UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

INFRAESTRUCTURA COMO CÓDIGO PARA IOT

por

David Oscanoa Lara

Memoria de Título presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al título profesional de Ingeniero Civil Electrónico(a)

Profesor guía

Mario Medina Carrasco

3 de Septiembre 2025
Concepción, Chile

Agradecimientos

Esta memoria va dedicada en primer lugar a todos mis seres queridos. De ellos, quisiera agradecer especialmente a mi padre Abel quien siempre ha apoyado que aprenda más, a mi madre Elena quien hace mucho ya no está en este mundo pero de quien aún así siempre sentí apoyo, a mi abuela Regula quien ha sido como una madre para mí y finalmente quisiera agradecer a Paula Castillo, quien en muy poco tiempo ha llegado a ser un pilar en mi vida.

Agradezco a Hackmetrix y especialmente a David Riveros, DevOps Manager en Hackmetrix, quienes me han dado las facilidades para realizar esta memoria y me han terminado de enseñar todo lo que no se puede expresar en los libros.

También quisiera agradecer a Travis Jungroth, quien durante la pandemia me enseñó Python y me introdujo a conceptos avanzados del lenguaje durante meses sin pedir algo a cambio.

Quiero agradecer a los profesores de la universidad y especialmente a Mario Medina, quien ha apoyado la mayoría de ideas locas que le he presentado, por su entusiasmo en los tópicos que enseña y por permitirme ampliar mis conocimientos más allá de lo que normalmente entrega la carrera.

Quisiera agradecer a todos quienes no conozco personalmente, pero han hecho posible esta memoria de título. Desde los mantenedores de los proyectos de código abierto de los cuales esta memoria depende, hasta de los artistas cuya música tuve el privilegio de disfrutar mientras realizaba la memoria.

Por último, quisiera agradecerle a los perros de la Universidad de Concepción. Su presencia no sólo ha representado un apoyo a mí, sino a todos los estudiantes. Tanto el nombre de la implementación de referencia «ovejas_project» como el nombre de la herramienta de CLI «ovejas» nacen del «Perro Oveja» quien alguna vez paseó por la Universidad de Concepción.

Como sé que la lista no tendría fin si agradezco a todos quienes me han ayudado, por más triste que sea el caso, si no los nombré y me ven por el centro de Concepción, sepan que les debo un completo italiano (uno a la vez por favor).

Resumen

El crecimiento de Internet de las Cosas (IoT) ha impulsado la proliferación de sistemas embebidos basados en Linux en una amplia gama de aplicaciones, desde la automatización industrial hasta dispositivos de monitoreo remoto. A pesar de este avance, la gestión de estos dispositivos —especialmente en despliegues a gran escala— continúa representando un desafío, debido a restricciones como conectividad limitada y recursos computacionales reducidos.

En este contexto, la metodología de Infraestructura como Código (IaC) se presenta como una posible solución para automatizar la configuración, el despliegue y la gestión de sistemas. No obstante, las herramientas actuales de IaC, como Pulumi, Ansible y Terraform, no están diseñadas específicamente para entornos embebidos ni abordan de manera efectiva sus particularidades.

Este documento propone el diseño e implementación de una herramienta de Infraestructura como Código orientada a sistemas embebidos Linux utilizados en dispositivos IoT. A diferencia de enfoques centrados en sistemas Linux inmutables con actualizaciones A/B —un enfoque costoso—, esta herramienta está pensada para operar sobre sistemas vivos, en donde los cambios se aplican directamente durante la ejecución y deben manejarse de forma segura y controlada. Esta orientación permite integrar prácticas del enfoque DevOps al desarrollo de dispositivos IoT, promoviendo una mayor agilidad y ciclos de entrega más rápidos en el manejo de la infraestructura de los dispositivos.

El diseño toma en cuenta las limitaciones propias de estos entornos y se apoya en tecnologías modernas como Rust, Diesel, Tokio y WebSockets para ofrecer una solución confiable. A lo largo del documento se detallan las decisiones de diseño adoptadas, se contrastan con herramientas del estado del arte y se presenta un ejemplo funcional que sirve como una implementación de referencia.

Finalmente, se realiza un flujo de trabajo en donde se demuestra la viabilidad de la solución dentro de una organización con prácticas de Integración Continua y Despliegue Continuo (CI/CD) para la administración de la infraestructura de dispositivos IoT.

Summary

The growth of the Internet of Things (IoT) has driven the proliferation of Linux-based embedded systems across a wide range of applications, from industrial automation to remote monitoring devices. Despite this progress, managing these devices—especially in large-scale deployments—continues to pose a challenge due to constraints such as limited connectivity and reduced computational resources.

In this context, the Infrastructure as Code (IaC) methodology emerges as a potential solution to automate the configuration, deployment, and management of systems. However, current IaC tools like Pulumi, Ansible, and Terraform are not specifically designed for embedded environments and do not effectively address their particular characteristics.

This document proposes the design and implementation of an Infrastructure as Code tool tailored for Linux-based embedded systems used in IoT devices. Unlike approaches focused on immutable Linux systems with A/B updates—a costly strategy—this tool is designed to operate on live systems, where changes are applied directly during execution and must be handled in a safe and controlled manner. This approach enables the integration of DevOps practices into the development of IoT devices, promoting greater agility and faster delivery cycles in managing device infrastructure.

The design takes into account the inherent limitations of these environments and leverages modern technologies such as Rust, Diesel, Tokio, and WebSockets to provide a reliable solution. Throughout the document, design decisions are explained and contrasted with state-of-the-art tools. A functional example is also presented, which serves as a successful reference implementation.

Finally, we present a workflow to demonstrate the viability of the solution within an organization that follows Continuous Integration and Continuous Deployment (CI/CD) practices to manage IoT infrastructure.

Tabla de contenidos

Tabla de figuras

Tabla de secciones de código

1. Gestión de la infraestructura

1.1. Introducción

Considerando la creciente inclusión de sistemas embebidos conectados a servicios de computación en la nube en tanto actividades industriales como en dispositivos orientados al usuario final, entre los principales desafíos que enfrenta la industria de desarrollo de software en sistemas embebidos está la incorporación de conceptos como integración continua, entrega continua y despliegue continuo (CI/CD) y el desarrollo de herramientas que gestionan el ciclo de vida del desarrollo de software (SDLC). La inclusión de estos procesos promete un aumento en la seguridad, un decrecimiento en el tiempo de lanzamiento al mercado y el correcto funcionamiento en estos dispositivos [1] [2].

Por ejemplo, entre las metodologías actuales de medición de rendimiento en el desarrollo de software se destacan las métricas DORA, cuyos cuatro pilares buscan obtener indicadores de la demora en introducir cambios, la frecuencia en la que se realizan cambios, el porcentaje de fallos y el tiempo de recuperación frente a fallos. [3]

Las herramientas de infraestructura como código (IaC) surgen como una respuesta a los puntos anteriores. Estas herramientas buscan realizar cambios en la infraestructura mediante un lenguaje de dominio específico que, al ser ejecutado, permite facilitar cambios en la infraestructura. Estas herramientas permiten que estas actualizaciones sean escritas de forma declarativa o imperativa, ejecutadas de manera idempotente y que administren el estado de la infraestructura, para evitar problemas de duplicación y la eliminación no deseada de recursos. [4]

Por otra parte, el código escrito en el lenguaje de la herramienta IaC puede ser ejecutado en procesos automatizados y compartido en repositorios de control de versiones, para hacer posible la trazabilidad y transferencia de conocimiento dentro de una organización.

Sin embargo, la literatura sobre herramientas de IaC y su uso en sistemas embebidos es escasa [2]. Entre los enfoques actuales de la industria, se encuentran generar imágenes inmutables del sistema operativo y gestionar los cambios mediante un sistema de actualizaciones Over-The-Air para realizar un reemplazo completo del sistema operativo o bien, realizar los cambios mediante la ejecución de scripts en el dispositivo IoT.

Esta memoria de título explora aspectos fundamentales sobre las consideraciones de seguridad, facilidad de uso, mantenibilidad y robustez de un sistema de infraestructura como código y, además, propone una herramienta de IaC para su uso en sistemas embebidos.

1.2. Infraestructura

La infraestructura de tecnologías de información comprende todas las tecnologías necesarias para operar y gestionar entornos digitales. Entre ellas se pueden considerar tanto el hardware en sí como el software [5].

En un contexto de computación en la nube, la definición de ≪infraestructura de la nube≫ de NIST se compone de dos capas: una capa física y una capa de abstracción [6]. Bajo este contexto de computación en la nube, la capa física corresponde a las máquinas, dispositivos de red y dispositivos

que el proveedor de nube tiene físicamente para disponer a sus clientes. La capa de abstracción, en cambio, permite que los clientes de los proveedores puedan pedir estos recursos bajo demanda mediante una abstracción de ellos, un proceso que se conoce como ≪aprovisionamiento≫ [7].

En el contexto de la administración de dispositivos IoT, se puede usar la misma definición teniendo en cuenta que los administradores de los dispositivos IoT son quienes se encargan tanto de manipular los dispositivos IoT (la capa física) como de incluir herramientas para configurar estos dispositivos (la capa de abstracción).

Un ejemplo de la interacción entre la capa física y la capa de abstracción es el caso de la configuración de periféricos de I2C en un dispositivo System-On-Chip (SoC) mediante una herramienta de administración de configuración como Salt. En la capa física también se incluye todo lo necesario para ejecutar aplicaciones en el dispositivo, como por ejemplo, mediante el uso de tecnologías de contenedores como Docker.

1.3. Infraestructura cómo Código (IaC)

Las herramientas de infraestructura como código (IaC) abordan el problema de definir esta infraestructura a través un lenguaje que puede ser:

- Un lenguaje de marcado. Ansible por ejemplo, define recursos por medio de YAML, un lenguaje de marado [5].
- Un lenguaje de uso general. Pulumi puede utilizar múltiples lenguajes de uso general como Python o C# [8].
- Un DSL (lenguaje de dominio específico, del inglés Domain Specific Language). Terraform utiliza un lenguaje llamado HCL creado específicamente para la herramienta [9].

Las instrucciones de estos lenguajes pueden ser ejecutadas de dos maneras: imperativa o declarativa. En el primer caso, las definiciones son ejecutadas de forma secuencial y en el segundo, las definiciones son ejecutadas de tal forma que la infraestructura refleje las definiciones en el código independientemente de su orden.

Entonces, la Infraestructura como Código (IaC) es la definición y configuración de tanto hardware como software mediante instrucciones en un lenguaje.

1.4. Agentes

Si bien en un contexto académico la definición de agente tiene asociado un cierto grado de autonomía, según Dorri et al. un agente puede ser definido como: ≪Una entidad que está situada en un entorno y que sensa parámetros que son usados para tomar una decisión basada en un fin para la entidad. La entidad ejecuta las acciones necesarias en el entorno basadas en esta decisión.≫ [10]

Por otra parte, en la industria de la automatización de tecnologías de la información de la cual las herramientas de Infraestructura como Código forman parte, la palabra clave ≪agente≫ normalmente implica la existencia de una arquitectura cliente-servidor en la que el servidor ejecuta comandos sobre clientes (los cuales usualmente son numerosos) con el objetivo de administrarlos. Por ejemplo,

algunos sistemas de monitoreo y gestión remota (Remote and Monitoring Management en inglés) tales como NinjaOne, funcionan a través del agente que un administrador de sistemas debe instalar en los dispositivos para poder realizar monitoreo de las acciones de una máquina [11].

