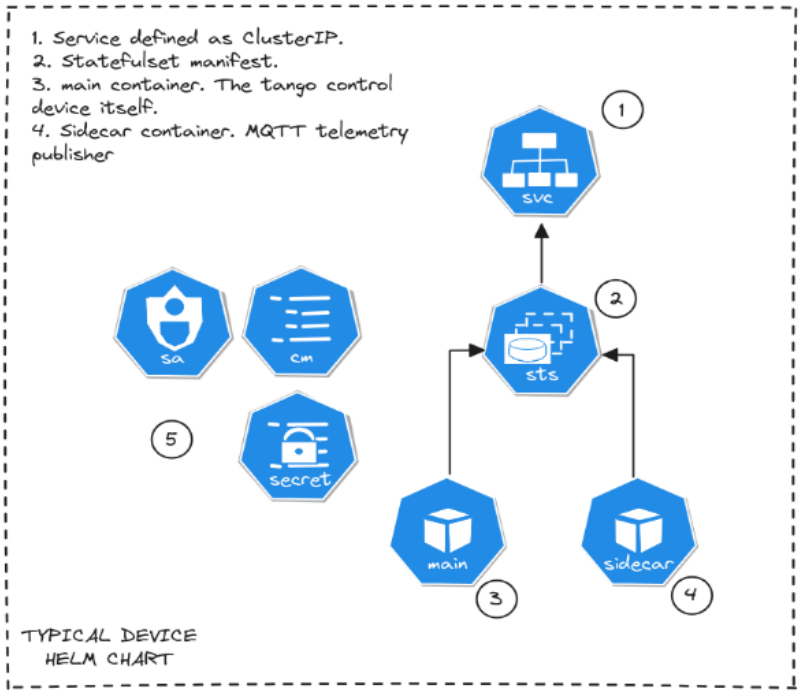

Ilustración 10 Objetos desplegados en un *Device Helm Chart*[16]. Fuente: elaboración propia

Todo esto hay que verlo en su contexto, como será explicado en el apartado **¡Error! No se encuentra el origen de la referencia.**, se reutiliza la *Helm Chart* creada para el *tfm-device-template* como biblioteca en cada *device,* de esta manera se evita el tener que reescribir siempre el mismo código en cada uno de ellos. Hay que citar también que el archivo de definición de parámetros está pensado para poder integrar los parámetros necesarios para usar esta biblioteca. A continuación, se mostrará partes del código desarrollado para las *Helm Charts*, concretamente los archivos '*manifests.yaml'* del *device tfm-dome*, la plantilla del tfm-device-template '*_statefulset.yaml'* y el archivo de parámetros del *tfm-dome* '*Values.yaml'*:

```
# tfm-dome. manifests.yaml

---

{{- include "tfm-device-template.configmap-envs-vars.tpl" $ }}

---

{{- include "tfm-device-template.serviceaccount.tpl" $ }}

---

{{- include "tfm-device-template.service.tpl" $ }}

---

{{- include "tfm-device-template.statefulset.tpl" $ }}

---

{{- include "tfm-device-template.configmap-telemetry.tpl" $ }}

---

{{- include "tfm-device-template.configmap-mqtt-parameters.tpl" $ }}

---

{{- include "tfm-device-template.secrets-mqtt-parameters.tpl" $ }}
```

```
# tfm-device-template. _statefulset.yaml

{{- define "tfm-device-template.statefulset.tpl" -}}

apiVersion: apps/v1

kind: StatefulSet
```

[16] Se puede apreciar como existen dos contenedores desplegados en el *statefulset*, uno donde está la funcionalidad propiamente relacionada con el *device*, y el otro que hace las veces de publicador de telemetría hacia el bróker MQTT

```
metadata:
  name: {{ include "device.sts" $ }}
  labels:
    {{- include "device.labels" $ | nindent 4 }}
  annotations:
    {{- include "device.ArgocdWaveLabel" $ | nindent 4 }}
spec:
  selector:
    matchLabels:
      {{- include "device.selectorLabels" $ | nindent 6 }}
  serviceName: {{ include "device.svc_name" $ }}
  template:
    metadata:
      labels:
        {{- include "device.selectorLabels" $ | nindent 8 }}
    spec:
      serviceAccountName: {{ include "device.serviceAccountName" $ }}
      securityContext:
        {{- toYaml $.Values.device.securityContext | nindent 8 }}
      volumes:
      - name: {{ include "companion-telemetry.volume.name" $ }}
        configMap:
          name: {{ include "device-telemetry.configmap" $ }}
      containers:
        - name: {{ include "device.container" $ }}
          securityContext:
            {{- toYaml $.Values.device.podSecurityContext | nindent 12 }}
          image: {{ include "device.image" $ }}
          imagePullPolicy: {{ $.Values.device.main.image.pullPolicy | quote }}
          ports:
            - containerPort: {{ $.Values.device.main.svc.port }}
              name: {{ include "device.corba_servicename" $ }}
```

```
          resources:
            {{- toYaml $.Values.device.resources | nindent 12 }}
          envFrom:
            - configMapRef:
                name: {{ $.Values.global.databaseds.configmaps }}
            - configMapRef:
                name: {{ include "device.configmap" $ }}
        {{- if $.Values.device.companion.enabled }}
        - name: {{ include "companion.container" $ }}
          securityContext:
            {{- toYaml $.Values.device.podSecurityContext | nindent 12 }}
          image: {{ include "companion.image" $ }}
          imagePullPolicy: {{ $.Values.device.companion.image.pullPolicy | quote }}
          resources:
            {{- toYaml $.Values.device.resources | nindent 12 }}
          envFrom:
            - configMapRef:
                name: {{ $.Values.global.databaseds.configmaps }}
            - configMapRef:
                name: {{ include "device.configmap" $ }}
            - configMapRef:
                name: {{ include "device-mqtt-parameters.configmap" $ }}
            - secretRef:
                name: {{ include "device-mqtt-parameters.secrets" $ }}
          volumeMounts:
            - name: {{ include "companion-telemetry.volume.name" $ }}
              mountPath: {{ include "device.telemetry_cfg_mountpath" $ }}
        {{- end }}
  volumeClaimTemplates: []
{{- end }}
```

```
# tfm-dome. Values.yaml
########################################
```

```
###             GLOBAL              ###
#########################################
global:
  databaseds:
    configmaps: tango-databaseds-configmaps
  mqtt:
    address: "emqx-listeners.fog-layer.svc.cluster.local"
    port: "1883"
    user: "admin"
    pass: "public"


#########################################
###             DEVICE              ###
#########################################
device:
  nameOverride: dome
  fullnameOverride: dome


  tango:
    domain: Telescope
    family: Dome
    member: dome


  serviceAccount:
    create: true
    annotations: {}
    name: tfm-tango-dome


  argocdSyncWave: 30


  podSecurityContext: {}
    # fsGroup: 2000
```

```
securityContext: {}
  # capabilities:
  #   drop:
  #   - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000


resources: # {}
  # We usually recommend not to specify default resources and to leave this as a conscious
  # choice for the user. This also increases chances charts run on environments with little
  # resources, such as Minikube. If you do want to specify resources, uncomment the following
  # lines, adjust them as necessary, and remove the curly braces after 'resources:'.
  limits:
    cpu: 100m
    memory: 128Mi
  requests:
    cpu: 100m
    memory: 128Mi


telemetry:
  - attr: wind_speed
    units: "m/seg"
    timeout: 10.0
main:
  image:
    registry: unir-tfm
    repository: devices/dome
    tag: 0.0.6-production
    pullPolicy: IfNotPresent
  svc:
```

```
    port: 25000


companion:
  enabled: true
  image:
    registry: unir-tfm
    repository: devices/telemetry
    tag: 0.0.6-production
    pullPolicy: IfNotPresent
```

4.2. DEFINICIÓN DE LA PRUEBA DE CONCEPTO

Para la contribución particular de este trabajo de fin de máster, se ha pensado en el planteamiento de una solución o prueba de concepto *(PoC)* desplegada en local. La idea es definir el despliegue de tal manera que su migración a otra plataforma sea lo más sencilla posible.

En orden de conseguir este nivel de migrabilidad, lo importante es que la aplicación sea desplegable en un clúster de *Kubernetes*. Para estandarizar este despliegue, se han utilizado *Helm Charts,* parametrizando de esta manera todo el proceso. Piénsese que para los servicios generalistas como *MQTT*, *Kafka*, *Cassandra*, *Grafana...,* ya existen disponibles de manera gratuita las *Helm Charts* correspondiente a cada uno de estos servicios, con lo que su despliegue en cualquier clúster es relativamente sencillo de disponer.

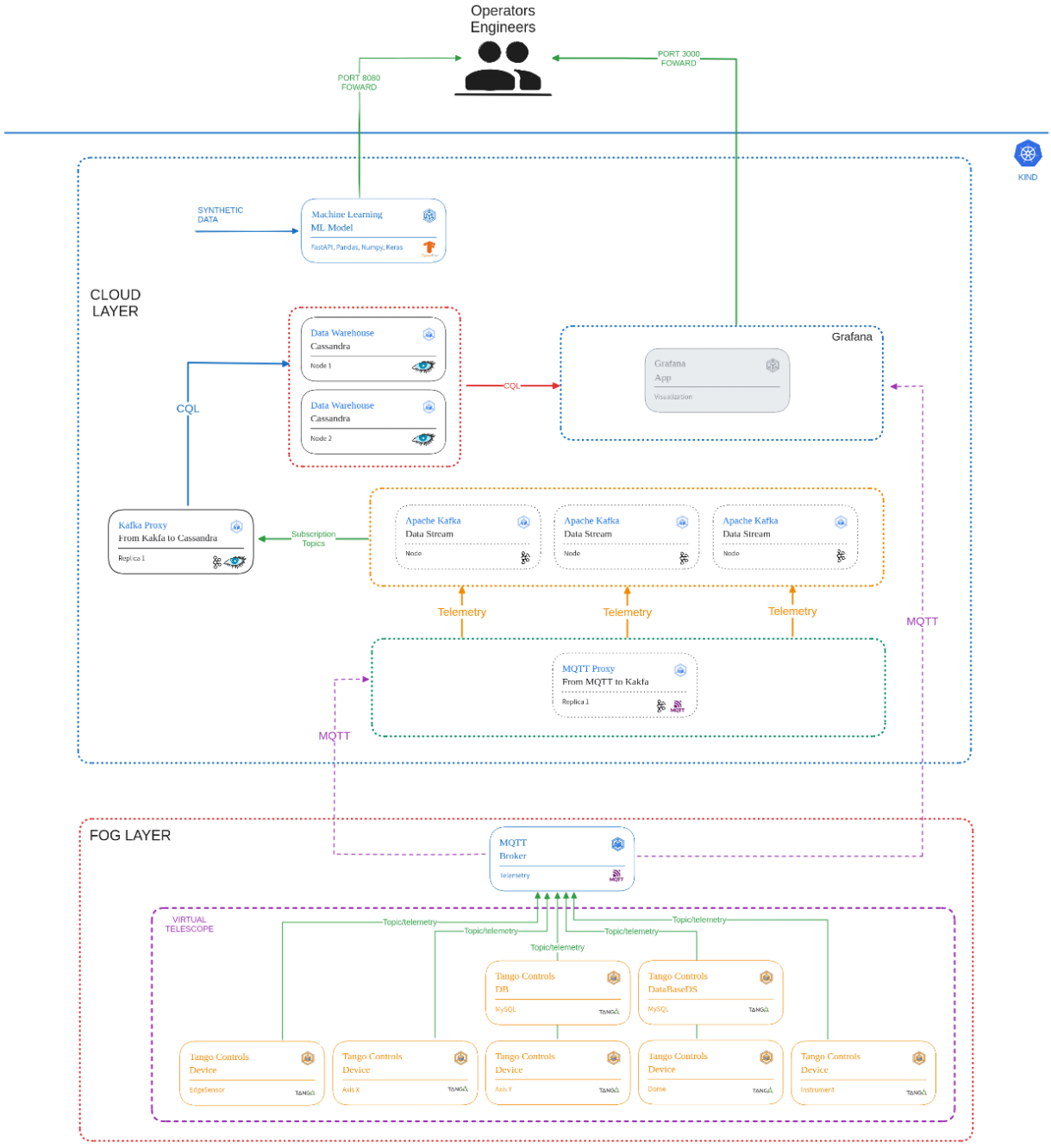

Ilustración 11 Despliegue de *PoC*. Fuente: elaboración propia

Sin embargo, esta situación no es así para el resto de los servicios planteados, sobre todo en lo relativo al framework *Tango Controls.* Esto ha supuesto el desarrollo del código de cada una de las *Helm Charts* necesarias para cada uno de los distintos *Tango Controls devices.*

El planteamiento de este tipo de *Helm Charts* se ha dispuesto con el desarrollo e integración de una *Chart* genérica, que representa el esqueleto real de cada uno de los futuros *devices* a generar. Es decir, la solución demuestra cómo se ha de reutilizar código integrando esta *Generic Device Helm Chart* como dependencia de otras (Barrera, 2024b). Esto se ve claramente en el siguiente *code snippet,* el cual representa un extracto de la *Helm Chart* desarrollada para el *Dome Device* (Barrera, 2024c), que representa una abstracción de la cúpula del telescopio:

```
apiVersion: v2
name: tfm-dome
description: |-
  Universidad Internacional de La Rioja (www.unir.net)
  Master in Internet Of Things
  Tango-Controls Dome Device Chart

# A chart can be either an 'application' or a 'library' chart.
#
# Application charts are a collection of templates that can be packaged into versioned archives
# to be deployed.
#
# Library charts provide useful utilities or functions for the chart developer. They're included as
# a dependency of application charts to inject those utilities and functions into the rendering
# pipeline. Library charts do not define any templates and therefore cannot be deployed.
type: application

# This is the chart version. This version number should be incremented each time you make changes
# to the chart and its templates, including the app version.
# Versions are expected to follow Semantic Versioning (https://semver.org/)
version: "0.0.6"

# This is the version number of the application being deployed. This version number should be
# incremented each time you make changes to the application. Versions are not expected to
# follow Semantic Versioning. They should reflect the version the application is using.
# It is recommended to use it with quotes.
appVersion: "0.0.6"
kubeVersion: ">= v1.27.0"
```

```
keywords:
  - unir
  - robotic telescope
  - IoT
  - telescope
icon: https://estudiar.unir.net/wp-content/uploads/2022/01/Unir_2021_logo_color.svg
home: www.unir.net
sources:
  - https://github.com/jbarrera-lz/tfm-dome
maintainers:
  - name: jbarrera-lz
    email: josue.barrera551@comunidadunir.net



dependencies:
  - name: tfm-device-template
    version: "x.x.x"
    repository: "file://../../../tfm-device-template/Helm/tfm-device-template
```

Se tiene que pensar que, al utilizar este tipo de enfoque a la hora de plantear el desarrollo de la solución, en realidad se busca que en algún momento se pueda dar el salto hacia la automatización del proceso de compilación con la ejecución de pipelines de integración continua. Este objetivo queda fuera del alcance del este Trabajo Fin de Máster, si bien sería algo a implementar en una próxima iteración de la solución.

Del mismo modo que se ha comentado que en un futuro habría que, como paso natural, automatizar el proceso de construcción de los *devices* (esto es, generación de imágenes *Docker* y de *Helm Charts* en lo que se denomina *build stage en* integración continua), otro paso natural sería la automatización del despliegue de cada una de las distintas capas (o *layers*, tal y como han sido definidas Ilustración 6 Diagrama por capas de la aplicación. Fuente: elaboración propia) de la solución. Esta implementación se realizaría usando *Argo CD* como herramienta. Principalmente se definiría una aplicación de aplicaciones (*App of Apps patter design*) (Anderson, 2023) por el equipo de SER, la cual plantearía el despliegue automatizado de la solución.

Una vez hecho estos comentarios sobre las *Helm Charts* generadas, continuaremos con la definición de la prueba de concepto. En nuestro caso, usaremos *Kind* como herramienta

utilizada para levantar un clúster de *Kubernetes* en local. Aunque es una herramienta de uso recomendado únicamente para entornos de desarrollo, es lo suficientemente versátil para desplegar nuestra prueba de concepto. A la hora de definir cada una de las dos capas (*fog* y *cloud layers*) lo haremos simplemente con la creación de una serie de espacios de nombres (o *namespaces,* si usamos terminología propia de *Kubernetes*), desplegando sobre los mismos según interese para cada capa. De esta manera, se simula una situación real de separación de recursos, aunque es cierto que en producción serían dos clústeres desplegados físicamente de manera independiente. También es importante citar que, a efectos prácticos, en lugar de definir un *Ingress* asociado a un servicio tipo *LoadBalancer*, se ha optado por una solución apropiada solo para entornos de desarrollo, como es el hacer un *port forward* del servicio de *Grafana*, exponiendo esta funcionalidad hacia *localhost* en el puerto 3000 (accesible por tanto desde nuestro navegador).

Resumiendo, para nuestra prueba de concepto se han tenido en cuenta las siguientes consideraciones:

- *Kind* como herramienta para crear el clúster
- Espacios de nombre separados, el primero llamado *fog-layer* y el segundo *cloud-layer*.
- *Port forward* de los servicios que tienen que ser accesibles desde fuera del clúster. Únicamente y exclusivamente el servicio de *Grafana.*

4.3. DESPLIEGUE DE LA PRUEBA DE CONCEPTO

En este apartado explicaremos punto por punto el proceso que hay que seguir para poder desplegar toda la vertical planteada para la prueba de concepto. Como se comentó antes, este procedimiento sería susceptible de automatizarse en iteraciones posteriores.

Para todo esto, hace falta de disponer previamente de una configuración básica en la máquina utilizada por parte del desarrollador:

Tabla 4 Recursos utilizados durante la prueba de concepto

Recurso	Comentarios

Máquina Linux	En el presente desarrollo, se ha usado *Ubuntu 22.04 LTS*
Docker y docker compose	En sus versiones *27.1.2* y *v2.29.1* respectivamente
Kubectl	Interfaz de línea de comandos para *Kubernetes*, en su versión *v1.30.4*
Helm. Gestor de dependencias de Kubernetes	En su versión *v3.15.4*
k9s	Herramienta bastante potente de línea de comandos que se utiliza para gestionar clústeres de *Kubernetes*. En este caso en concreto, se está usando la versión *v0.32.4*
Kubeselect	Herramienta de consola que se usa para cambiar de contextos a la hora de seleccionar diferentes clústeres sobre los que trabajar (Breitfelder, 2024)

4.3.1. Creación de clúster

Tabla 5 Pasos del proceso de creación de clúster de *Kubernetes* usando *Kind*

Kind Cluster

`$ cd /path/to/my/workspace/folder`
`$ git clone –recurse-submodules https://github.com/jbarrera-lz/tfm-deployment.git`
`$ cd tfm-deployment/k8ssandra-operator`
`$ ./script/setup-kind-multicluster.sh --clusters 1 –kind-worker-nodes 4`

Una vez finalizado el proceso, ya disponemos de nuestro clúster en local con *Kind*. Usamos *kubeselect* para movernos hacia el contexto de nuestro recién creado clúster.
`$ kubeselect kind-k8ssandra-0`

4.3.2. Instalación de Helm Charts

Tabla 6 Pasos a seguir para el despliegue de Helm Charts

Despliegue de Helm Charts
Para desplegar el clúster, utilizaremos las directrices expuestas en la web de k8ssandra(Bradford & Garnsey, 2024).

$ helm repo add k8ssandra https://helm.k8ssandra.io/stable
$ helm repo add jetstack https://charts.jetstack.io
$ helm repo add emqx https://repos.emqx.io/charts
$ helm repo add bitnami https://charts.bitnami.com/bitnami
$ helm repo add grafana https://grafana.github.io/helm-charts
$ helm repo add apache-airflow https://airflow.apache.org
$ helm repo update

De esta manera se ha sincronizado nuestro cliente de *Helm* con una serie de repositorios públicos utilizados en nuestro trabajo. Ahora, procedemos a utilizar el código propio creado para el este trabajo de fin de máster.

$ cd /path/to/my/workspace/folder/tfm-deployment
$ helm install cert-manager jetstack/cert-manager \
--namespace cert-manager –-create-namespace --set installCRDs=true
$ helm install k8ssandra-operator k8ssandra/k8ssandra-operator \
--namespace cloud-layer –-create-namespace
$ cd Helm/
$ helm install –-namespace cloud-layer –-create-namespace grafana tfm-cloud-umbrella/
$ helm install –-namespace fog-layer –-create-namespace emqx tfm-fog-umbrella/

Procedemos a instalar diferentes *charts* usando *Helm*.

$ cd /path/to/my/workspace/folder/tfm-deployment
$ helm install cert-manager jetstack/cert-manager \
--namespace cert-manager –-create-namespace --set installCRDs=true
$ helm install k8ssandra-operator k8ssandra/k8ssandra-operator \
--namespace cloud-layer –-create-namespace
$ cd Helm/
$ helm install –-namespace cloud-layer –-create-namespace grafana tfm-cloud-umbrella/
$ helm install –-namespace fog-layer –-create-namespace emqx tfm-fog-umbrella/

Con estos dos últimos comandos, hemos procedido a instalar *Grafana*, *Argocd*, *Apache Airflow* y el operador de *Emqx*. Si bien, a efectos prácticos tanto *Argocd* como *Apache Airflow* no son necesarios para la *PoC*, su presencia no significa que se vaya a perder funcionalidad. Finalmente hay que disponer de los clústeres de *Emqx*, *Kafka* y *Cassandra*.

```
$ cd /path/to/my/workspace/folder/tfm-deployment
$ kubectl apply –n fog-layer -f emqx/emqx.yaml
$ kubectl apply –n cloud-layer –f kafka/kafka-deployment.yaml
$ kubectl apply –n cloud-layer –f k8ssandra/k8ssandracluster.yaml
```

Llegados a este punto, hemos conseguido desplegar un clúster de *Kubernetes* con *Kind*, *Grafana* para visualización, *Apache Cassandra*, *Apache Kakfa* y *EMQX* cómo bróker MQTT.

4.3.3. Despliegue de Tango-Controls

Desplegaremos sobre la capa *fog* los siguientes elementos:

- Simulación de la cúpula con el *device tfm-dome.*
- Simulación de un *controlador* para el movimiento sobre el eje X, usando el *device tfm-driver-axis-x.* Es decir, estamos simulando el Azimut del telescopio.
- Simulación de un *controlador* para el movimiento sobre el eje Y, usando el d*evice tfm-driver-axis-y*. En realidad, con esta abstracción, lo que se está simulando es la declinación o altura del telescopio.
- Un instrumento simulado, el cual es capaz de generar archivos *fits*. Esto es vital para simular el funcionamiento real de un telescopio, ya que el objetivo final es poder adquirir imágenes cuando se está en observación.
- Simulación de un sensor de borde. De esta manera se monitoriza el estado de *cofaseamiento* entre distintos segmentos del espejo primario.

La idea, es que todos estos dispositivos generen automáticamente telemetría que es enviada hacia la capa *cloud* mediante publicación en un bróker *MQTT*. Dicha generación de telemetría hacia el bróker se hace de manera periódica por cada uno de los diferentes *devices* desplegados. Estos despliegues se harán utilizando sus respectivas Helm Charts, las cuales son una colección de objetos de *Kubernetes* que representarán un *device* de *Tango Controls.*

Procedemos ahora a desplegar los distintos *devices* de *Tango-Controls*. Para cada uno de ellos, tendremos que construir las imágenes *Docker*, y además habrá que subir las mismas al clúster *Kind,* evitando de esta manera tener errores del tipo *ImagePullBackOff*.