Como ejemplos de herramientas de IaC que utilizan agentes disponibles actualmente se encuentran Puppet y Salt [12] [13]. En contraste, Ansible, una herramienta de IaC que se autodenomina como *agentless*, ejecuta los comandos necesarios por medio de SSH en vez de requerir un servicio instalado en la máquina que el administrador desea controlar, es decir, no requiere de agente [14].

Por otra parte, existen herramientas como Terraform o Pulumi en donde la relación directa entre cliente y agente se hace más difusa. En la arquitectura descrita por Pulumi [15] y mostrada en la figura 1.1, el cliente es quien quiere que el servidor ejecute una acción por medio de una aplicación con interfaz por línea de comandos (denotada por ≪Pulumi CLI≫ en el diagrama de arquitectura).

Para efectos de esta memoria de título, se optó por diseñar una arquitectura cliente-servidor en donde el servidor controla al dispositivo IoT de forma remota mediante un agente. La razón del porqué no se escogió una arquitectura *agentless* fue debido a los siguientes antecedentes:

- Es preferible que el dispositivo IoT inicie la conexión cuando estime conveniente. Si bien el servidor posiblemente hará *push* cuando se necesite actualizar la infraestructura, es necesario que se sepa su disponibilidad para realizar esta actualización. Esto requiere un servicio que inicie la conexión al servidor, rol que puede cumplir un agente.
- Es posible que el dispositivo IoT esté en una red con tecnología CG-NAT tras la cual el abrir un puerto presenta una dificultad adicional para el administrador del sistema.

1.5. Estado

Para una herramienta IaC, el estado describe los parámetros de todos los recursos que fueron creados con ésta [16]. Es decir, si al usar la herramienta se define un usuario (asumiendo un sistema operativo similar a UNIX) con nombre `foobar` con `uid` igual a `1337`, ésta debe llevar un registro del usuario con todos los parámetros que fueron ingresados a la hora de crearlo, es decir: el `uid` y el nombre en este caso. Si bien es posible asignar un parámetro que haga referencia a un recurso que no fue creado por la herramienta, ésta no almacenará el estado de este recurso. Por ejemplo, si al usuario se le asigna un grupo con `gid` de valor `101`, la herramienta sólo registrará el `gid` como parámetro del usuario ya que sólo el usuario fue creado con la herramienta y no el grupo. En caso de omisiones (en el ejemplo original, el `gid` fue omitido) la herramienta debe asegurar que el estado sea válido, es decir, cada recurso en el estado debe tener los parámetros necesarios para garantizar su creación exitosa.

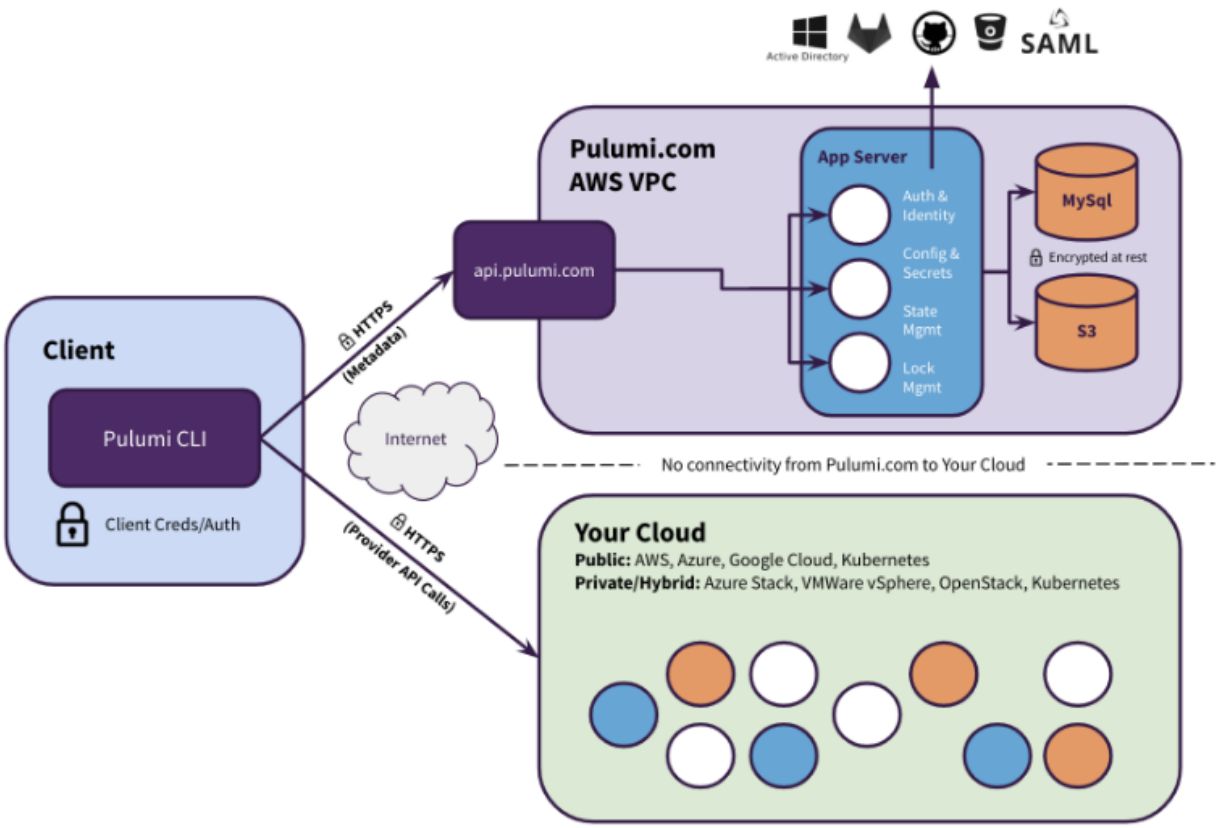

Figura 1.1: Diagrama de arquitectura de Pulumi

El estado actual (también llamado *current state*) representa el estado de todos los recursos que registra la herramienta y corresponde a la última revisión ejecutada por la herramienta. En el caso de que se realicen cambios del estado, una siguiente ejecución de la herramienta puede determinar un estado objetivo (también llamado *target state*) [16].

El objetivo de que la herramienta mantenga el estado es para que más adelante se pueda crear una nueva revisión de este. Una revisión futura con parámetros distintos o con recursos distintos realizará cambios en el sistema para que la realidad del sistema sea la misma que la representación del estado en la herramienta [16].

En nuestro ejemplo con un estado actual con un usuario de nombre `foobar`, `uid` con valor `1337` y sin `gid`:

- Si se borra la referencia al usuario en la herramienta: el usuario se elimina con `userdel`
- Si se mantiene la referencia al usuario en la herramienta y además se agrega otro usuario `foobar2`: se crea el usuario nuevo `foobar2` con `useradd`.
- Si el usuario contempla un nuevo parámetro `gid` en una nueva revisión: la herramienta usará `usermod` para actualizarlo.

Estas herramientas manejan internamente un identificador de recurso único (URN) separado del identificador del recurso en si. Por ejemplo, en la sección código 1 se crea un bucket de S3 en Amazon Web Services (AWS) mediante la herramienta Pulumi, en donde si bien el identificador único que AWS utiliza es el valor `nombreUsadoPorAWS` correspondiente a la llave `bucket`, el que usa Pulumi para generar el URN es `nombreIdentificador`.

```
import * as pulumi from "@pulumi/pulumi";
import * as aws from "@pulumi/aws";

const bucket = new aws.s3.Bucket("nombreIdentificador", {
    bucket: "nombreUsadoPorAWS",
});
```

Sección de código 1: Creación de un bucket de S3 usando Pulumi con Typescript

Una consecuencia de que la herramienta sea la que administre estos recursos es que cualquier cambio que se haga sin ella generará un *state drift*, en donde el estado actual reportado por la herramienta no corresponderá al estado real de los recursos creados [16][17].

Adicionalmente, el estado permite que los recursos sean administrados de forma declarativa. Por ejemplo, en la sección de código 2 se muestra el cambio en una pieza de código en donde el estado objetivo no contiene un recurso que el estado actual tiene. En tal caso, la herramienta eliminará el recurso `nombreIdentificador`.

```
import * as pulumi from "@pulumi/pulumi";
import * as aws from "@pulumi/aws";

const bucket = new aws.s3.Bucket(
    "nombreIdentificador", {
    bucket: "nombreUsadoPorAWS",
});
// Fin del archivo
```

Código con el estado actual

```
import * as pulumi from "@pulumi/pulumi";
import * as aws from "@pulumi/aws";

// Fin del archivo
```

Código con el estado objetivo

Sección de código 2: Ejemplo de eliminación de recurso mediante Pulumi

En contraste, Ansible no posee estado. Es decir, la herramienta describe el estado objetivo, pero al no haber un registro del estado actual, la herramienta no tiene información sobre qué recursos actualizar o eliminar.

Para el sistema propuesto en esta memoria se almacenará el estado. Las razones son que esto facilita tareas administrativas sobre el registro de los recursos creados y la herramienta puede tomar decisiones más inteligentes sobre qué recursos deben limpiarse cuando no son necesarios.

1.6. Proveedores

Si bien las herramientas IaC se encargan de llevar a cabo la creación, modificación y eliminación de recursos, a las sub-aplicaciones que realmente ejecutan estas tareas de administración se les llama

proveedores [18]. Como ejemplo, el siguiente código muestra una posible implementación de un proveedor usado para crear y destruir un directorio mediante la herramienta SSH:

```
#!/bin/bash
## nombre de archivo: ovejas
COMMAND="$1"
DIRECTORY="$2"

if [[ $COMMAND -eq 'up' ]]; then
    ssh remote-host mkdir "$DIRECTORY"
elif [[ $COMMAND -eq 'down' ]]; then
    ssh remote-host rmdir -f "$DIRECTORY"
fi
```

Sección de código 3: Ejemplo de un proveedor remoto

donde el comando `./ovejas up`, crea el recurso y el comando `./ovejas down` lo destruye. Este caso simplificado demuestra un modelo en donde el proveedor es remoto (desde la perspectiva del dispositivo IoT), toda la lógica de creación de recursos reside en el dispositivo remoto.

En contraste, una implementación en donde las instrucciones estén almacenadas de forma local (desde la perspectiva del dispositivo IoT) requiere mayor esfuerzo en la comunicación, puesto que los parámetros para la creación del recurso deben estar incluidos en la información intercambiada.