Tabla 7 Despliegue de Tango Control *devices*

tfm-device-template (Barrera, 2024b)

`$ cd /path/to/my/workspace/folder`
`$ git clone –recurse-submodules https://github.com/jbarrera-lz/tfm-device-template.git`
`$ cd tfm-device-template/Docker/`
`$ docker compose up –d –build`

Una vez finalizado el proceso de construcción de la imagen, la subimos a nuestro clúster de *Kind*
`$ docker compose down` `$ kind load docker-image unir-tfm/devices/telemetry:0.0.6-production --name k8ssandra-0`
tfm-tango-controls (Barrera, 2024j)

`$ cd /path/to/my/workspace/folder`
`$ git clone –recurse-submodules https://github.com/jbarrera-lz/tfm-tango-controls.git`
`$ cd tfm-tango-controls/Docker/`
`$ docker compose up –d –build`

Una vez finalizado el proceso de construcción de la imagen, la subimos a nuestro clúster de *Kind*:

`$ docker compose down`
`$ kind load docker-image unir-tfm/tango-controls-databaseds:0.0.1`
`$ kind load docker-image unir-tfm/tango-controls-db:0.0.1`
`$ cd /path/to/my/workspace/folder/tfm-tango-control`
`$ helm install –n fog-layer tango Helm/`

tfm-dome (Barrera, 2024c)

`$ cd /path/to/my/workspace/folder`
`$ git clone –recurse-submodules https://github.com/jbarrera-lz/tfm-dome.git`
`$ cd tfm-dome/src/Docker/`
`$ docker compose up –d –build`

Una vez finalizado el proceso de construcción de la imagen, la subimos a nuestro clúster de *Kind*:

`$ docker compose down`
`$ kind load docker-image unir-tfm/devices/dome:0.0.6-production`
`$ cd /path/to/my/workspace/folder/tfm-dome/src/Helm`
`$ helm dependency update tfm-dome/`
`$ helm install –n fog-layer dome tfm-dome/`

tfm-driver-axis-x (Barrera, 2024d)

```
$ cd /path/to/my/workspace/folder
$ git clone –recurse-submodules https://github.com/jbarrera-lz/tfm-driver-axis-x.git
$ cd tfm-driver-axis-x/src/Docker/
$ docker compose up –d –build
```

Una vez finalizado el proceso de construcción de la imagen, la subimos a nuestro clúster de *Kind*:

```
$ docker compose down
$ kind load docker-image unir-tfm/devices/driver-x-device:0.0.6-production
$ cd /path/to/my/workspace/folder/tfm-driver-axis-x/src/Helm
$ helm dependency update driver-axis-x/
$ helm install –n fog-layer x driver-axis-x/
```

tfm-driver-axis-y (Barrera, 2024e)

```
$ cd /path/to/my/workspace/folder
$ git clone –recurse-submodules https://github.com/jbarrera-lz/tfm-driver-axis-y.git
$ cd tfm-driver-axis-y/src/Docker
$ docker compose up –d –build
```

Una vez finalizado el proceso de construcción de la imagen, la subimos a nuestro clúster de *Kind*

```
$ docker compose down
$ kind load docker-image unir-tfm/devices/driver-y-device:0.0.6-production
```

Para a continuación desplegarlo contra el clúster, en el *namespace* fog-layer

```
$ cd /path/to/my/workspace/folder/tfm-driver-axis-y/src/Helm
$ helm dependency update driver-axis-y/
$ helm install –n fog-layer y driver-axis-y/
```

tfm-edge-sensor (Barrera, 2024f)

```
$ cd /path/to/my/workspace/folder
$ git clone –recurse-submodules https://github.com/jbarrera-lz/tfm-edge-sensor.git
$ cd tfm-edge-sensor/src/Docker
$ docker compose up –d –build
```

Una vez finalizado el proceso de construcción de la imagen, la subimos a nuestro clúster de Kind

```
$ docker compose down
$ kind load docker-image unir-tfm/devices/edge-sensor:0.0.2-production
```

Y ahora la desplegamos contra el clúster, en el namespace fog-layer

```
$ cd /path/to/my/workspace/folder/tfm-edge-sensor/src/Helm
$ helm dependency update edge-sensor/
$ helm install –n fog-layer edge-1-m1 edge-sensor/
```

tfm-simulated-instrument (Barrera, 2024i)

```
$ cd /path/to/my/workspace/folder
$ git clone –recurse-submodules https://github.com/jbarrera-lz/tfm-simulated-instrument.git
$ cd tfm-simulated-instrument/src/Docker
$ docker compose up –d –build
```

Una vez finalizado el proceso de construcción de la imagen para el instrumento simulado, la subimos a nuestro clúster de Kind

```
$ docker compose down
$ kind load docker-image unir-tfm/devices/simulated-instrument:0.0.3-production
$ cd /path/to/my/workspace/folder/tfm-simulated-instrument/src/Helm
$ helm dependency update SimulatedInstrument/
$ helm install –n fog-layer siminstrument SimulatedInstrument/
```

tfm-proxy-mqtt-kafka (Barrera, 2024h)

```
$ cd /path/to/my/workspace/folder
$ git clone –recurse-submodules https://github.com/jbarrera-lz/tfm-proxy-mqtt-kafka.git
$ cd tfm-proxy-mqtt-kafka/src/Docker
$ docker compose up –d –build
```

Una vez finalizado el proceso de construcción de la imagen para el proxy mqtt kafka, la subimos a nuestro clúster de *Kind*

```
$ docker compose down
$ kind load docker-image unir-tfm/devices/proxy-mqtt-kafka:0.0.8-production
$ cd /path/to/my/workspace/folder/tfm-proxy-mqtt-kafka/src/Helm
$ helm dependency update proxy_mqtt_kafka/
$ helm install –n cloud-layer m-to-k proxy_mqtt_kafka/
```

tfm-kafka-cassandra-proxy (Barrera, 2024g)

```
$ cd /path/to/my/workspace/folder
$ git clone –recurse-submodules https://github.com/jbarrera-lz/tfm-kafka-cassandra-proxy.git
$ cd tfm-kafka-cassandra-proxy/src/Docker
$ docker compose up –d –build
```

Una vez finalizado el proceso de construcción de la imagen para el proxy Kafka Cassandra, la subimos a nuestro clúster de *Kind*

$ docker compose down
$ kind load docker-image unir-tfm/devices/kafka-k8ssandra-proxy:0.0.13-production
$ cd /path/to/my/workspace/folder/tfm-kafka-cassandra-proxy/src/Helm
$ helm dependency update Kafka_cassandra_proxy/
$ helm install –n cloud-layer k-to-k Kafka_cassandra_proxy/

Al desplegar estos dos *proxies*, hemos conseguido que los datos se muevan desde los dispositivos de *Tango-Controls*, hasta la persistencia en *Cassandra*. De esta manera, están accesibles para su consumo por lotes, o por *streaming*, teniendo de esta manera completada las dos ramas de una arquitectura canónica lambda.

4.3.4. Instalación de modelo de TensorFlow

Tabla 8 Para a seguir durante el despliegue de modelo de Machine Learning

tfm-telescope-ml (Barrera, 2024k)

$ cd /path/to/my/workspace/folder
$ git clone –recurse-submodules https://github.com/jbarrera-lz/tfm-telescope-ml.git
$ cd tfm-telescope-ml/src/Docker
$ docker compose up –d –build

Una vez finalizado el proceso de construcción de la imagen para el instrumento simulado, la subimos a nuestro clúster de *Kind*. Este proceso de creación de la imagen Docker se realizará en un *Dockerfile multistage* – de varias etapas -. De esta manera, se genera el modelo de ML y se exporta como archivo *pkl,* para luego ser inyectado en una etapa posterior al contenedor final. Dicho contenedor final, es una aplicación accesible mediante API REST gracias a la utilización de la librería *FastAPI*, realizando una petición a dicho modelo y proporcionando de vuelta la respuesta de este.

$ docker compose down
$ kind load docker-image unir-tfm/devices/telescope-ml:0.0.1-production
$ cd /path/to/my/workspace/folder/tfm-telescope-ml/src/Helm
$ helm dependency update tfm-telescope-ml/
$ helm install –n cloud-layer ml tfm-telescope-ml/

A la hora de plantear el despliegue del modelo de predicción, se ha planteado un proceso donde los diferentes componentes son primeramente integrados a nivel de *Dockerfile,* y luego

son lanzados en el clúster vía *Helm Chart.* Es el mismo mecanismo utilizado durante el desarrollo de la aplicación.

En este sentido, hay que reseñar que, en una fase de desarrollo, como primer paso se ha utilizado *Jupyter notebooks* para la generación del modelo, y luego se ha exportado el código a una aplicación Python, la cual exporta el modelo como un archivo *pkl* (utilizando la librería *Pickle*[17] para ello). Para poder acceder al modelo y realizar peticiones al mismo, se ha desplegado una aplicación accesible vía http, conseguido gracias a la integración de *FastAPI* y modelos de *Machine Learning* (Machado, 2023).

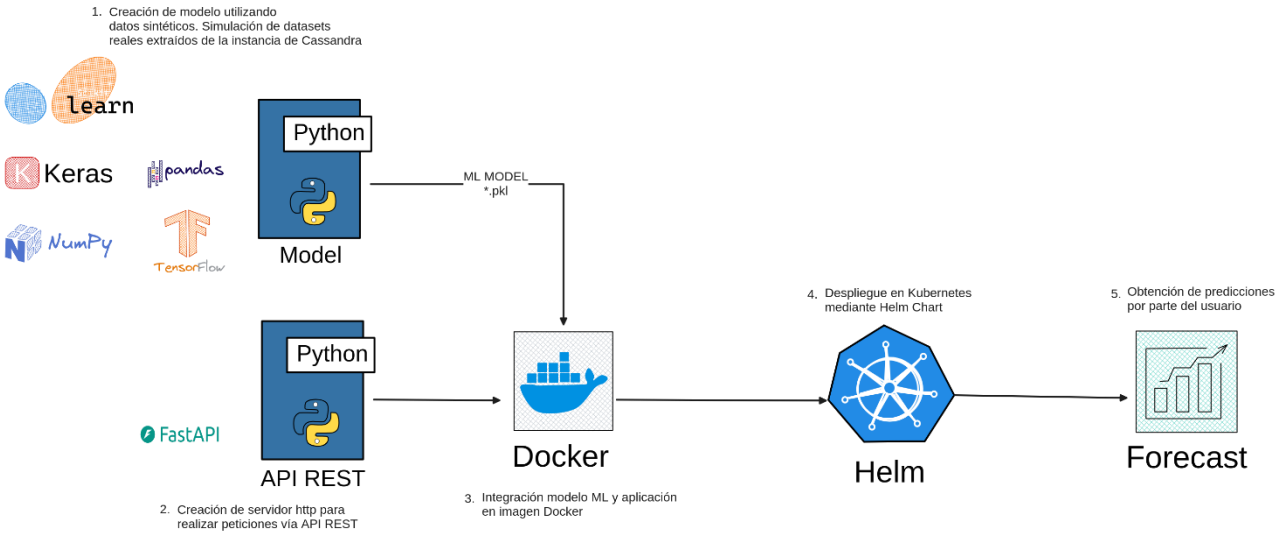

Ilustración 12 Integración de modelo de *Machine Learning* con *Docker* y *Helm Chart*. Fuente: elaboración propia

4.3.5. Creación de modelo de Machine Learning

Para este proceso se ha planteado la generación de datos sintéticos, para ser utilizados en el notebook de *Jupyter*. Se han generado *datasets* de valores de telemetría generados por los *devices* desplegados, usando para ello la función *random.normal* de la librería *numpy,* tal y cómo es representado en el siguiente *snippet* de código *Python:*

```
# Generating synthetic NIGHT SKY BRIGHTNESS (mag/arcsec2) dataset
```

[17] (Python Software Foundation, 2024)

```
data["Sky_Brightness_R_mag_arcsec2"] = np.random.normal(NSB_R, 0.18, size=DATASET_POINTS)

data["Sky_Brightness_V_mag_arcsec2"] = np.random.normal(NSB_V, 0.26, size=DATASET_POINTS)

data["Sky_Brightness_B_mag_arcsec2"] = np.random.normal(NSB_B, 0.20, size=DATASET_POINTS)

data["Sky_Brightness_U_mag_arcsec2"] = np.random.normal(NSB_U, 0.22, size=DATASET_POINTS)

# Generating synthetic wind speed (m/s)

data["Wind_Speed_m_s"] = np.random.normal(WIND_SPEED, 0.5, size=DATASET_POINTS)


# Generating synthetic relative humidity (%)

data["Humidity_percentage"] = np.random.normal(HUMIDITY, 0.05, size=DATASET_POINTS)


# Generating synthetic temperature (ºC)

data["Temperature_Celsius"] = np.random.normal(TEMPERATURE, 5.0, size=DATASET_POINTS)


# Generating synthetic pressure (hPa)

data["Pressure_hPa"] = np.random.normal(PRESSURE, 10.0, size=DATASET_POINTS)


# Generating synthetic rainning condition ( TRUE / FALSE )

data["rainning"] = np.random.rand(DATASET_POINTS) > 0.75


# Generating synthetic acceptance ( TRUE / FALSE )

data["acceptance"] = [ i > NSB_V for i in data["Sky Brightness [V] (mag/arcsec2)"]]

df = pd.DataFrame(data)

df.to_csv("mydataframe.csv") # dump to csv file

df.head()
```

Los datos generados de manera sintética se han representado en la siguiente gráfica:

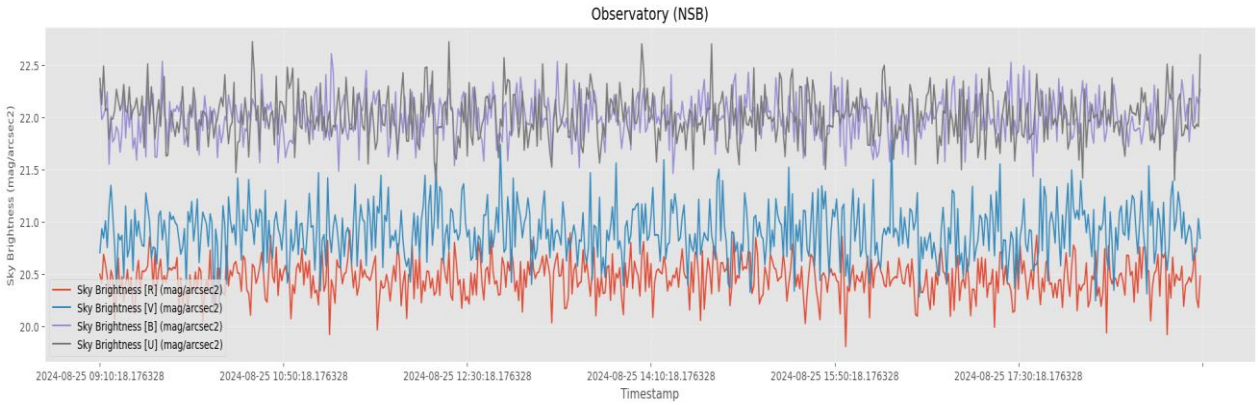

Ilustración 13 Reprentación obtenida con un notebook de *Jupyter* de los datos sintéticos generados. Fuente: elaboración propia

A continuación, se generará un *pipeline*, la cual realizará una transformación de columnas mediante un escalado de tipo *MinMaxScaler* de la librería *Scikit-learn:*

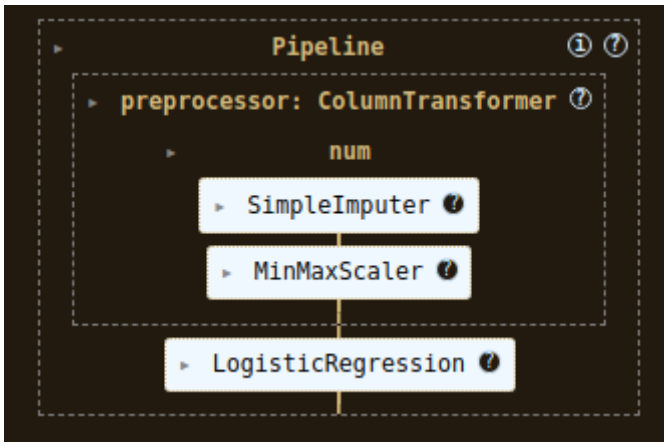

Ilustración 14 Representación de esquema de pipeline de transformación de datos obtenido con notebook de *Jupyter*. Fuente: elaboración propia

Una vez planteada el *pipeline*, se generará un modelo exportado mediante la utilización de la librería *pickle* en *Python*:

```
import pickle

with open("mymodel.pkl", "wb") as f:

    pickle.dump(pipeline, f, protocol=pickle.HIGHEST_PROTOCOL)
```

Para el ajuste del modelo de *Machine Learning* se ha utilizado un ajuste de tipo *LogisticRegression* (Moez, 2023), el cual es usado para problemas de clasificación. En este caso,

se busca determinar si las condiciones de calidad de los espejos son las adecuadas, en función de los parámetros recogidos con la telemetría generada con los *devices* desplegados con *Tango-Controls.*

4.3.6. Visualización de datos

Desde un punto de vista de prueba de concepto, se ha determinado que la visualización de la telemetría generada por los distintos componentes de la capa *fog* demuestra la factibilidad de la solución planteada. Dicha capacidad de visualización demuestra que el *hot path* de la arquitectura lambda ha sido planteada correctamente y que, a su vez, se puede persistir la telemetría en una base de datos tipo *Cassandra*, la cual es ampliamente utilizada en verticales *IoT* (Khurana, 2022).

En esta primera iteración, donde se ha propuesto una prueba de concepto con un alcance limitado, se ha usado *Grafana* como plataforma de visualización. Sin embargo, en un alcance real, se plantearía una solución web customizada para la plataforma, donde además de la visualización de telemetría, se disponga de un conjunto mayor de funcionalidades [18].

En nuestra prueba de concepto, se ha dispuesto la creación de un pequeño *Dashboard* de *Grafana*. Además, para verificar tráfico, y tener visibilidad de nuestro clúster, hemos utilizado profusamente la herramienta *k9s*, de donde se ha obtenido un conjunto de captura de pantallas donde se muestran los logs de algunos *pods* representativos.

[18] Esto es explicado con algo más de detalle en este aquí

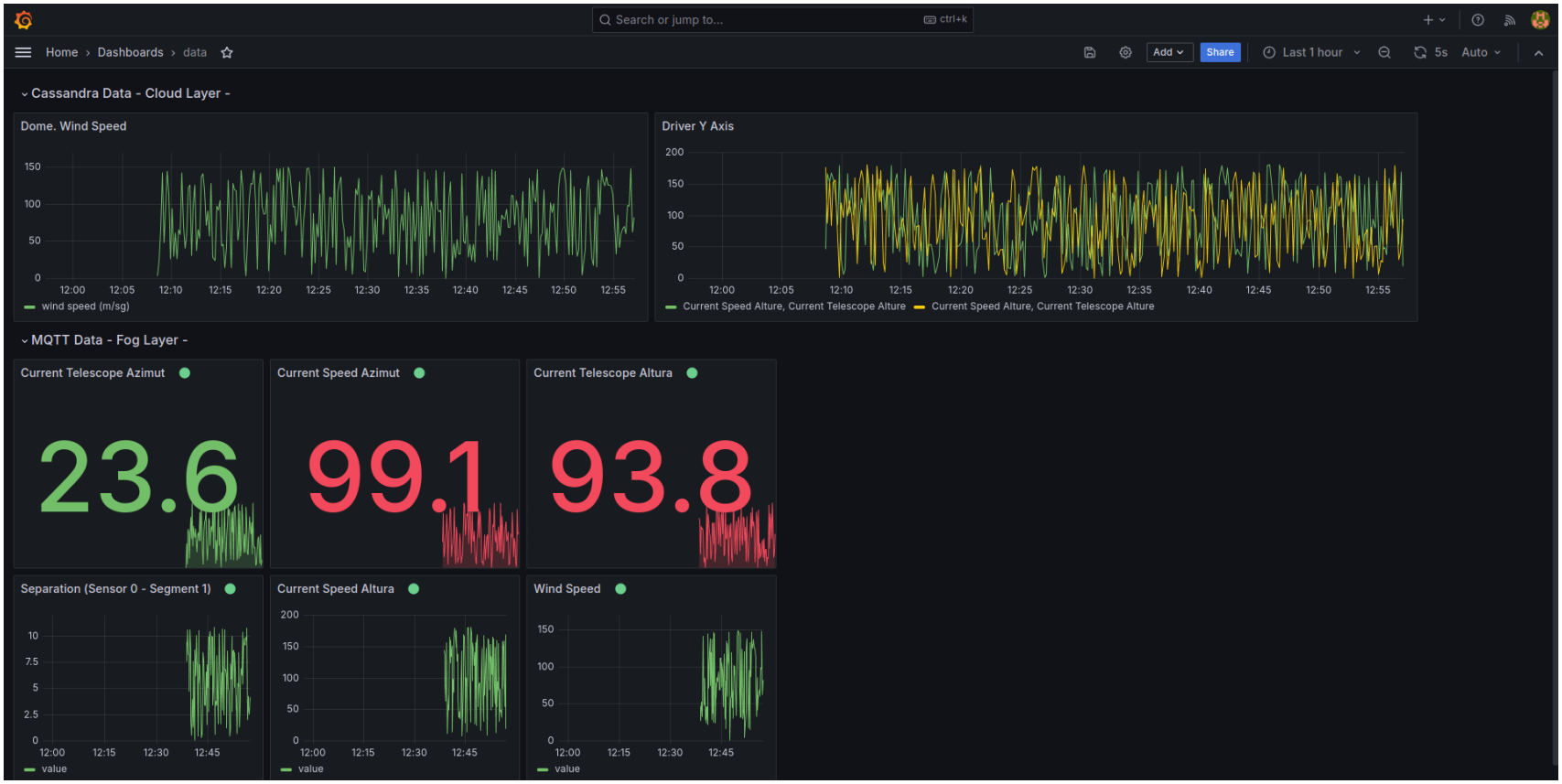

Ilustración 15 Captura de pantalla de Grafana[19]. Fuente: elaboración propia

[19] En esta *dashboard,* están siendo representados en tiempo real las métricas correspodientes a *Cassandra* y a *MQTT*.