Un posible ejemplo usando sockets TCP en Python es:

```
#!/usr/bin/python
# nombre de archivo server.py
import socket, os, subprocess

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(('127.0.0.1', 3000))
sock.listen(1)

while True:
    connection, address = sock.accept()
    msg = connection.recv(1024).decode()
```

```
    # Primeros 4 bytes son el comando, el resto es el nombre del directorio
    command = msg[:4].strip() # Quitamos el padding en caso de estarlo
    directory = msg[4:]

    if command == "up":
        subprocess.run(["mkdir", os.path.expanduser(directory)])
    elif command == "down":
        subprocess.run(["rmdir", os.path.expanduser(directory)])

    connection.close()
```

Sección de código 4: Ejemplo del cliente de un proveedor local en Python

```
#!/usr/bin/python
# nombre de archivo client.py
import socket, sys

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('127.0.0.1', 3000))

if sys.argv[1] == "up":
    command = sys.argv[1] + '  ' # Padding de 2 bytes
elif sys.argv[1] == "down"
    command = sys.argv[1]
else:
    return

directory = sys.argv[2]

# Primeros 4 bytes son el comando, el resto es el nombre del directorio
sock.send((command + directory).encode())

msg = sock.recv(1024)

sock.close()
```

Sección de código 5: Ejemplo del servidor de un proveedor local en Python

Este trabajo define dos modelos de administración de recursos: uno en donde los proveedores son remotos y otro en donde son locales. En el primer caso, los proveedores remotos ejecutan código arbitrario en la máquina huésped para crear los recursos. En el segundo caso, los proveedores locales definen una interfaz sobre la cual el dispositivo remoto puede interactuar para que la máquina huésped cree los recursos. La consecuencia de esto es que para un proveedor local la lógica para administrar los recursos está en el agente y para un proveedor remoto esta lógica está fuera del agente.

En contraste, las principales herramientas de IaC orientadas a la nube trabajan principalmente

por medio de llamadas API REST a un servicio para ejecutar ≪plantillas≫ [19]. En tal caso, la herramienta de IaC está separada por varios niveles de abstracción de la máquina huésped. El modelo de ejecución local sólo aparece en casos específicos, como lo es el caso del recurso `local_exec` en Terraform y el recurso `command.local.Command` en Pulumi [20][21].

Un ejemplo de una herramienta con proveedores remotos es Ansible, en donde los proveedores (módulos de Ansible) son *scripts* ejecutados de forma remota en la máquina huésped [22]. Por otro lado, la implementación de los agentes de Salt Project es un ejemplo de una herramienta que soporta proveedores locales, en donde el servidor (Salt Master) delega la ejecución al agente (Salt Minion) [23].

En este trabajo se escogió usar proveedores locales debido a los siguientes antecedentes:

- Un modelo de ejecución remoto puede ejecutar comandos de forma arbitraria en el dispositivo, lo que tiene consecuencias graves en la seguridad en caso de existir una vulnerabilidad en la comunicación cliente-servidor. En contraste, el modelo local debe obligatoriamente interpretar el mensaje enviado por el servidor para su ejecución.
- En caso de que la comunicación presente interrupciones a la hora de crear recursos, el servidor debe incluir lógica adicional para garantizar idempotencia en la creación de recursos. Por ejemplo, si la herramienta crea varios directorios y la conexión se interrumpe después del primero, una nueva ejecución fallará al intentar crear el primer directorio a menos que se verifique su existencia.
- Se hace posible serializar todo el estado objetivo y enviarlo desde el servidor al cliente una sola vez para su ejecución. Si se interrumpe la conexión en un modelo de ejecución remoto puede crearse una inconsistencia entre el estado real y el estado del dispositivo que el servidor tiene almacenado.

1.7. Pilas

Una pila (o *stack* en inglés) es una agrupación de recursos que son administrados como una sola unidad. Este concepto es utilizado por herramientas de infraestructura como código para crear recursos a partir de una misma pieza de código, pero que estos recursos sean administrados de forma separada entre cada ejecución [24].

Un caso concreto es la implementación de una aplicación bajo la metodología de 12 factores. Uno de sus pilares es que un entorno de producción debe estar separado del entorno de desarrollo, pero debe existir una paridad entre ellos para poder replicar los cambios del entorno de desarrollo hacia producción. En ambos casos, la infraestructura debiese tener la misma estructura para garantizar la paridad, y además asegurar que los recursos creados sean distintos para aislar ambos entornos [25]. Una herramienta de IaC puede facilitar la tarea mencionada anteriormente utilizando una misma pieza de código para cada entorno, pero cambiando los recursos en cada pila mediante variables insertadas en tiempo de ejecución.

1.8. Proyectos

Durante el transcurso de este trabajo se hablará de «proyectos», un término que se utilizará para una aplicación que está escrita en un lenguaje de uso general y que además tiene sus propias dependencias, su propia configuración, un directorio base y su propio nombre. Un proyecto define recursos bajo el lenguaje y al ejecutarse, creará un archivo de estado el cual puede ser usado para crear, modificar o eliminar recursos en cada agente.

Por ejemplo, un proyecto de Python puede tener sus dependencias administradas mediante *Poetry* y ser configurado mediante un archivo `pyproject.toml` en la raíz del proyecto, en donde además se le asigna el nombre.

1.9. Comparación con otras herramientas de IaC

Si bien durante las secciones anteriores se realizaron comparaciones con distintas herramientas de IaC, no todas estas tienen el mismo objetivo. Por ejemplo, si bien Terraform y Pulumi son herramientas IaC, su enfoque es la administración de recursos de proveedores de nube. En contraste, Salt, Chef y Ansible están orientadas a la administración de configuración de dispositivos, el mismo fin de la herramienta expuesta en este trabajo.

La primera diferencia es que, ya que Terraform y Pulumi están orientadas a la administración de recursos en la nube, no es necesario que usen agentes; simplemente hacen peticiones HTTP al proveedor de los recursos. En contraste, la distinción es muy útil en las herramientas Salt, Chef y Ansible ya que éstas configuran los dispositivos cuando están disponibles. En este caso, existen herramientas que no necesitan instalar un agente para configurar los dispositivos (configuración *agentless*) y las que sí lo necesitan (configuración *agent-based*). [14]

Otra diferencia entre herramientas IaC es el lenguaje para definir los recursos. Por ejemplo, Ansible usa un lenguaje de marcado (llamado *YAML*) para definir recursos en el dispositivo. A su vez, Chef, Pulumi y Salt utilizan lenguajes de dominio específico para definir los recursos de forma declarativa. Por último, Pulumi define los recursos mediante un lenguaje de uso general. Entre los lenguajes soportados por esta última herramienta están C#, Python y TypeScript. [15]

La última diferencia corresponde a la de los proveedores. La distinción de proveedores locales y remotos expuesta en la sección 1.6 no aplica para Terraform y Pulumi, puesto que estos no se comunican directamente con un dispositivo. Si bien muchos de los elementos de comparación no aplican para las herramientas Pulumi y Terraform ya que estos administran recursos de proveedores de nube, estas sí tuvieron una gran influencia en el diseño final de la herramienta de este trabajo. Para el caso de Salt y Ansible, los comandos están definidos en el dispositivo remoto que controla al dispositivo IoT y delegan la ejecución al dispositivo enviando código de Python y ejecutándolo en el cliente [26] [27]. En contraste, el funcionamiento de Chef consiste en que el dispositivo remoto envía un plan de ejecución y el dispositivo IoT debe descargar los proveedores encargados de realizar los cambios sobre la infraestructura mediante un agente [28].

El siguiente cuadro comparativo presenta las diferencias entre las decisiones tomadas durante las secciones anteriores frente a otras herramientas de uso similar.

Nombre	Agente	Lenguaje	Tipo de proveedor
Terraform	No aplica	DSL propio	No aplica
Pulumi	No aplica	Lenguaje de uso general	No aplica
Salt project	Agent-based & agentless	DSL Propio	Remoto
Chef	Agent-based	DSL Propio	Local
Ansible	Agentless	YAML	Remoto
Ovejas (este trabajo)	Agent-based	Lenguaje de uso general	Local

Cuadro 1: Tabla comparativa entre diferentes herramientas de IaC

Finalmente, existen diferencias adicionales entre las herramientas de administración de configuración, puesto que las decisiones de diseño en la arquitectura de la herramienta impactan su viabilidad en dispositivos IoT. Por ejemplo, Ansible y Salt requieren que Python esté instalado en el dispositivo y Chef requiere de una instalación de Ruby en el dispositivo. Esta diferencia es notable considerando que en dispositivos IoT frecuentemente tienen limitaciones en el uso de memoria.

2. Diseño del sistema

2.1. Arquitectura

El sistema propuesto en este documento comprende los siguientes componentes:

1. Un programa de Python (**main.py**) que representa el estado objetivo en un lenguaje de programación desde la máquina de un usuario.
2. Un **coordinador de agentes** que distribuye las actualizaciones de estado en un conjunto de dispositivos IoT.
3. Una herramienta por interfaz de línea de comandos (**CLI**) que se encarga de ejecutar código mediante un entorno de ejecución (**Ovejas Runtime**).
4. Un conjunto bibliotecas **SDK Ovejas** y un **SDK Proveedor**, en donde el primero permite poder tener una representación del estado a partir del código y el segundo define los diferentes recursos que pueden ser creados mediante el lenguaje de programación.
5. Una **base de datos SQLite** que almacena los archivos de estado y a qué dispositivos IoT se deben distribuir.
6. **Proveedores**, los cuales permiten que el agente interactúe con los recursos que se deben crear, modificar y/o eliminar.

El diagrama de arquitectura de la figura 2.1 presenta una vista general de cómo los subsistemas interactúan entre sí. Finalmente, las siguientes secciones presentan las razones que influyeron en el diseño de esta arquitectura.

2.2. Caso de uso

El caso de uso al que la herramienta de este documento está dirigida a una organización que tiene las siguientes necesidades:

- La organización que administra los dispositivos IoT tiene interés en integrarlos en procesos de integración continua y despliegue continuo.
- Los cambios de infraestructura mediante la actualización de una imagen inmutable del sistema operativo se considera muy costoso, pero administrar cada dispositivo manualmente también es demasiado costoso.
- La infraestructura que se quiere definir no es la misma en cada dispositivo.
- Los dispositivos están en una red inestable, por lo que la conexión puede interrumpirse en medio de una operación de actualización de infraestructura.

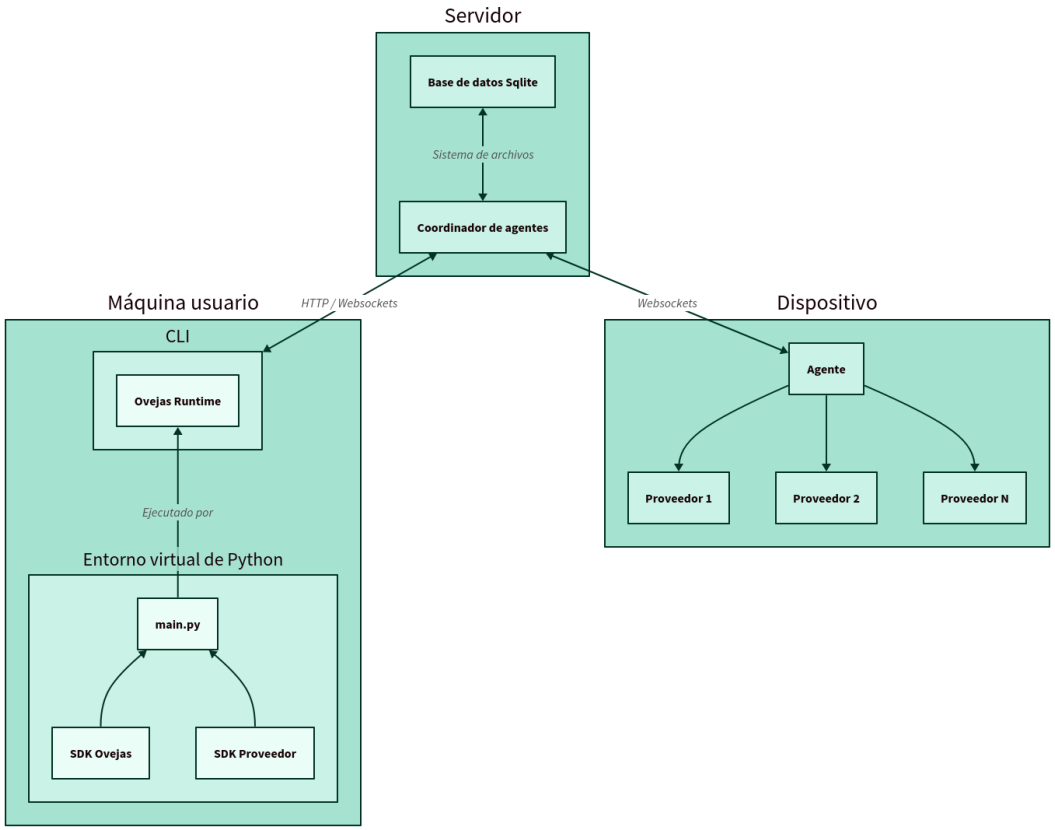

Figura 2.1: Diagrama de la herramienta de IaC

Además, la organización a la que esta herramienta va dirigida debe tener las siguientes capacidades:

- Los dispositivos tienen un sistema operativo de uso general como Linux o Windows.
- La organización debe tener disponible en su infraestructura un servidor en el cual ejecutar el Coordinador de Agentes.

Por otra parte, se tienen las siguientes consideraciones prácticas para el uso de la herramienta:

- La herramienta CLI puede ser ejecutada por una persona o por una máquina.
- Los proveedores son un proceso colaborativo entre la comunidad que mantiene la herramienta y la organización que la utiliza, por lo que es extremadamente deseable dar facilidades y abstracciones para hacer la tarea de crearlos lo más fácil posible. Por ello, la arquitectura contempla una biblioteca para crear y registrar proveedores por el lado de la herramienta de CLI.

2.3. Historia de usuario

Esta herramienta busca ayudar en un flujo de trabajo que consiste en que el usuario previamente haya instalado:

- El **agente** en todos los dispositivos que quiera controlar.
- El **coordinador de agentes** en un servidor públicamente accesible.
- La **herramienta CLI** desde una máquina en donde se tenga el código en donde se definen los recursos.

Luego, el usuario deberá registrar los dispositivos en el coordinador de agentes y, posteriormente, ejecutar el código del usuario mediante la herramienta CLI. Una ilustración del flujo de trabajo completo puede verse en la figura 2.2, en donde también se muestran etapas intermedias que serán detalladas en secciones posteriores.

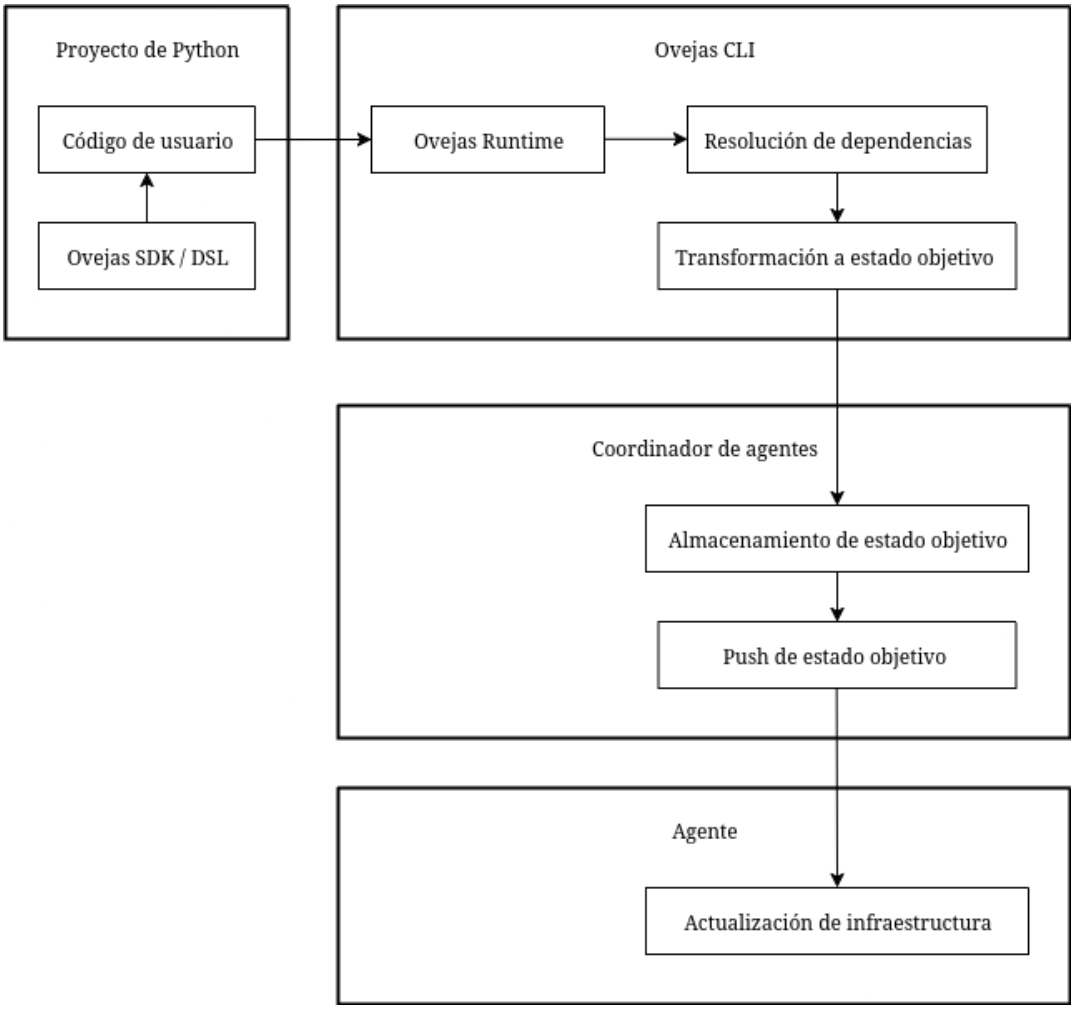

Figura 2.2: Flujo de trabajo normal con la herramienta

2.4. Alcances y limitaciones

Sin desmedro de las secciones anteriores, este trabajo presenta los siguientes alcances y limitaciones:

- Sólo se implementó un proveedor de administración de usuarios. Por lo que, si bien en los ejemplos posteriores se hace referencia a proveedores adicionales, no fueron implementados.
- No se implementaron transacciones para revertir la ejecución de estado objetivo.
- Si bien se consideraron mecanismos para soportar autenticación y autorización en el diseño, no se implementó.
- Si bien la biblioteca puede ser en cualquier lenguaje de uso general y bajo cualquier entorno, sólo se usará Python con el administrador de paquetes Poetry.
- Los agentes sólo pueden ejecutarse en una máquina con el sistema operativo Linux.

3. Lenguaje de dominio específico embebido (eDSL)

En este sistema propuesto, se propone crear una biblioteca de Python para abordar la tarea de crear los recursos mediante la herramienta de CLI. Para ello, se crearán modificaciones al comportamiento normal del lenguaje para añadirle nuevas extensiones, convirtiéndolo en un lenguaje de dominio específico embebido (conocido como *eDSL* o *internal DSL* en inglés) [29] para la tarea de crear, actualizar y eliminar recursos mediante el agente.

3.1. Definición de recursos

Para la definición de recursos en la herramienta propuesta en este documento, se tomó en consideración la decisión de diseño de Pulumi en donde la instanciación de una clase implica la creación de un recurso. Para ello, primero se creó el decorador `@registry` que define un constructor para cada campo definido (de manera similar al decorador `@dataclass`) y más importante, intercepta la llamada a la función `__call__()` que es llamada cada vez que se instancia una clase en Python. De esta forma, se puede llevar un registro de todas las clases instanciadas y, por tanto, de los recursos del estado del programa.

```
class ResourceRegistry:
    _registered_resources = []

    def __init__(self, cls):
        ...
        cls.__init__ = create_init(cls) # Creación dinámica del constructor
        self.cls = cls
    ...
    def __call__(self, *args: Any, **kwargs: Any) -> Any:
        object_instance: Resource = self.cls(*args, **kwargs) # Instanciación
        object_urn = self.get_uri(object_instance.resource_name)
        ...
        # Registro del recurso y sus parámetros
        ResourceRegistry._registered_resources.append({
            'urn': object_urn,
            'parameters': kwargs,
        })
        return object_instance # Retorno del objeto creado post intercepción
```

Sección de código 6: Registro de recursos

Una posible implementación del decorador es la descrita en la sección de código 6, en donde registra todos los recursos en la variable de clase `_registered_resources` para su posterior procesamiento.

Es importante notar que la generación dinámica de clases dificulta el análisis del código por un analizador de tipado estático como *MyPy* o *pyright* ya que no es posible determinar el constructor sin

ejecutar el código. Notablemente, es posible anotar el constructor creado dinámicamente mediante el decorador `@typing.dataclass` debido a que `ResourceRegistry` creará el constructor de la misma forma en que lo hace el decorador `@dataclasses.dataclass` [30].

Finalmente, el decorador queda como en la sección de código 7 y se muestra en la sección de código 8.

```
@typing.dataclass_transform()
def register(cls: type[T]) -> type[T]:
    return cast(type[T], ResourceRegistry(cls))
```

Sección de código 7: Decorador register

```
# Definición de clase con campos y creación dinámica del constructor
@register
class User:
    resource_name: str
    name: str
    uid: int

    def hello_uid(self):
        result "hola " + str(uid)

user = User("hola", "foobar", 1337) # Intercepción de __call__(), llamada a
                                    # constructor y registro

print(type(user)) # >>> <class '__main__.User'>
print(user.uid) # >>> 1337
print(user.hello_uid())
# >>> "hola 1337"; La intercepción debe dejar las funciones miembro intactas
print(ResourceRegistry._registered_resources)
# >>> [{
#   'urn': '__main__::User::hola',
#   'parameters': {
#       'resource_name': 'hola',
#       'name': 'foobar',
#       'uid': 1337
#   }
#}]
```

Sección de código 8: Ejemplo de definición de un recurso

Teniendo en cuenta lo anterior, más adelante en este documento se abordarán posibles formas de usar para generar un archivo de estado a partir de la variable de clase `_registered_resources`.

3.2. Ejecución del lenguaje

El objetivo final del lenguaje embebido es poder generar un archivo de estado después de ejecutar el archivo de Python. Sin embargo, para que el usuario no tenga que crearlo manualmente accediendo al archivo de clase, es necesario crear un entorno de ejecución (conocido como *runtime environment* en inglés) en el cual se ejecute el archivo del usuario y extraiga el valor escrito en la variable de clase `_registered_resources`.