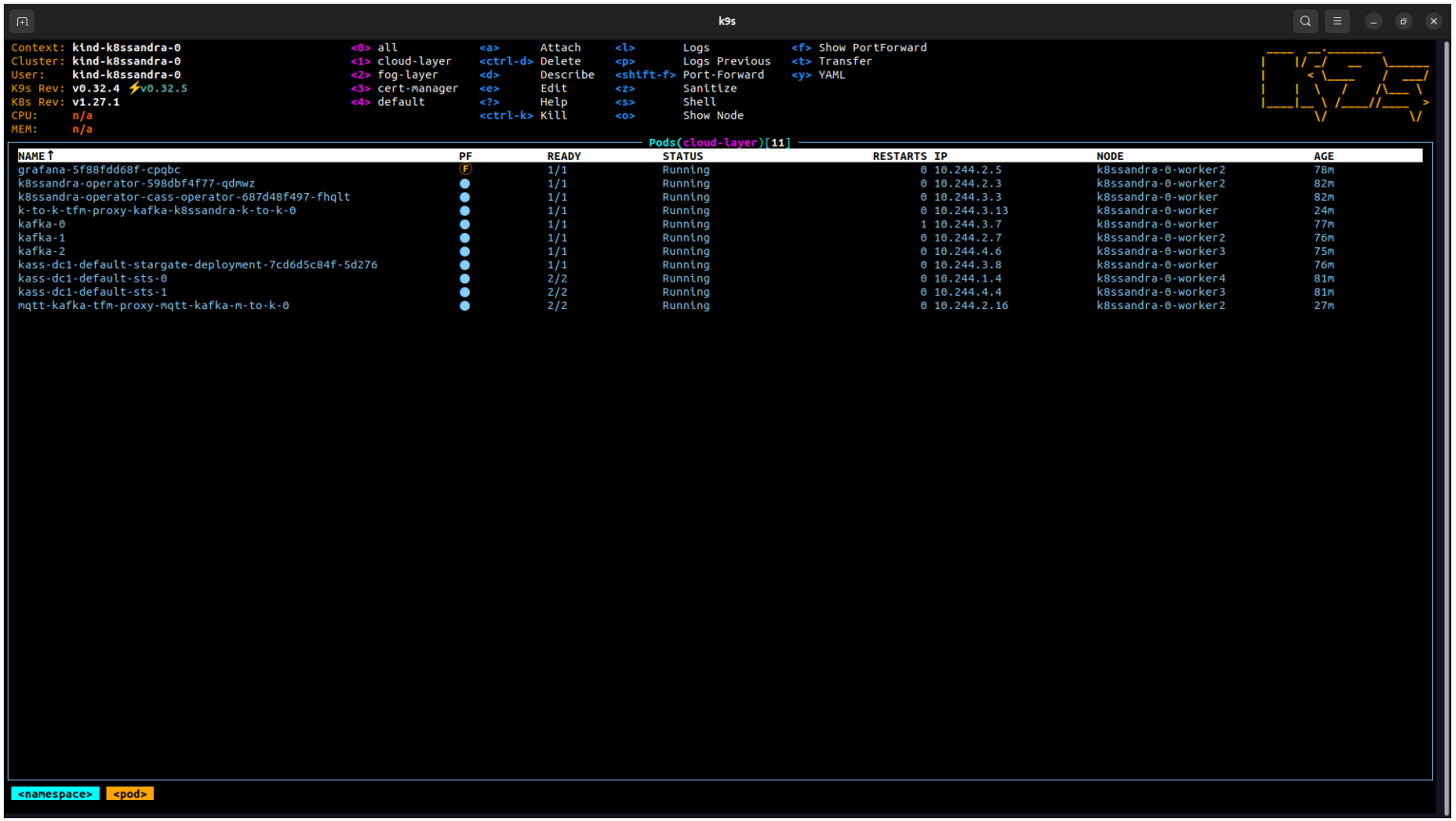

Ilustración 16 Captura de pantalla de *k9s*. Distribución completa de *pods* en ejecución en el *namespace cloud-layer*. Fuente: elaboración propia

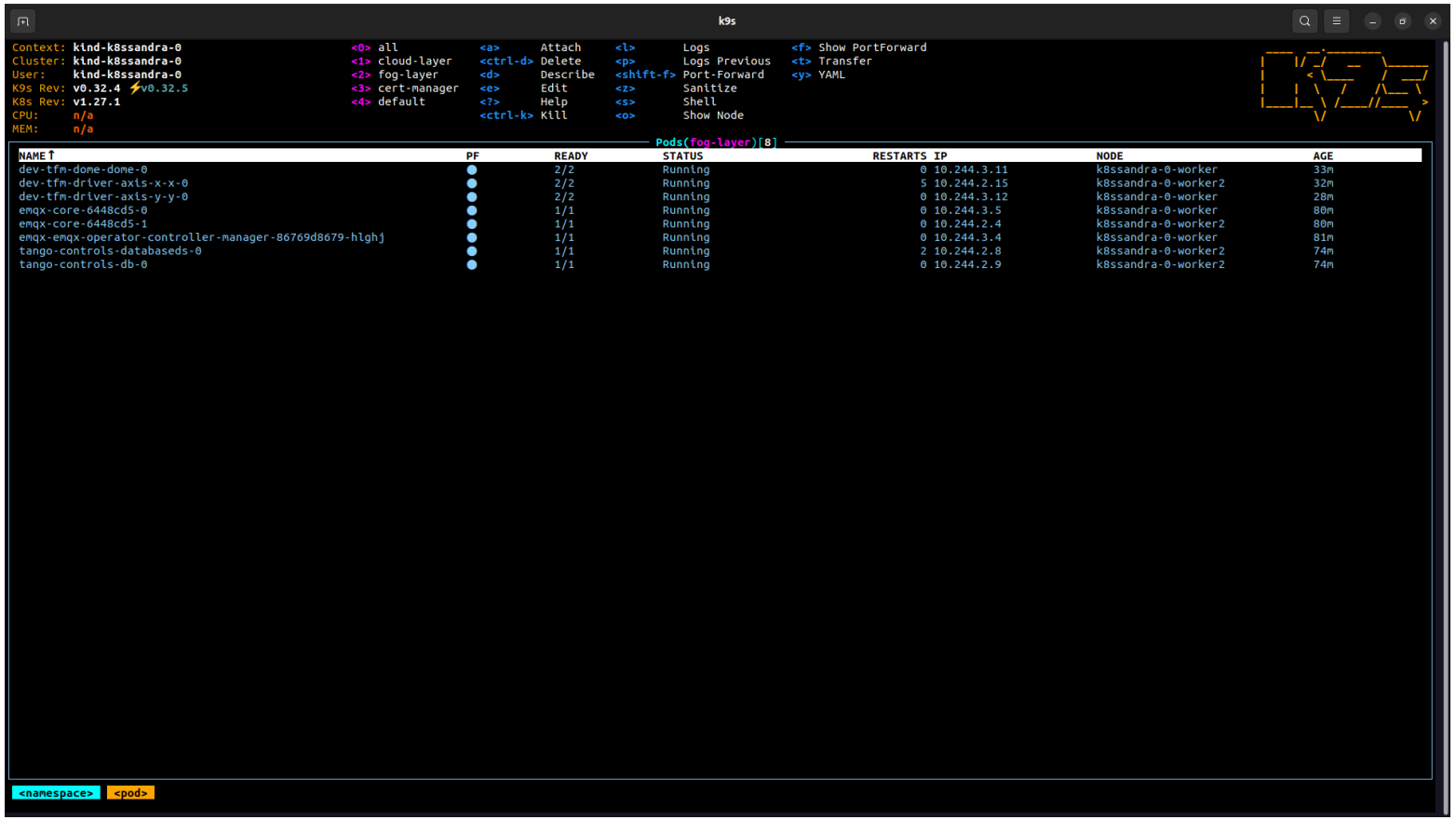

Ilustración 17 Captura de pantalla de *k9s*. Distribución completa de *pods* en ejecución en el *namespace fog-layer*. Fuente: elaboración propia

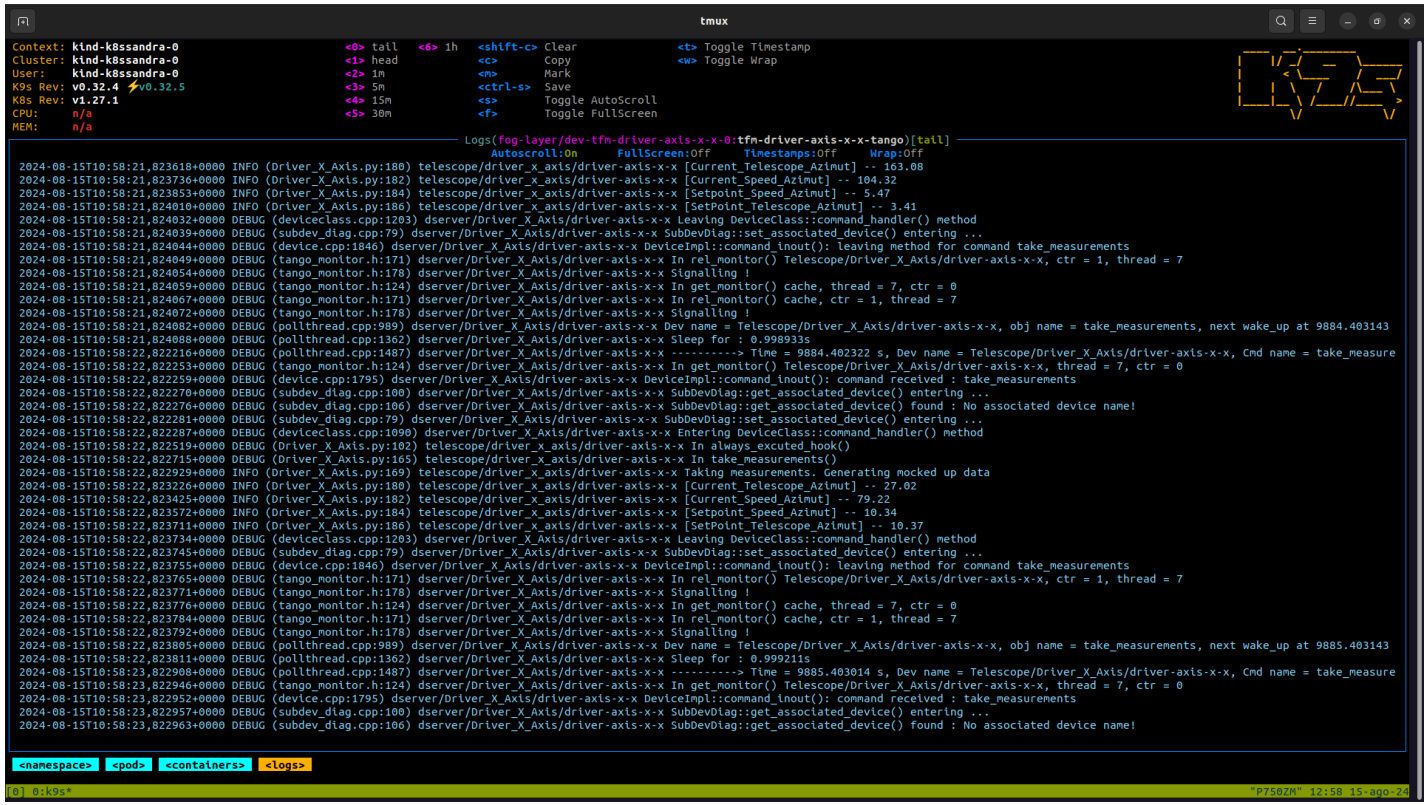

Ilustración 18 Captura de pantalla de k9s. Log del *device driver axis x*[20]. Fuente: elaboración propia

[20] Véase la ejecución periódica de lecturas simuladas de la posición del *driver*, todo ello integrado con un método *CORBA* en el dispositivo de *Tango Controls*.

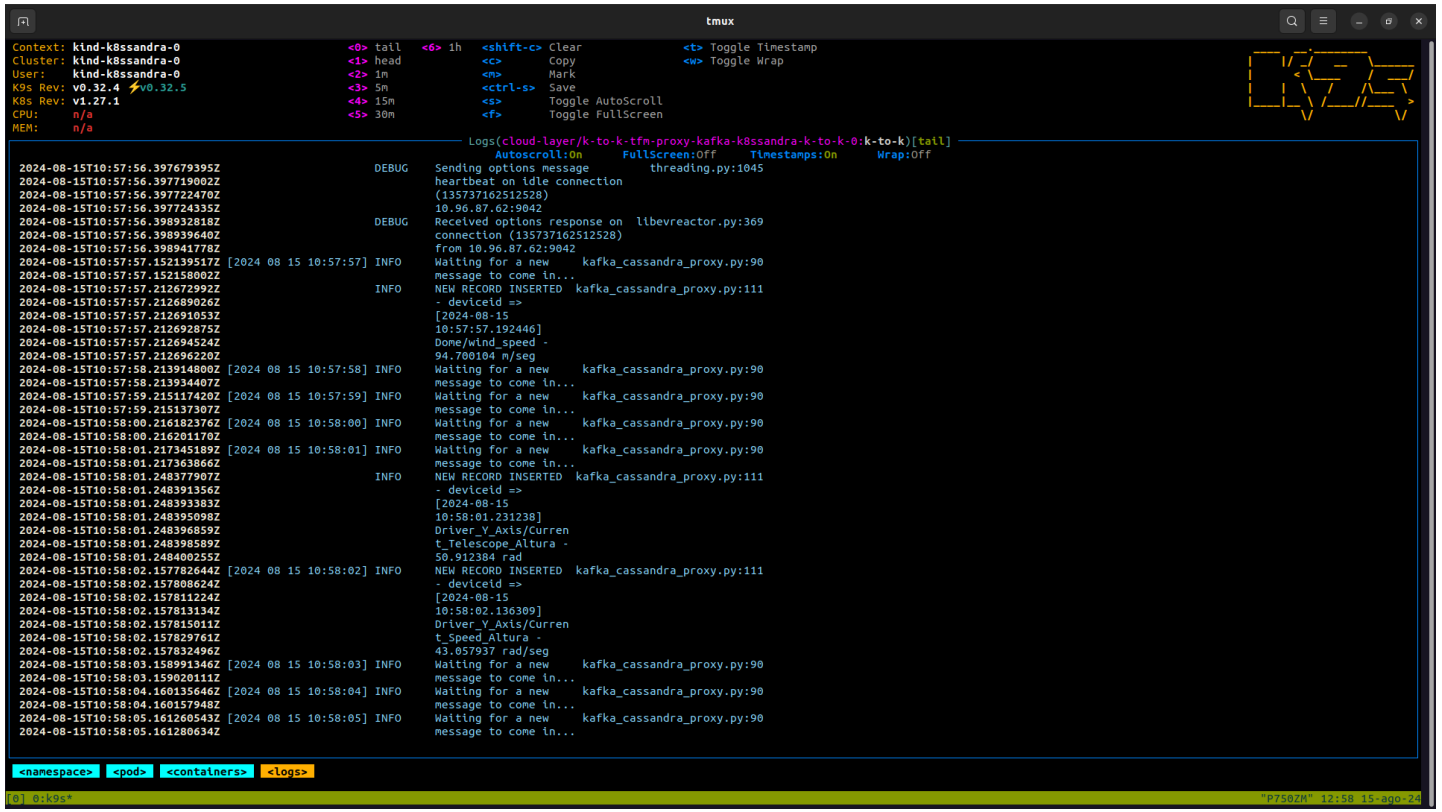

Ilustración 19 Captura de pantall de k9s. Log del proxy Kafka to Cassandra[21]. Fuente: elaboración propia

[21] Nótese la publicación periódica de datos desde *Kafka* hacia *Cassandra* por parte del cliente *Python*.

5. CONCLUSIONES Y TRABAJOS FUTUROS

5.1. CONCLUSIONES

Remitiéndonos al apartado 3.1, el objetivo general de este Trabajo de Fin de Máster es el citado a continuación:

" (...) Definición de una vertical *IoT* que permita la operación de un telescopio, así como la recogida, análisis y visualización de la telemetría generada durante la operación de las instalaciones. (...)"

A su vez, se ha llegado a concretar una serie de objetivos más específicos, los cuales son mostrados en el siguiente párrafo:

"(...)

Disminuir el tiempo empleado en las fases de definición y desarrollo del sistema de control de un telescopio

Estandarizar las tecnologías utilizadas en el desarrollo y despliegue del sistema de control de un telescopio

Aumentar la escalabilidad e integración de cualquier telescopio en un observatorio, al disponer de una plataforma cloud capacitada para añadir nueva funcionalidad.

Mejorar el proceso de mantenimiento y fabricación de espejos para telescopios.

(...)"

Cómo se ha comentado en el capítulo 2.1, uno de los mayores inconvenientes que presenta una instalación de investigación astronómica suele ser su remota ubicación. Por eso existe una predisposición a la automatización y robotización de estas instalaciones. Además, aunque desde observaciones científicas se recogen datos para su posterior estudio, el resto de telemetría generada no se suele almacenar y estudiar a largo plazo.

Esto implica que exista una oportunidad que no está siendo aprovechada para mejorar otros aspectos de la operación de un telescopio. En este sentido, lo que sería la óptica de un telescopio podría beneficiarse de la aplicación de técnicas de inteligencia artificial con los datos recogidos por una plataforma *IoT*.

Para el NRT, como ejemplo más destacado, la óptica se manufacturará en el CSOA. Sería una situación bastante interesante el poder almacenar y estudiar la telemetría generada durante

todo el proceso de fabricación de los espejos. Y luego continuar con la adquisición de datos sobre la vida útil de los mismos en el observatorio.

Analizando este tipo de situaciones, se ve que l**a ejecución del presente proyecto permitiría realizar la adquisición, almacenamiento, visualización y estudio de los datos generados durante la vida útil de dichos espejos**. Es más, a**l tener una arquitectura diferenciada en dos partes, capa *fog* y capa *cloud,* se podría crea una federación a nivel de observatorio donde cada telescopio** (que tendría su propio *datacenter* independiente de baja latencia, ejecutando *Tango Controls* como *framework* para el software de control) **volcaría datos hacia un *cloud* común vía MQTT - *Kafka.***

Con esta estrategia en mente, se cumplirían los objetivos de:

- **Definición de una vertical IoT que permita la operación de un telescopio**.
- **Disminución del tiempo de desarrollo de un telescopio**, al reutilizar diferentes elementos del sistema. Esto se consigue por tanto debido a la estandarización del software de control de un telescopio
- Al trabajar con una estructura fog-cloud, se permite la **integración de nuevas plataformas de observación**, al definirlas como una nueva capa en la niebla, que se conecta con una nube común.
- Al recopilar las métricas en **la base de datos Cassandra desplegada en la nube**, tanto el proceso de manufactura, como el de la vida útil, se puede determinar cómo afectan las inclemencias del tiempo a la calidad de imagen obtenida por un espejo. Con ello se podría buscar **por tanto posibles soluciones (basadas en técnicas de inteligencia artificial)** o paliativos a los problemas generados por la exposición al medio ambiente.

5.2. LÍNEAS DE TRABAJO FUTURO

Por la envergadura de estos proyectos, queda mucho por hacer en lo relativo a la plataforma de control:

- Una vez definidos los contenedores Docker y la generación de Helm Charts, el siguiente paso sería trabajar con el equipo de SRE para **automatizar estos procesos con técnicas**

DevOps. Esto implica el despliegue de las herramientas necesarias para ello, como son los registros de artefactos, la elaboración de *workflows*

- **Creación y despliegue de las capas *fog* y *cloud* en entornos de producción.** Definir y desarrollar ambos despliegues de manera adecuada. Es decir, para la capa *fog* plantear la creación de un clúster con *k0s* en lugar de *Kind.* Integrar *ArgoCD* en el mismo, y automatizar todo el proceso con herramientas como *Ansible.*

  Para la capa *cloud* se estudiará como desplegar la solución en un proveedor de la nube. Se incluirán todos los recursos necesarios para gestión de balanceo de cargas, alta disponibilidad y acceso mediante roles definidos. A su vez, se desplegará Kafka utilizando plataformas ya existentes en la industria, buscando robustez y facilidad de mantenimiento.

  Finalmente, habrá que securizar dichas plataformas, de tal manera que se cumplan con los estándares de ciberseguridad establecidos.
- **Integración total de Tango-Controls.** Todo el sistema de control estará basado en este *framework*. Esto implica que el *hardware* a utilizar esté definido mediante sus respectivos *devices* y que, a su vez, tengamos accesibilidad a los mismos desde un secuenciador u otra aplicación. Todo ello significa que, probablemente, haya que desarrollar algún tipo de plugin que facilite este requerimiento, integrando una API REST para cada *device* desarrollado
- **Desarrollo de aplicación web**. Por limitaciones de alcance, el presente TFM ha utilizado *Grafana* para visualización de telemetría. En la solución final, se deberá contar con una plataforma web con dicha funcionalidad integrada. Además, no solo se tendrá que disponer en la misma de gestión de usuarios, sino también la posibilidad de gestionar alarmas, eventos, datos científicos, gestión de instrumentación etc. Esta plataforma web, no se limitará únicamente a un telescopio, sino que en realidad se planteará como algo extensible y escalable a nivel de observatorio
- **Integración con secuenciador**. Adaptar una solución como *Apache Airflow* a la operación de un telescopio es algo fundamental para la automatización de todo el proceso. Habrá que estudiar la integración de *Apache Airflow* con *Tango Controls*, así

como desarrollar y depurar cada una de las recetas a utilizar a la hora de realizar distintas tareas.

- **Model tuning.** Con todos los datos persistidos en *Cassandra,* se buscará optimizar la utilización de modelos predictivos para el comportamiento a largo plazo de los espejos. Como ya se citó con anterioridad en este mismo capítulo, se podría plantear la integración del proceso de manufacturación en la plataforma *cloud* que ya existiría, ligando de esta manera toda la vida de un espejo, desde su creación hasta su operación.

Anexo A. Gestión del proyecto

A la hora de gestionar el proyecto, se ha propuesto la utilización de GitHub como plataforma de gestión. Así se pretende dividir el trabajo para cada equipo y definir en el tiempo los diferentes paquetes de trabajo.

Cómo ha sido comentado en el apartado Planificación general, el grupo de trabajo se dividirá en 3 equipos diferentes:

- Equipo Nube
- Equipo Niebla
- Equipo SRE.

El siguiente conjunto de ilustraciones mostradas en este apéndice, muestra la integración con tableros *Kanban*, diagramas de *Gantt* y *backlogs* realizada para el presente proyecto. Esto es bastante útil al disponer de los recursos en el tiempo, y a la vez realizar un tracking sobre el estado real del trabajo realizado hasta el momento y programado para un futuro.