Para realizar este entorno de ejecución se tomó la decisión de usar el mismo lenguaje Python para ejecutar el archivo del usuario así como también para extraer los recursos definidos en este. En la sección de código 9 se puede observar una posible implementación del entorno de ejecución.

```python
def execute(command: str, main_path: str, lib_path: str):
    sys.path.append(lib_path)
    program = open(main_path, 'r').read() # Lectura del archivo
    ...
    # Garantizamos que el código por el usuario en la
    # sección de código '__main__' se ejecutará.
    executor_context: dict[str, Any] = {
        '__name__': '__main__',
    }
    exec(program, executor_context) # Ejecución del archivo
    ...
    resource_class = executor_context['ResourceRegistry']
    ...
    return resource_class._registered_resources # Recursos registrados
```

Sección de código 9: Entorno de Ejecución

El funcionamiento de este entorno de ejecución se puede desglosar en los siguientes pasos:

- El argumento `lib_path` establece la ruta en donde Python puede resolver los módulos.
- La variable `executor_context` que establece la variable global `__main__` en el entorno de ejecución al valor que corresponde por defecto. Esto mantiene el comportamiento normal de Python cuando un archivo se ejecuta directamente.
- El argumento `main_path` establece la ruta del archivo sobre el cual el usuario define recursos mediante la instanciación de clases.
- Después de la evaluación del código del usuario, se puede obtener el objeto global de la clase `ResourceRegistry`. Esto es posible porque en Python, todas las definiciones de clases son objetos de ámbito global.
- Finalmente, a partir el objeto global de la clase `ResourceRegistry` se obtiene la variable de clase `_registered_resources`, la cual contiene todas las definiciones de recursos del usuario.

Después del proceso anteriormente descrito, es importante notar que el valor retornado por la función `execute` hace posible generar un archivo de estado objetivo que puede ser posteriormente utilizado en una aplicación de línea de comandos. Un proceso similar ocurre en el caso de Terraform que utiliza el comando `terraform show -json` para mostrar la representación del estado objetivo en un JSON.

3.3. Resolución de recursos

Normalmente, la semántica de los lenguajes en herramientas de IaC es que la creación, eliminación y actualización de recursos se ejecute de forma concurrente y/o paralela.

La principal razón de esta decisión es que muchos de los recursos reflejan datos que son obtenidos de la web, en donde existe un tiempo de espera a que ciertas consultas sean resueltas. La dependencia entre recursos normalmente es gestionada mediante valores futuros (también conocidos como *futures* o *promises*), en donde estos son resueltos en tiempo de ejecución de forma asíncrona.

Adicionalmente, esta relación de dependencia se denota cuando un recurso le otorga un valor futuro a otro [31]. En caso de que esta relación no sea directa (un recurso depende de que otro esté creado, pero el dependiente no acepta un valor futuro de su dependencia en el lenguaje) es necesario designar una directiva del tipo `dependsOn` de forma similar a la de las herramientas Docker Compose y Pulumi [32] [33].

```
@dataclass
class CurlResponse(Resolvable):
    status_code: int

    @staticmethod
    def resolve(dependent: 'Curl') -> 'CurlResponse':
        result = subprocess.run(['curl', '-I', cast(str, dependent.url)], ...)
        status_code = int(result.stdout.splitlines()[0].split()[1])
        return CurlResponse(status_code=status_code)

@register
class Curl(Resource):
    resource_name: str
    url: Result[str] | str

    response: Result[CurlResponse] = resolve(CurlResponse)

    def _set_dependents(self):
        self.response.set_dependency(weakref.ref(self))

req = Curl('req', 'https://www.google.com/')
print(req.response) # >> objeto de tipo Result
print(req.response.resolve()) # >> CurlResponse(status_code=200)
```

Sección de código 10: Ejemplo de un proveedor con dependencia que debe ser resuelta

Lo que esta memoria de título propone es que cada recurso que sea una dependencia que deba ser resuelta dinámicamente lo haga mediante un método estático `resolve()` que instancie el objeto resuelto como en la sección de código 10, en donde se muestra un posible caso en donde es necesario resolver el recurso de forma asíncrona.

```
class ResourceRegistry:
    ...
    def __call__(self, *args: Any, **kwargs: Any) -> Any:
        object_instance: Resource = self.cls(*args, **kwargs)
        ...
        object_instance._set_dependents() # Asignación de dependientes
        ...
        return ...
```

Sección de código 11: Asignación de dependientes

En tal caso, mediante la función miembro `_set_dependents`, el dependiente `Curl` le otorga sus parámetros a la dependencia `CurlResponse`. Esta asignación es hecha mediante la intercepción de `__call__()` como es mostrado en la sección de código 11 para que sea resuelta cuando se estime conveniente. Finalmente, una clase Result puede ser implementada de la forma mostrada en la sección 12.

```
class Result(Generic[T]):
    def __init__(self, resolvable: type, dependency: Any = None):
        self.resolvable = resolvable
        self.dependent = dependency

    def set_dependent(self, dependent: Any):
        self.dependent = dependent

    def resolve(self) -> 'Result[T]':
        return self.resolvable.resolve(self.dependent)

def resolve(cls):
    return Result(cls)
```

Sección de código 12: Clase Result

3.4. Dependencias entre resoluciones

La ventaja de realizar el trabajo de la sección anterior es que los recursos pueden ser resueltos de manera asincrónica. Es decir, el programa completo del usuario será evaluado por el intérprete de Python y en una etapa posterior, cada recurso será definido o evaluado (un ejemplo de un recurso evaluado corresponde al de `Curl`, que representa un recurso externo). El objetivo final de esta evaluación posterior de recursos es poderlos evaluar de manera concurrente, ya que estos recursos externos frecuentemente están limitados por operaciones de entrada y salida.

La complejidad de lo anteriormente descrito se debe a que un recurso puede depender de otro, y a su vez, otro recurso puede depender de este último, por lo que debe existir un mecanismo para ordenar la evaluación de estos recursos a partir de su relación de dependencia.

La obtención del orden de evaluación puede ser primero modelado como un problema de grafos, en donde cada recurso representa un nodo y una relación de dependencia representa un vértice del recurso a su dependiente. Este problema puede ser ejecutado de forma concurrente obteniendo el orden topológico mediante el algoritmo de Kahn [34] para luego llamar a `resolve()` en cada recurso mediante hebras o procesos, cuidando de que todos los recursos de los que se dependa ya hayan sido resueltos.

Debido a la complejidad de realizar la resolución de cada recurso de forma concurrente (y su baja utilidad), este paso fue realizado como prueba de concepto. Por lo tanto, las llamadas a `resolve()` fueron implementadas de forma bloqueante.

4. Gestión del estado

4.1. Esquema del archivo de estado

Como se mencionó en la sección «1.5 Estado», la herramienta trabajará con una representación del estado en el formato JSON. Recalcando lo mencionado en esa sección, si bien esta representación puede ser en otros formatos como XML o incluso en una representación binaria, la ventaja de esta representación es que es más fácil de modificar tanto manualmente como con herramientas dedicadas a su edición (tales como `yq` o `jq`).

En primera instancia, se crea una representación de estado generada a partir del código del usuario por medio de la herramienta CLI. Luego, esta última envía este estado al Coordinador de Agentes, el cual finalmente envía este estado a los dispositivos correspondientes.

Teniendo en cuenta lo anterior, se estableció un esquema de representación de estado con el objetivo de que cada parte del sistema pueda obtener la información pertinente desde el archivo de estado. En consecuencia, el archivo de estado debe tener el siguiente formato:

- Cada recurso tiene un atributo de identificador único de recurso (URN).
- El atributo `parameters` que incluye todos los parámetros para la actualización y creación de un recurso para el proveedor.
- Para designar relaciones de dependencia entre recursos, cada recurso puede darle un valor el atributo `depends_on` con una lista de identificadores únicos.

Por otra parte, el formato del URN está conformado por tres secciones separadas por dos caracteres del signo de puntuación «dos puntos». La primera sección detalla el espacio de nombres del proveedor, la segunda sección indica el nombre del proveedor que ejecutará la acción y una tercera sección indica el nombre del recurso. Un ejemplo de un archivo de estado con el formato detallado en esta sección puede verse en la sección de código 13.

```
{
  "version": 1,
  "created_at": "2025-03-04 00:27:58.953802",
  "resources": [
    {
      "urn": "ovejas.system::User::my_user",
      "parameters": {
        "gid": 0,
        "name": "example_user",
        "uid": 4096
      },
      "depends_on": []
    }
  ]
}
```

Sección de código 13: Ejemplo de un archivo de estado con un recurso

Adicionalmente, la raíz del archivo de estado incluye un atributo `version` con la versión del esquema. Algo que permitiría en un futuro realizar cambios en el esquema sin necesariamente perder la compatibilidad de los archivos de estado ya creados con la herramienta.

4.2. Diferencias en el estado

Teniendo en cuenta que se trabajará con el formato JSON, una diferencia de estado puede ser calculada comparando los recursos en la llave `resources`. En donde es importante notar que al existir una diferencia debiese ocurrir lo siguiente en el agente:

- Si un recurso está en el estado actual del agente, pero no lo está en el estado objetivo que el administrador determinó, este debe eliminarse.
- Si un recurso no está en el estado actual del agente, pero está en el estado objetivo, este debiese crearse.
- Si un recurso se encuentra tanto en el estado actual agente como en el estado objetivo de la herramienta CLI, pero los parámetros entre ambos estados son distintos, este recurso debe actualizarse en el dispositivo para que su estado actual corresponda al de los parámetros del estado objetivo.

Para el caso de la sección de código 14, por ejemplo, el agente elimina al usuario `example_user_a` con `uid` de valor `4000` y crea al usuario `example_user_b` con `uid` de valor `4001`.

```
{
    "version": 1,
    "created_at": "2025-03-04 00:27:58.953802",
    "resources": [
        {
            "urn": "ovejas.system::User::my_user_a",
            "parameters": {
                "gid": 0,
                "name": "example_user_a",
                "uid": 4000
        },
        "depends_on": []
        }
    ]
}
```

```
{
  "version": 1,
  "created_at": "2025-03-04 00:27:59.953802",
  "resources": [
    {
      "urn": "ovejas.system::User::my_user_b",
      "parameters": {
        "gid": 0,
        "name": "example_user_b",
        "uid": 4001
      },
      "depends_on": []
    }
  ]
}
```

Sección de código 14: Diferencia de estado. Izquierda representa estado actual y la derecha representa el estado objetivo

5. Herramienta por línea de comandos

La herramienta por línea de comandos (CLI por sus siglas en inglés) tiene como principal función el distribuir el estado objetivo al servidor. Para ello, la herramienta ejecuta el programa de Python descrito en la sección 3.2 por medio de PyO3, el cual permite llamar a código de Python desde Rust. Finalmente, esto permite ejecutar un programa de Python que use una biblioteca similar a la descrita en la sección de código 8 y obtener una representación en JSON del estado objetivo.

El objetivo secundario de la herramienta CLI es la administración del servidor en sí. Es decir, realizar tareas administrativas como la creación de usuarios, entornos y el enrolamiento de nuevos dispositivos.

5.1. Comunicación con el servidor

Existe una disyuntiva en la comunicación entre el cliente de CLI y el servidor. Por una parte, conviene que las tareas de administración de recursos tengan una comunicación constante entre el servidor, de tal forma que sea posible tener una actualización en tiempo real de cómo progresa una actualización en el dispositivo. Por otra parte, existen otras que pueden ser ejecutadas una sola vez, como lo son la creación de un usuario o el enrolamiento de un dispositivo.

Teniendo en cuenta lo anterior, se tomó la decisión de diseño de soportar conexiones por WebSockets y por HTTP. Si el usuario decide iniciar una conexión HTTP por medio de la ruta `/socket` se cambiará el protocolo a WebSocket. Cualquier otra ruta se tomará como una petición HTTP sin cambio de protocolo.