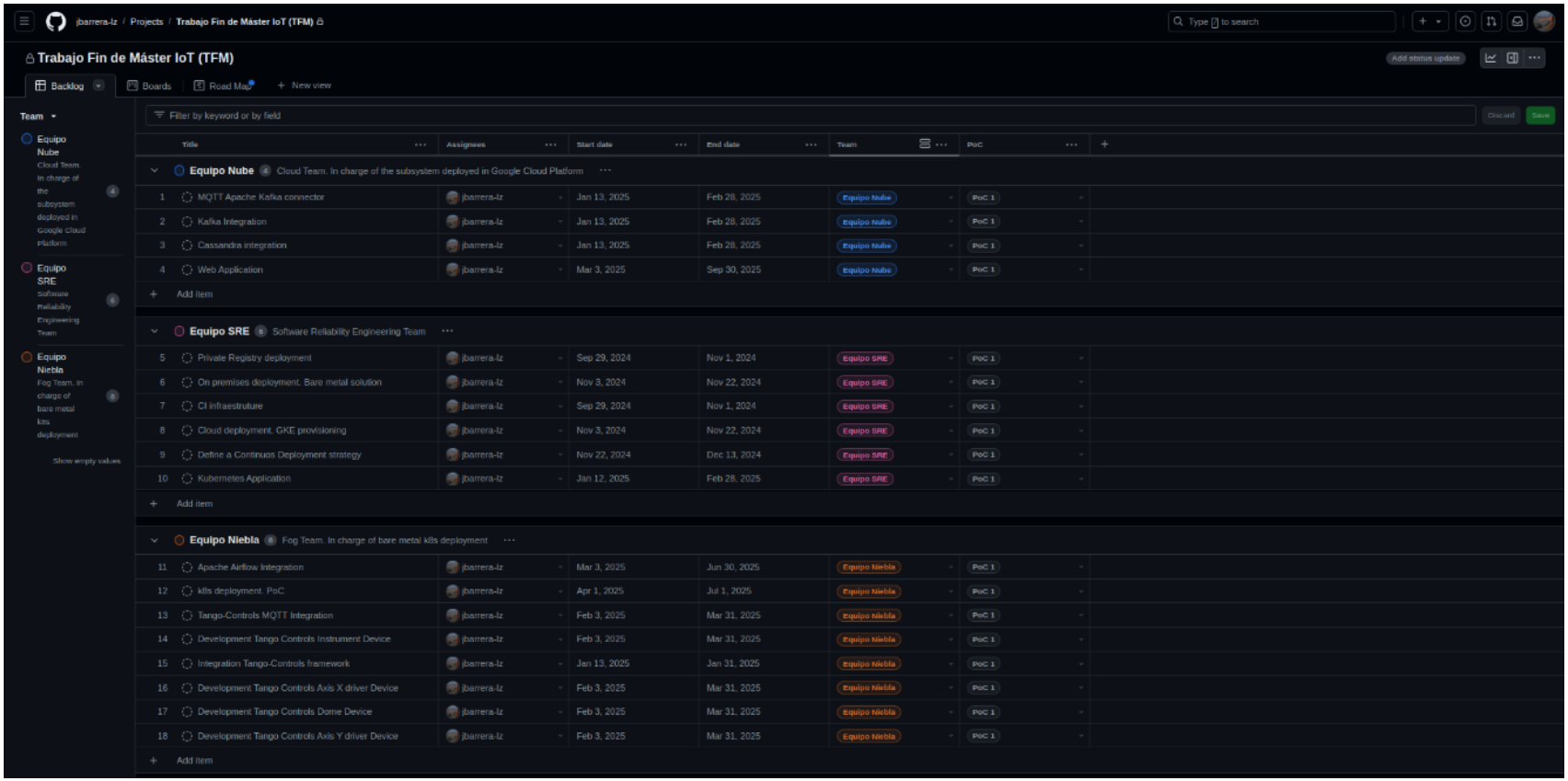

Ilustración 20 *Github project* backlog. Fuente elaboración propia

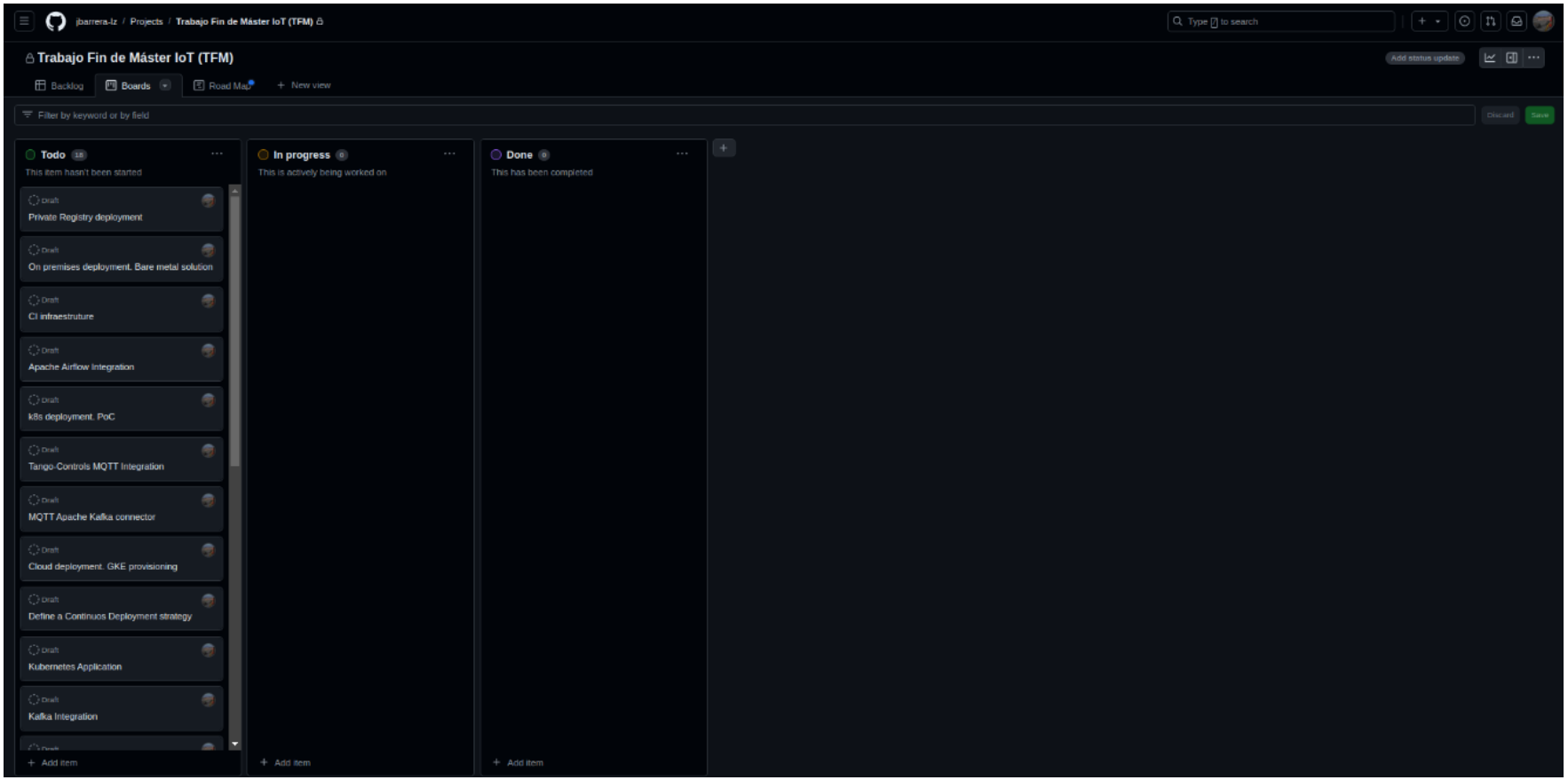

Ilustración 21 *Github Project Board*. Fuente Elaboración propia

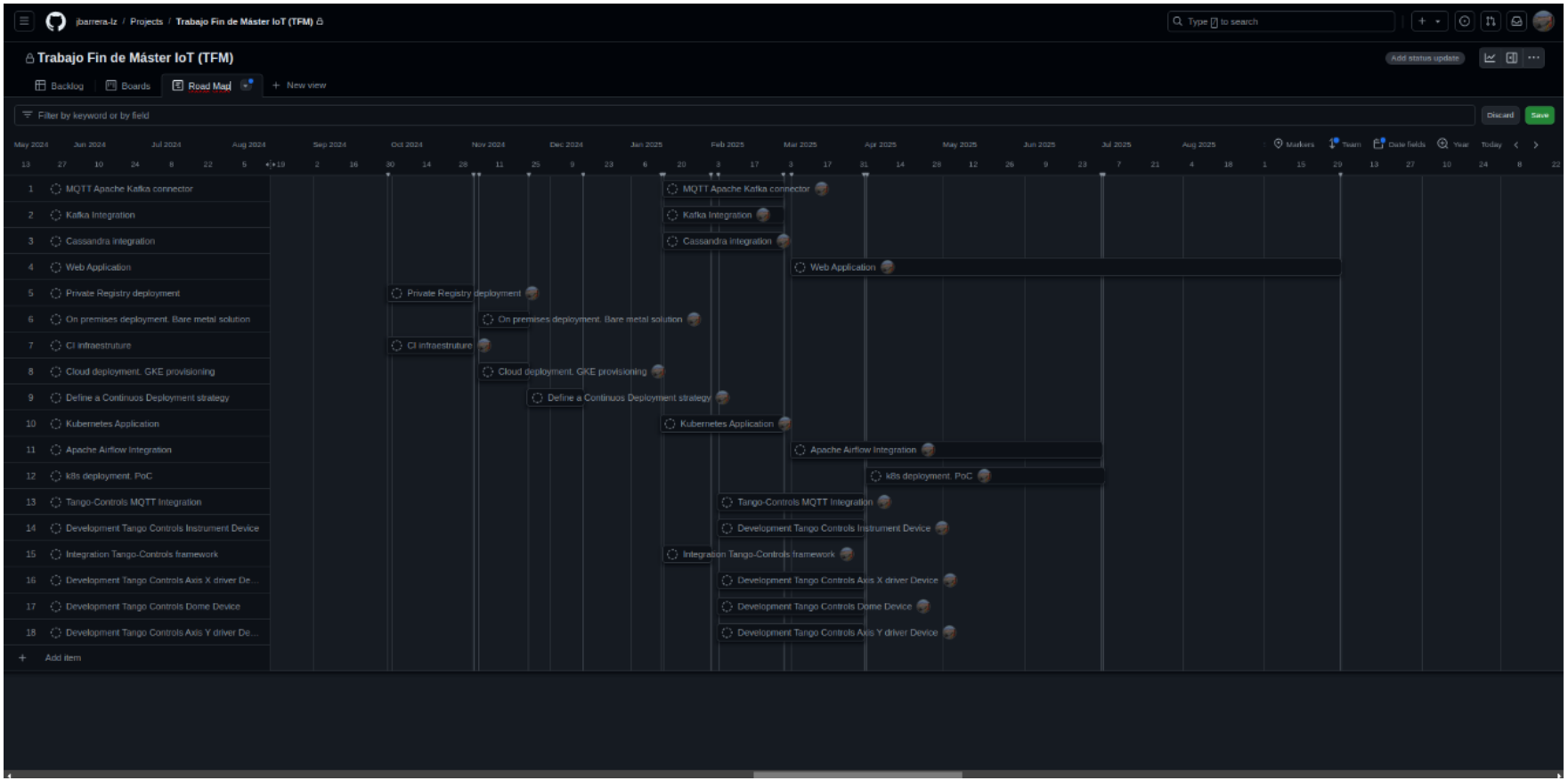

Ilustración 22 *Github Gantt Diagram*. Fuente: elaboración propia

Anexo B. Código fuente. Listado de repositorios

Tabla 9 Colección de repositorios desarrollados por el autor del trabajo fin de máster

Repositorio tfm-deployment (Barrera, 2024a)	
Descripción	Conjunto de *scripts* y *Helm charts* utilizadas para el despliegue del clúster de *Kubernetes*, *Cassandra* y *Kafka*
Versión	v0.0.1
Repositorio tfm-device-template (Barrera, 2024b)	
Descripción	Creación de dependencia que será luego utilizada en los diferentes Tango-controls *devices. Helm chart* definida como *library*, para de esta manera ser reutilizada posteriormente por otra *Helm chart*
Versión	v0.0.6
Repositorio tfm-proxy-mqtt-kafka (Barrera, 2024h)	
Descripción	*Proxy translator* entre los servicios MQTT y Kafka.
Versión	v0.0.3
Repositorio tfm-simulated-instrument (Barrera, 2024i)	
Descripción	Instrumento simulado Tango-Controls *device*
Versión	v0.0.3
Repositorio tfm-edge-sensor (Barrera, 2024f)	
Descripción	Sensor de borde Tango-controls *device*
Versión	v0.0.2

Repositorio tfm-driver-axis-y (Barrera, 2024e)	
Descripción	Driver eje-Y Tango-Controls *device*
Versión	v0.0.6
Repositorio tfm-dome (Barrera, 2024c)	
Descripción	Cúpula Tango-Controls *device*
Versión	v0.0.6
Repositorio tfm-driver-axis-x (Barrera, 2024d)	
Descripción	Driver eje-X Tango-Controls *device*
Versión	v0.0.6
Repositorio tfm-tango-controls (Barrera, 2024j)	
Descripción	Despliegue de componentes de Tango-Controls framework: *MySQL database - db -* y *Device Server Database - dbds service -*
Versión	v0.0.2
Repositorio tfm-kafka-cassandra-proxy (Barrera, 2024g)	
Descripción	Proxy translator entre los servicios Kafka y Cassandra
Versión	v0.0.13
Repositorio tfm-telescope-ml (Barrera, 2024k)	

Descripción	Planteamiento de integración de modelo *ML* a partir de la telemetría generada por los *devices Tango-Control*, desplegada a través de *FastAPI*
Versión	v0.0.1

Anexo C. Kubernetes instance calculator

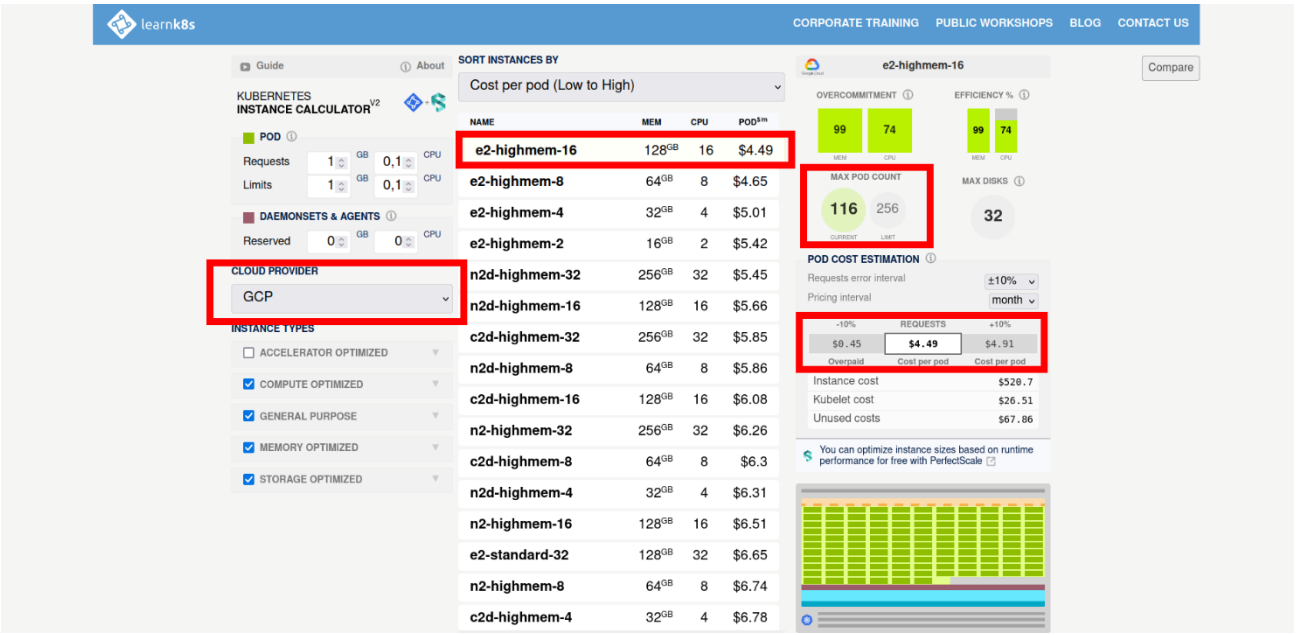

Ilustración 23 Captura de pantalla de calculadora de instancias de Kubernetes[22].