Adicionalmente, para asegurar el cumplimiento del protocolo WebSocket el cliente CLI agregará las cabeceras correspondientes. Por último, como las acciones de mantención deberían ser permitidas sólo para un administrador de los dispositivos, se agregará un mecanismo de autorización por medio de la cabecera `Authorization`.

6. Agente

El agente tiene como función procesar el estado objetivo recibido desde el coordinador de agentes y crear, actualizar o eliminar recursos dinámicamente. Este agente debe ser ejecutado como un servicio en segundo plano durante el funcionamiento del dispositivo. Adicionalmente, el usuario bajo el cual se ejecute el agente debe tener los privilegios correspondientes para crear, modificar o eliminar los recursos que se deseen administrar.

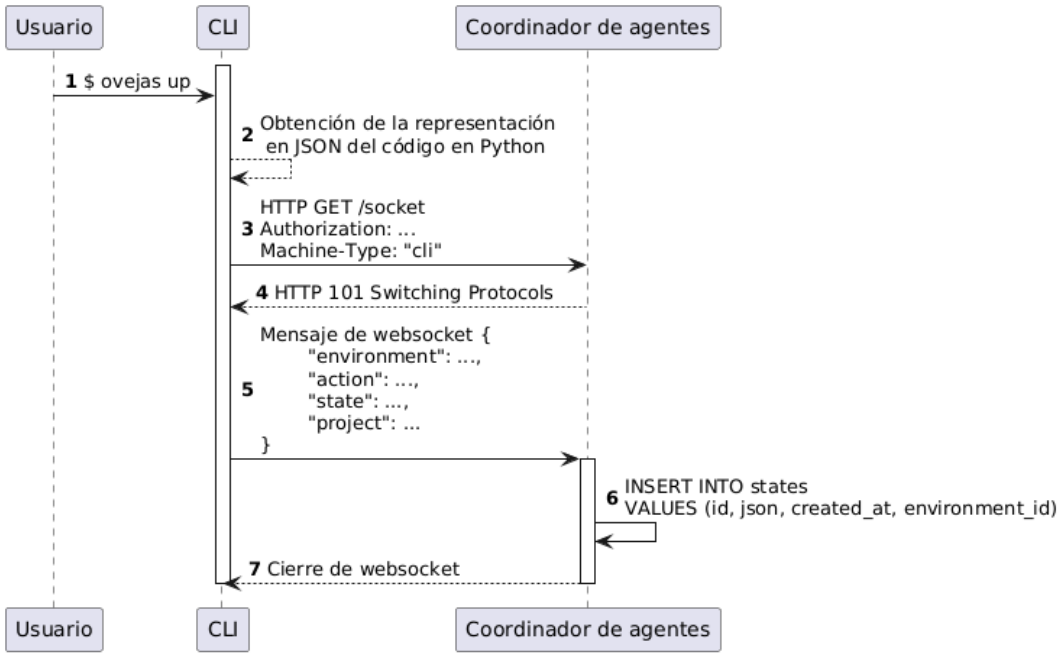

Figura 6.1: Diagrama de secuencia para la interacción usuario-servidor

6.1. Comunicación con el servidor

A diferencia del cliente por línea de comandos, el agente está implementado exclusivamente con WebSockets.

El desglose de los pasos en la comunicación con el coordinador de agentes del diagrama de secuencia de la figura 6.2 es como sigue:

1. El dispositivo se conecta al servidor mediante una petición HTTP con el método GET a la ruta `/socket`. Al igual que la herramienta CLI, incluye la cabecera ≪Authentication≫ para autenticar al dispositivo, la cabecera ≪Machine-Id≫ para identificarlo y la cabecera ≪Machine-Type≫ con valor ≪Device≫ para identificar al cliente como agente.
2. Respuesta HTTP 101 para iniciar el protocolo WebSocket.
3. El servidor envía un mensaje pidiendo el estado del dispositivo.
4. El dispositivo responde con el estado del dispositivo, una marca de tiempo y un diccionario

llave-valor que mapea el nombre de un entorno a un hash del estado de sus recursos. Los detalles de este proceso están definidos en la sección 6.2.

5. En caso de que el hash del estado en el servidor sea distinto al del cliente, el servidor responderá con un mensaje `UpdateEnvironmentsRequest` cuyo *payload* contiene el estado serializado conforme al detallado en la sección 4.1. En caso contrario, el coordinador de agentes volverá a pedir el estado y se repetirá el proceso desde el paso 3.
6. Envío del estado.
7. Actualización del estado en el dispositivo.

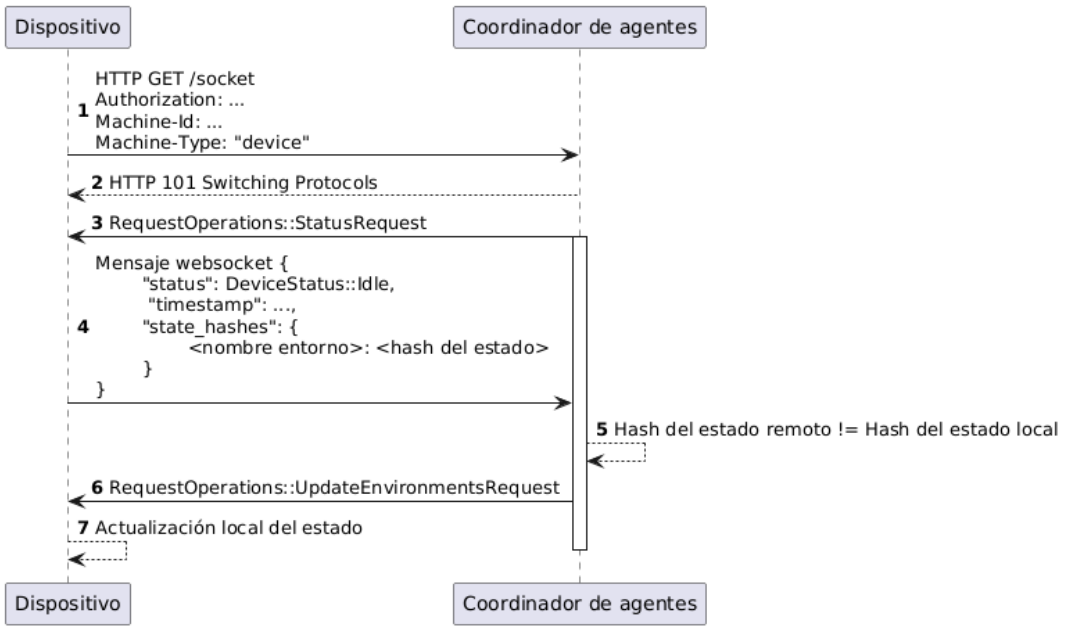

Figura 6.2: Diagrama de secuencia para la interacción dispositivo-servidor

6.2. Administración de estado

Como se mencionó en la sección anterior, los pasos del 4 al 6 de la figura 6.2 corresponden a los principales pasos de la actualización del estado.

El primer aspecto a destacar es el mapa llave-valor `state_hashes` que el agente envía al coordinador de agentes mediante hashes MD5 de los archivos de estado almacenados en el dispositivo. Por ejemplo, en la implementación de referencia, los archivos de estado se almacenan con nombre de archivo en el formato `nombre_de_entorno.state.json` y en el subdirectorio `.ovejas/` del directorio *home* correspondiente al usuario que ejecute el agente.

Asumiendo el caso que sea la primera vez que el agente se conecte al servidor y no exista ningún archivo de estado almacenado en el dispositivo, el mapa `state_hashes` estará vacío. En tal caso, el servidor envía como respuesta un mensaje `UpdateEnvironmentsRequest` con el estado de cada entorno faltante.

En caso de que ya exista un entorno, se comparan los recursos uno a uno entre el estado actual del archivo de estado local y el estado objetivo del mensaje `UpdateEnvironmentsRequests`. Luego, en caso de existir una diferencia entre el estado objetivo y el actual almacenado en el dispositivo, el agente ejecuta los cambios requeridos mediante los proveedores de los recursos cuyos parámetros sean distintos entre el estado actual y el estado objetivo. Finalmente, el estado objetivo reemplaza al archivo de estado actual almacenado en el dispositivo.

6.3. Proveedores locales

Como se mencionó previamente en la sección 1.6, se decidió que los proveedores sean implementados de forma local.

Para la implementación de referencia se tomó la decisión de que los proveedores sean implementados mediante una biblioteca enlazada estáticamente con el binario del Agente. Sin embargo, esta decisión dificulta la tarea de agregar nuevos proveedores, ya que para agregar un nuevo proveedor, se debe crear una biblioteca adicional y recompilar el binario del Agente.

En la práctica, un proveedor debe ser capaz de interactuar con la herramienta que abstrae, aún cuando esta herramienta descarte o cambie funcionalidad en su interfaz de programación para aplicaciones (API). En consecuencia, el binario del agente deberá almacenar múltiples versiones incompatibles y/o innecesarias de cada proveedor.

Un ejemplo concreto de lo descrito anteriormente es el caso de un proveedor que abstrae la herramienta ≪Docker≫: si el proveedor que está en el binario del agente sólo soporta la versión 19.1.1 de Docker pero la versión de Docker en el dispositivo IoT es 20.1.1, el primer número indicaría que existe un cambio incompatible en la API [35]. En tal caso, la herramienta de infraestructura como código propuesta en este documento dejaría de funcionar a menos que almacene un proveedor de ≪Docker≫ para cada versión que rompa la compatibilidad.

En contraste, tanto los proveedores de Terraform como de Pulumi implementan estos proveedores mediante servidores web, los cuales están almacenados en un repositorio de proveedores (Terraform Registry y Pulumi Registry) hospedado en la infraestructura de las compañías que mantienen las herramientas [18][36]. De las herramientas mencionadas anteriormente, cada uno de sus proveedores funciona como un servidor web que se comunica con su respectiva herramienta mediante HTTP y gRPC [37]. Esta decisión presenta la ventaja de que desacopla por completo la implementación de los proveedores del Coordinador de Agentes.

Para un futuro trabajo se propone realizar la implementación de proveedores mediante HTTP y gRPC de manera similar a las herramientas mencionadas anteriormente.

6.4. Actualizaciones de estado

Como se explicó en la sección 4.2, para poder actualizar los recursos en el dispositivo de tal manera que correspondan al estado objetivo, es necesario saber qué recursos crear, actualizar y/o eliminar. Es por esto que en el evento de que el coordinador de agentes envíe un mensaje de tipo `UpdateEnvironmentsRequest` con el nuevo estado objetivo, el agente realiza una comparación

recurso a recurso entre el estado serializado del servidor (el estado objetivo) y el estado actual almacenado en el dispositivo. Esta comparación es posible debido a que el URN de un recurso debe ser el mismo tanto en el estado actual como en el estado objetivo.

Luego de que el agente haya determinado qué acciones tomar para cada recurso a partir de los criterios de la sección 4.2, el agente usará el URN para obtener un objeto proveedor del recurso. Este objeto proveedor debe implementar la interfaz `ResourceProvider` que implementa los métodos `update()`, `destroy()`, `create()` y en cada uno debe ejecutar las acciones correspondientes para actualizar el recurso. Por ejemplo, la implementación del recurso `User` que abstrae a un usuario en Linux, implementa el método create por medio del comando `useradd`.

7. Coordinador de agentes

El coordinador de agentes tiene como objetivo distribuir las actualizaciones de estados objetivos a cada agente para que estos sean ejecutados en el dispositivo por medio de un proveedor de recursos.

7.1. Conexión con los clientes

Para la conexión con los clientes se utilizan las bibliotecas Tokio y Hyper. El primero proporciona un entorno de ejecución asíncrono que le permite al cliente conectarse de forma concurrente y el segundo es una biblioteca que proporciona abstracciones para trabajar con el protocolo HTTP. Adicionalmente, se utiliza la biblioteca Tungstenite que permite trabajar con WebSockets.

Como se explicó en secciones anteriores, existen dos tipos de clientes: uno para el Agente y otro para la herramienta CLI. Al existir dos tipos de clientes, es necesario determinar qué protocolo se usará para la conexión. Para ello, la conexión inicial al servidor **siempre** será mediante el protocolo HTTP.

Existen dos componentes que determinan el tipo de conexión. El primer componente es la cabecera ≪Machine-Type≫ que puede tener como valor ≪device≫ o ≪cli≫; el primero siendo una comunicación con un agente y el segundo con la herramienta de CLI. El segundo componente es la ruta en donde se hace esta petición que sería el caso de que se trate de un cliente de tipo CLI; si la ruta de la petición HTTP es `/socket`, la conexión se cambiará a WebSocket y en caso contrario, será una petición HTTP normal sin cambio de protocolo. Este último caso se puede ver de mejor forma en la figura 6.1.

Por último, los clientes de tipo ≪device≫ deben además incluir la cabecera ≪Machine-Id≫ con el objetivo de poder consultar el estado del despliegue de la infraestructura en el dispositivo.

7.2. Autenticación y autorización

De acuerdo con los alcances, si bien no se implementó Autenticación ni Autorización como tal, sí se consideró en el diseño. Es por ello que la implementación de referencia incluye mecanismos básicos para limitar el acceso a dispositivos que no están registrados.

La especificación del protocolo de WebSockets admite que el *handshake* inicial contenga cabeceras adicionales (tales como `Authorization`) justamente para soportar autenticación y autorización [38]. La implementación de referencia hace uso de esta cabecera con el objetivo de facilitar una futura implementación con protocolos de autorización que utilicen *tokens* de acceso como OAuth [38].

7.3. Modelo de base de datos

Para el almacenamiento de los datos se utilizó la base de datos SQLite, el cual mediante el sistema de archivos guarda la base de datos en un solo archivo [39]. Esto lo hace ideal para pruebas de concepto y cargas de trabajo livianas [40].

Adicionalmente, se utilizó la biblioteca de Mapeo Objeto-Relacional (ORM por sus siglas en inglés) Diesel para facilitar una posible migración a otro sistema de gestión de bases de datos relacionales (RDBMS) como PostgreSQL sin alterar la lógica de negocio.

Además, las conexiones a la base de datos fueron gestionadas con Deadpool, un gestor de conexiones a la base de datos. Este último ayuda a garantizar la exclusión mutua para la lectura y escritura en la base de datos entre cada conexión concurrente al servidor web.

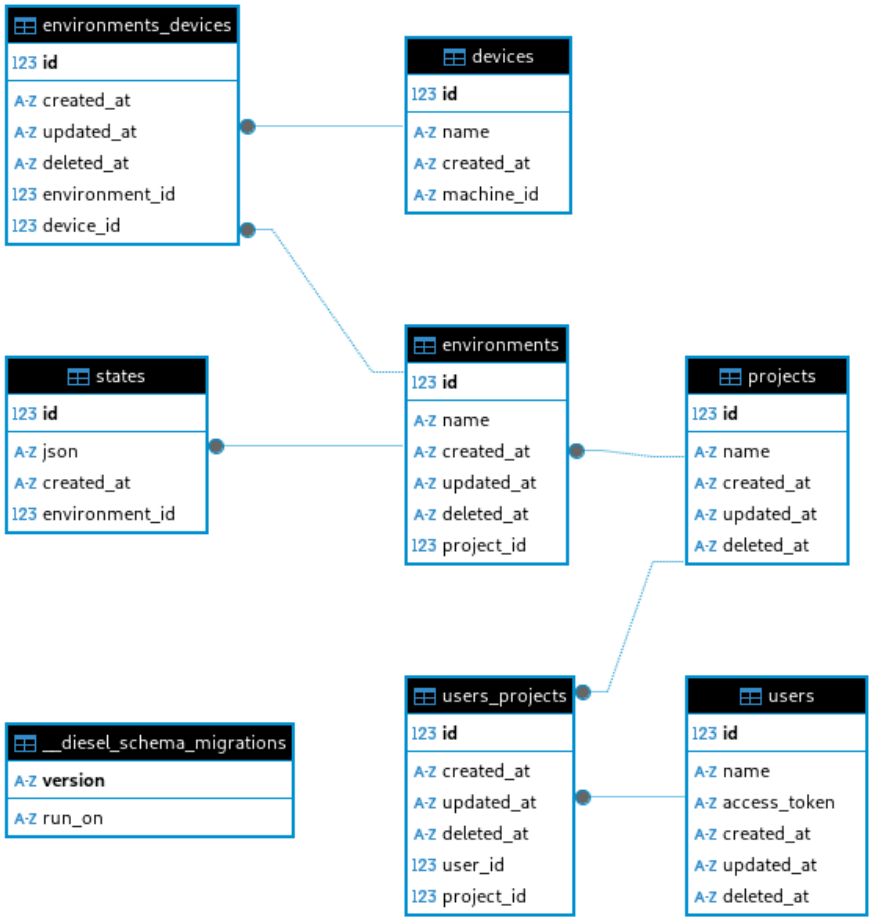

Figura 7.1: Diagrama Entidad-Relación del esquema de la base de datos

Finalmente, el esquema de la base de datos mostrado en el diagrama Entidad-Relación de la figura 7.1 modela los siguientes requerimientos funcionales:

- Un usuario puede tener acceso a múltiples proyectos y viceversa. El control de acceso se determina si existe una relación entre ambos.
- Un proyecto puede tener múltiples entornos y cada entorno sólo puede tener un estado objetivo. Esto facilita el hecho de que puedan existir entornos de pruebas y de producción con el mismo código en un proyecto.
- Cada entorno puede estar en múltiples dispositivos y cada dispositivo puede tener múltiples entornos.

7.4. Actualizaciones de estado

Teniendo en cuenta la sección anterior, un dispositivo puede tener varios entornos y cada entorno puede tener un solo estado. Esto significa que un dispositivo puede tener múltiples cambios de estado a la vez. Adicionalmente, como una actualización de estado implica un cambio en el archivo de estado, se tomó la decisión de que cada vez que el Coordinador de Agentes le haga una petición de estado al dispositivo se calcule un hash MD5 del archivo de estado actual almacenado en el dispositivo y en el estado almacenado en la base de datos. Si existe una colisión entre ambos, significa que es necesario realizar una actualización de estado mediante un mensaje `UpdateEnvironmentsRequest`.

Para un trabajo futuro se recomienda agregar una tabla adicional que contenga el historial de cada ejecución de actualización de estado con el objetivo de auditar los cambios de infraestructura en el dispositivo.

8. Resultados

8.1. Equipos y herramientas usadas

Para probar la herramienta en su implementación de referencia, se replicó un flujo de trabajo acorde a un caso de uso de la sección 2.2. Para ello, se hizo uso de los siguientes equipos:

- 2 Raspberry Pi 4 Model B 8 GB con sistema operativo Raspberry Pi OS x64.
- 1 máquina virtual del proveedor Github Actions con 4 vCPU, 16 GB de memoria RAM y sistema operativo Ubuntu 24.04 x64.
- 1 máquina virtual del proveedor DigitalOcean con 1 vCPU, 1 GB de memoria RAM y sistema operativo Debian 12 x64.
- 1 computador personal con CPU Intel i7-11370H @ 3.30GHz, 32 GB de memoria RAM y sistema operativo Fedora 40 x64.

En donde los dispositivos Raspberry Pi ejecutan el agente, la máquina virtual de DigitalOcean ejecuta el Coordinador de Agentes, el computador personal fue usado como cliente de un usuario de CLI y la máquina virtual de Github Actions fue usada como un cliente de usuario de CLI automatizado.

Adicionalmente, se utilizó la herramienta de IaC OpenTofu para levantar la infraestructura del Coordinador de Agentes.

8.2. Implementación de referencia

La implementación de referencia, de acuerdo a lo expuesto en este trabajo, está disponible en el repositorio de git https://github.com/duskje/ovejas_project con commit `be8d3c1`.

Las instrucciones para ejecutar el proyecto se encuentran en el archivo `README.md` del mismo repositorio.

8.3. Uso de la herramienta

Para este proyecto de ejemplo se tomará como base la estructura mostrada en la figura 8.1.

```
[david@fedora-dell] $ tree
.
├── main.py
├── poetry.lock
├── pyproject.toml
└── README.md

1 directory, 4 files
```

Figura 8.1: Ejemplo de la estructura de un proyecto

Mediante Poetry, se establece el nombre del proyecto en el archivo `pyproject.toml` y se cargan las dependencias necesarias. En particular, la implementación de referencia incluye un proyecto de Python llamado `python_sdk` el cual debe ser incluido como dependencia en el proyecto para poder obtener el estado objetivo.

8.3.1. Registro de dispositivos

Para que un dispositivo pueda interactuar con el Coordinador de Agentes, este primero debe ser registrado mediante el comando `ovejas device write`. Este comando debe incluir las opciones `--name` y `--machine-id`, en donde el primero es sólo para uso del administrador y el último representa un identificador único.

Por otra parte, para que un estado objetivo sea asignado a un dispositivo, este tiene que ser asignado previamente mediante el comando `ovejas environment` que debe llevar la opción `--env` para asignarle el entorno del cual se debe recibir el estado objetivo y la opción `--machine-id` para establecer a qué dispositivos se les enviará el estado objetivo. Este proceso puede observarse en las figuras 8.2 y 8.3.

```
[david@fedora-dell] $ ovejas device write --machine-id ovejas-0 --name ovejas-0
  2025-04-15T03:38:02.465218Z  INFO ovejas: response: "Ok(\n    ServerResponse {
\n        msg: \"Created device successfully\",\n        data: Null,\n    },\n)"

[david@fedora-dell] $ ovejas environment --env test-env add-device --machine-id
ovejas-0
  2025-04-15T03:38:09.018741Z  INFO ovejas: response: "Ok(\n    ServerResponse {
\n        msg: \"Device enrolled successfully\",\n        data: Null,\n    },\n)
"
```

Figura 8.2: Registro de dispositivos y asignación de entorno

```
  2025-04-15T03:37:56.160009Z  INFO server: Listening at 0.0.0.0:9734

  2025-04-15T03:38:02.343304Z  INFO server: New incoming request

  2025-04-15T03:38:02.343815Z  INFO server: path: "/device", headers: "{\n    \"
content-type\": \"application/json\",\n    \"machine-type\": \"cli\",\n    \"aut
horization\": \"hola\",\n    \"content-length\": \"43\",\n    \"accept\": \"*/*\
",\n    \"host\": \"64.23.252.58:9734\",\n}"

  2025-04-15T03:38:02.344149Z  INFO server: protocol: "HTTP"

device create result Ok(())
```

Figura 8.3: Creación de dispositivo desde el Coordinador de Agentes

8.3.2. Creación de recursos

El comando `ovejas up` se utiliza para crear un nuevo estado objetivo. Al ser ejecutado en la base del proyecto o en un subdirectorio de este, se ejecuta el archivo *main.py* del cual se obtiene el estado objetivo a partir de la creación de un nuevo objeto. El ejemplo de esta sección viene en la figura 15, en donde se instanciará la clase `User` con el objetivo de crear el usuario `my_user` con uid `3999` en el dispositivo.