[22] Nótese que el proveedor seleccionado es Google Cloud Platform, el tipo de vCPU es e2-highmem-16, con un precio unitario por pod desplegado de 4.49USD

References

Amazon Web Services Inc. (2023, *¿Qué es Docker?* https://aws.amazon.com/es/docker/

Anderson, D. (2023, Oct 7,). *ArgoCD App-of-Apps -- A GitOps Approach.* https://medium.com/@andersondario/argocd-app-of-apps-a-gitops-approach-52b17a919a66

anonymous user from IP , & 71.41.210.146. (2024, *Comparison of nominal sizes of primary mirrors of notable optical reflecting telescopes, and a few other objects.* https://commons.wikimedia.org. https://commons.wikimedia.org/w/index.php?curid=33613161

Apache Kafka. (2024, May 19,). *Apache Kafka.* https://kafka.apache.org/. https://kafka.apache.org/

Argo Project. (2024). argo-cd [computer software]. https://argo-cd.readthedocs.io/en/stable/:

Aristofanes: Nubes (2005). . Editorial Universitaria.

Barrera, J. (2024a). tfm-deployment [computer software]. https://github.com/jbarrera-lz/tfm-driver-axis-x.git:

Barrera, J. (2024b). tfm-device-template [computer software]. https://github.com/jbarrera-lz/tfm-driver-axis-x.git:

Barrera, J. (2024c). tfm-dome [computer software]. https://github.com/jbarrera-lz/tfm-dome.git:

Barrera, J. (2024d). tfm-driver-axis-x [computer software]. https://github.com/jbarrera-lz/tfm-driver-axis-x.git:

Barrera, J. (2024e). tfm-driver-axis-y [computer software]. https://github.com/jbarrera-lz/tfm-driver-axis-x.git:

Barrera, J. (2024f). tfm-edge-sensor [computer software]. https://github.com/jbarrera-lz/tfm-edge-sensor.git:

Barrera, J. (2024g). tfm-kafka-cassandra-proxy [computer software]. https://github.com/jbarrera-lz/tfm-kafka-cassandra-proxy.git:

Barrera, J. (2024h). tfm-proxy-mqtt-kafka [computer software]. https://github.com/jbarrera-lz/tfm-driver-axis-x.git:

Barrera, J. (2024i). tfm-simulated-instrument [computer software]. https://github.com/jbarrera-lz/tfm-simulated-instrument.git:

Barrera, J. (2024j). tfm-tango-controls [computer software]. https://github.com/jbarrera-lz/tfm-driver-axis-x.git:

Barrera, J. (2024k). tfm-telescope-ml [computer software]. https://github.com/jbarrera-lz/tfm-telescope-ml.git:

Bekker, A. (2018, August 14,). *When to use Cassandra and when to steer clear.* https://towardsdatascience.com/when-to-use-cassandra-and-when-to-steer-clear-72b7f2cede76. https://towardsdatascience.com/when-to-use-cassandra-and-when-to-steer-clear-72b7f2cede76

Bento, J., Arnold, D. M., Smith, R. J., Fern 'andez-Valdivia, J. J., Gil, J. L. '., Martin, J. '. B., & Gill, M. A. T. (2022a). Experience of utilising CI/CD practices in the development of software for a modern astronomical observatory. Paper presented at the *Software and Cyberinfrastructure for Astronomy VII, , 12189* 1218905. 10.1117/12.2629975 https://doi.org/10.1117/12.2629975

Bento, J., Smith, R. J., Arnold, D. M., Fernandez-Valdivia, J. J., Gil, J. L. '., Mart 'in, J. '. B., & Torres, M. (2022b). Software architecture and development plan for a 4m fully

autonomous observatory (New Robotic Telescope). Paper presented at the *Software and Cyberinfrastructure for Astronomy VII, , 12189* 121890. 10.1117/12.2629053 https://doi.org/10.1117/12.2629053

Bradford, C., & Garnsey, M. (2024, *K8ssandra - k8ssandra, Apache Cassandra on Kubernetes.* https://k8ssandra.io/. https://k8ssandra.io/

Breitfelder, S. (2024, July 3,). *kubeselect.* https://github.com/sbreitf1. https://github.com/sbreitf1/kubeselect.git

Confluent Inc. (2024, *Pay as you go pricing.* https://www.confluent.io/confluent-cloud/pricing/. https://www.confluent.io/confluent-cloud/pricing/

Docker Inc. (2024, *Building best practices.* https://docs.docker.com/. https://docs.docker.com/build/building/best-practices/

EMQX Team. (2024, March 26,). *MQTT to Kafka: Benefits, Use Case & A Quick Guide.* https://www.emqx.com/en/blog/mqtt-and-kafka.

European Solar Telescope Home. (2024, May 18,). https://www.est-east.eu.

Filgueira, J. M., & Rodriguez, D. (1998). GTC control system: an overview. Paper presented at the *Telescope Control Systems III, , 3351* 2. 10.1117/12.308815 https://doi.org/10.1117/12.308815

GitHub, I. (2024, Jun 22,). *About Github.* https://docs.github.com. https://docs.github.com/en/get-started/start-your-journey/about-github-and-git

Google. (2022, July 17,). *¿Qué es kubernetes?* https://kubernetes.io/. https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/

Google. (2024, Aug 09,). *Configure maximum Pods per node.* https://cloud.google.com/kubernetes-engine/docs/how-to/flexible-pod-cidr. https://cloud.google.com/kubernetes-engine/docs/how-to/flexible-pod-cidr

Guerra Ramos, D. (2020). *Cofaseado de espejos segmentados con aprendizaje automático* http://riull.ull.es/xmlui/handle/915/24288

Helm. (2024, *Helm. El administrador de paquetes para kubernetes.* https://helm.sh/es/.

Instituto Astrofísico de Canarias. (2024, May 18,). *Gran Telescopio de CANARIAS.* https://www.gtc.iac.es. https://www.gtc.iac.es

Instituto Astrofísico de Canarias, (. (2024, May 18,). *New Robotic Telescope.* https://www.iac.es/es/observatorios-de-canarias/telescopios-y-experimentos/new-robotic-telescope.

Instituto de Astrofísica de Canarias, - IAC. (2024, *CSOA.* https://www.iac.es. https://www.iac.es/es/proyectos/centro-de-sistemas-opticos-avanzados-csoa

Jermak, H. E., Barrera, J. '., Copley, D., Copperwheat, C. M., Juez, J. D. C., Rodriquez, J. G., Fernandez-Valdivia, J. J., Pi nero, A. G. '. a., Guti 'errez, C. M., Harvey, '., Insausti, M., Knapen, J. H., Guti 'errez, A. M., McGrath, A. M., Oria, A., Ranjbar, A., Rebolo-L 'opez, R., Steele, I. A., Torres, M., . . . Aukkaravittayapun, S. (2020). The New Robotic Telescope: progress report. Paper presented at the *Ground-Based and Airborne Telescopes VIII, , 11445* 114453D. 10.1117/12.2562706 https://doi.org/10.1117/12.2562706

k0s. (2024, *k0s. Kubernetes distribution for bare-metal, on-prem, edge, IoT.* https://k0sproject.io/. https://k0sproject.io/

Khurana, S. (2022, Jun 14,). *System Design Solutions: When to use Cassandra and when not to.* https://medium.com/geekculture/system-design-solutions-when-to-use-cassandra-and-when-not-to-496ba51ef07a

Kind. (2024, Feb 27,). *Kind - Kubernetes.* https://kind.sigs.k8s.io/. https://kind.sigs.k8s.io/

Liverpool Jonh Moores University. (2024, *Liverpool Telescope.* https://telescope.livjm.ac.uk/. https://telescope.livjm.ac.uk/

Machado, D. (2023, Mar 13,). *ML - Deploy Machine Learning Models Using FastAPI.* https://medium.com/. https://dorian599.medium.com/ml-deploy-machine-learning-models-using-fastapi-6ab6aef7e777

Microsoft Corporation. (2024, *Development Containers.* https://containers.dev/. https://containers.dev/

Moez, A. (2023, Sep 8,). *Easily deploy machine learning models form the comfort of your Notebook.* https://medium.com. https://moez-62905.medium.com/easily-deploy-machine-learning-models-from-the-comfort-of-your-notebook-9068a88f4cf5

New Robotic Telescope. (2024, May 18,). *New Robotic Telescope.* https://www.robotictelescope.org.

Parashar, N. (2022, Jul 12,). *What is IIOP?* https://medium.com. https://medium.com/@niitwork0921/what-is-iiop-4bcf85bb6f01

Penataro, R., Filgueira, J. M., Gomez-Cambronero, P., Gonzalez, M., & Puig, M. P. i. (2000). Application of CORBA to the GTC control system. Paper presented at the *Advanced Telescope and Instrumentation Control Software, , 4009* 152. 10.1117/12.388385 https://doi.org/10.1117/12.388385

PlanetScale Inc. (2023, March 13,). *The eight phases of DevOps.* https://planetscale.com/docs/devops/intro-to-the-eight-phases-of-devops.

Python Software Foundation. (2024). Pickle [computer software]

Ramírez, S. (2024). FastAPI [computer software]

Sharma, T. (2022, Jul 12,). *CORBA Technology.* https://medium.com. https://medium.com/@2022a1r152/corba-technology-76c38265f126

Singh, A. (2024, April 26,). *Who should use Cassandra?* https://medium.com/@sam700007/who-should-use-cassandra-f4f70f490859.

SKA Observatory. (2024, *SKAO.* https://www.skao.int. https://www.skao.int/en/about-us/skao

SKAO. (2024). SKAO - Gitlab [computer software]. https://gitlab.com/ska-telescope:

Tango Community. (2023, *Pogo (Class Generator).* https://tango-controls.readthedocs.io. https://tango-controls.readthedocs.io/en/latest/tools-and-extensions/built-in/pogo/index.html

Tango Controls. (2024, *Tango Controls: Home.* https://www.tango-controls.org/.

Universidad Internacional de La Rioja, (. (2024). Plataforma de Internet de las Cosas de Google. Ingestión y procesamiento de datos en Google Cloud.

Vinci, A. (2023, Dec 25,). *Lambda Architecture, A Big Data processing framework.* https://medium.com/. https://medium.com/@vinciabhinav7/lambda-architecture-a-big-data-processing-framework-introduction-74a47bc88bd3

Vinci, A. (2024, May 30,). *Cassandra vs Mongo db? How to choose?* https://medium.com/@vinciabhinav7/cassandra-vs-mongo-db-how-to-choose-e2de97a6dc45.

W. M. Keck Observatory. (2024, *Keck Observatory.* https://www.keckobservatory.org/. https://www.keckobservatory.org/