Adicionalmente, este comando crea tanto el proyecto como el entorno si estos no existen en la base de datos.

```python
from ovejas.system import User

user = User(f"example_user", name=f"my_user", uid=3999)
```

Sección de código 15: Archivo ejecutado para crear un usuario

```
[david@fedora-dell] $ nvim main.py
[david@fedora-dell] $ ovejas up --env test-local
  2025-04-15T02:54:47.357927Z  INFO ovejas: Connected successfully to remote

  2025-04-15T02:54:47.389099Z  INFO ovejas: Target state pushed to remote.
```

Figura 8.4: Logs de actualización de estado en la herramienta de CLI

Por otra parte, para establecer a qué entorno se subirá el estado objetivo se estableció por medio de la opción `--env`. En la figura 8.4 se puede ver el envío del estado objetivo desde la herramienta de CLI ejecutando la sección de código 15 y su posterior recepción en la figura 8.5 desde el Coordinador de Agentes.

```
  2025-04-15T02:54:47.357466Z  INFO server: New incoming request

  2025-04-15T02:54:47.357550Z  INFO server: path: "/socket", headers: "{\n    \"
host\": \"example.com\",\n    \"connection\": \"upgrade\",\n    \"upgrade\": \"w
ebsocket\",\n    \"sec-websocket-version\": \"13\",\n    \"sec-websocket-key\":
\"foo\",\n    \"authorization\": \"hola\",\n    \"machine-type\": \"cli\",\n
\"machine-id\": \"cli\",\n}"

  2025-04-15T02:54:47.357686Z  INFO server: protocol: "WebSocket"

Command: Up
loaded project: python-example-project
loaded environment: test-local
  2025-04-15T02:54:47.389784Z  INFO server: opeartion: "up", environment: "test-
local", project: "python-example-project", state: "{\n  \"version\": 1,\n  \"cre
ated_at\": \"2025-04-14 22:54:47.389025\",\n  \"resources\": [\n    {\n      \"u
rn\": \"ovejas.system::User::example_user\",\n      \"parameters\": {\n        \
"gid\": 0,\n        \"name\": \"my_user\",\n        \"uid\": 3999\n      },\n
   \"depends_on\": []\n    }\n  ]\n}"
```

Figura 8.5: Logs de actualización de estado en el Coordinador de Agentes

Finalmente, en la figura 8.6 se puede ver reflejada la creación del recurso desde el agente y en la figura 8.7 se puede ver el recurso creado.

```
2025-04-15T03:51:00.403464Z  INFO device: Remote requested current state
2025-04-15T03:51:00.418461Z  INFO device: local_state_hashes="{\"test-env\": [54
, 9, 156, 242, 2, 150, 107, 76, 192, 244, 89, 150, 196, 182, 169, 44]}"
2025-04-15T03:51:00.594432Z  INFO device: Remote requested to update current sta
te
{"ovejas.system::User::example_user"}
```

Figura 8.6: Logs de actualización de estado en el Agente

```
ovejas@ovejas-0:~/ovejas_project/device $ cat /etc/passwd | grep my_user
my_user:x:3999:0::/home/my_user:/bin/bash _
```

Figura 8.7: Recursos creados por medio del Agente

8.3.3. Actualización de recursos

De forma similar al caso de la creación de recursos, el comando `ovejas up` también permite actualizar el estado objetivo.

```python
from ovejas.system import User

for i in range(10):
    user = User(f"user_{i}", name=f"user_{i}", uid=4000 + i)
```

Sección de código 16: Archivo ejecutado para crear diez usuarios

Si la sección de código 16 se ejecuta después de la sección de código 15 se crearán diez usuarios y se eliminará el usuario `my_user` en el dispositivo. Esto último se puede observar en las figuras 8.8 y 8.9.

Es importante notar que el primer argumento del constructor de cada recurso debe ser único entre cada uno, ya que ese nombre determina el URN del recurso. Por otra parte, el hecho de que la variable `user` sea reasignada en cada iteración no afecta el funcionamiento, porque la etapa de registro del recurso se hace al momento de llamar al constructor.

```
2025-04-15T04:02:59.622796Z  INFO device: Remote requested current state
2025-04-15T04:02:59.634938Z  INFO device: local_state_hashes="{\"test-env\": [15
0, 167, 168, 224, 107, 77, 82, 154, 147, 195, 81, 130, 53, 1, 42, 164]}"
2025-04-15T04:02:59.810996Z  INFO device: Remote requested to update current sta
te
{"ovejas.system::User::user_8", "ovejas.system::User::user_2", "ovejas.system::U
ser::user_0", "ovejas.system::User::user_1", "ovejas.system::User::user_3", "ove
jas.system::User::user_6", "ovejas.system::User::user_4", "ovejas.system::User::
user_7", "ovejas.system::User::user_9", "ovejas.system::User::user_5"}
{"ovejas.system::User::example_user"}
```

Figura 8.8: Logs de actualización de estado en el Agente

```
ovejas@ovejas-0:~/ovejas_project/device $ cat /etc/passwd | grep user
hplip:x:111:7:HPLIP system user,,,:/run/hplip:/bin/false
user_8:x:4008:0::/home/user_8:/bin/bash
user_2:x:4002:0::/home/user_2:/bin/bash
user_0:x:4000:0::/home/user_0:/bin/bash
user_1:x:4001:0::/home/user_1:/bin/bash
user_3:x:4003:0::/home/user_3:/bin/bash
user_6:x:4006:0::/home/user_6:/bin/bash
user_4:x:4004:0::/home/user_4:/bin/bash
user_7:x:4007:0::/home/user_7:/bin/bash
user_9:x:4009:0::/home/user_9:/bin/bash
user_5:x:4005:0::/home/user_5:/bin/bash _
```

Figura 8.9: Recursos actualizados en el dispositivo

8.3.4. Eliminación de recursos

Finalmente, en el caso de que se desee eliminar todos los recursos creados en el dispositivo con la herramienta, se puede utilizar el comando `ovejas down` como se muestra en las figuras 8.10 y 8.11. Exceptuando el nombre del proyecto, este comando no toma en cuenta el contenido de este.

```
[david@fedora-dell] $ ovejas down --env test-env
  2025-04-15T04:08:18.897006Z  INFO ovejas: Connected successfully to remote
```

Figura 8.10: Recursos eliminados en el dispositivo

```
2025-04-15T04:08:45.342113Z  INFO device: Remote requested current state
2025-04-15T04:08:45.354260Z  INFO device: local_state_hashes="{\"test-env\": [20
4, 231, 73, 50, 45, 208, 48, 146, 28, 141, 44, 25, 119, 107, 148, 236]}"
2025-04-15T04:08:45.526970Z  INFO device: Remote requested to update current sta
te
^C
ovejas@ovejas-0:~/ovejas_project/device $ cat /etc/passwd | grep user
hplip:x:111:7:HPLIP system user,,,:/run/hplip:/bin/false
```

Figura 8.11: Recursos eliminados en el dispositivo

8.3.5. Ejecución en flujo de CI/CD

Para este ejemplo, se considerará el caso de uso mencionado en las secciones anteriores, en donde se mencionó la ejecución de la herramienta CLI desde un pipeline de CI/CD.

En este caso, asumiremos que el proyecto está bajo versionamiento por la herramienta git y que su administración es por medio de la plataforma GitHub. Bajo este ejemplo se considera el caso de que una organización decida deshabilitar el *push* a una rama directamente mediante la interfaz por línea de comandos de git y que, en cambio, esta acción deba hacerse mediante una plataforma como GitHub que administre el repositorio.

Considerando esto, un pipeline de CI/CD permitiría ejecutar comandos de forma automatizada ante eventos como un *push* de código a la rama. Ya que la infraestructura está administrada con código, cualquier cambio en la infraestructura también se verá reflejado por el control de versiones y podrá ser revisado por los integrantes de la organización con acceso al repositorio. Este flujo de trabajo es usualmente llamado GitOps [41].

Para replicar esto, se creó una rama de git en donde se introduce un cambio en el archivo del proyecto mostrado en la figura 8.12.

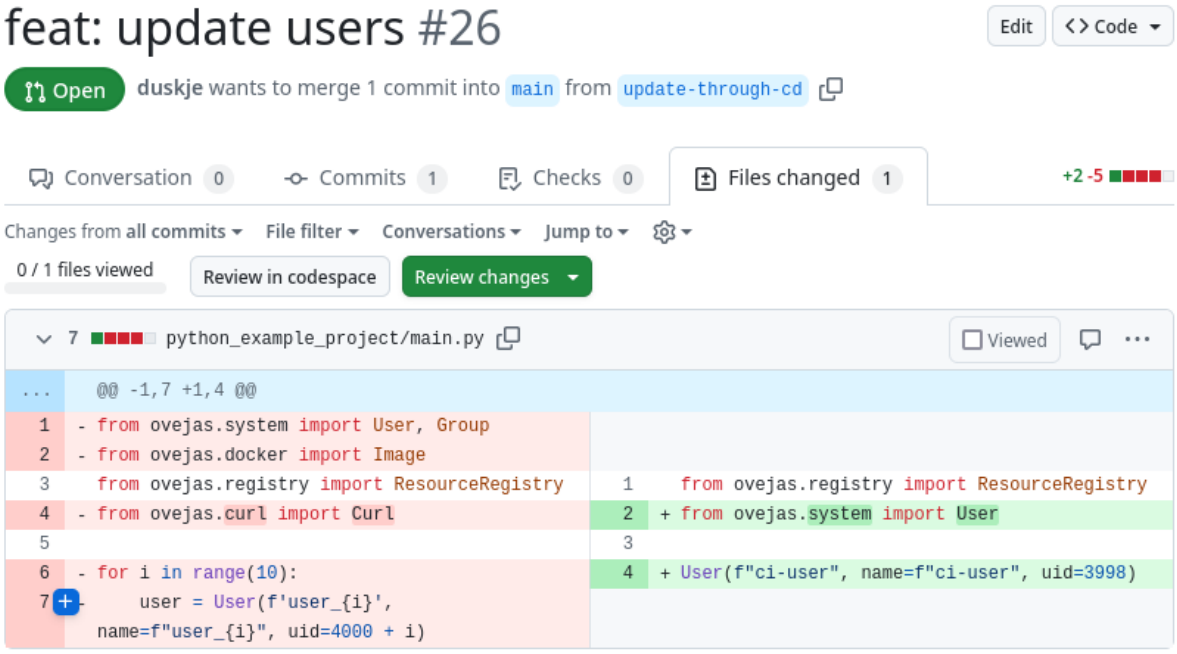

Figura 8.12: Diferencia introducida en el código

Luego, este cambio puede ser introducido por el *Pull Request* mostrado en la figura 8.13

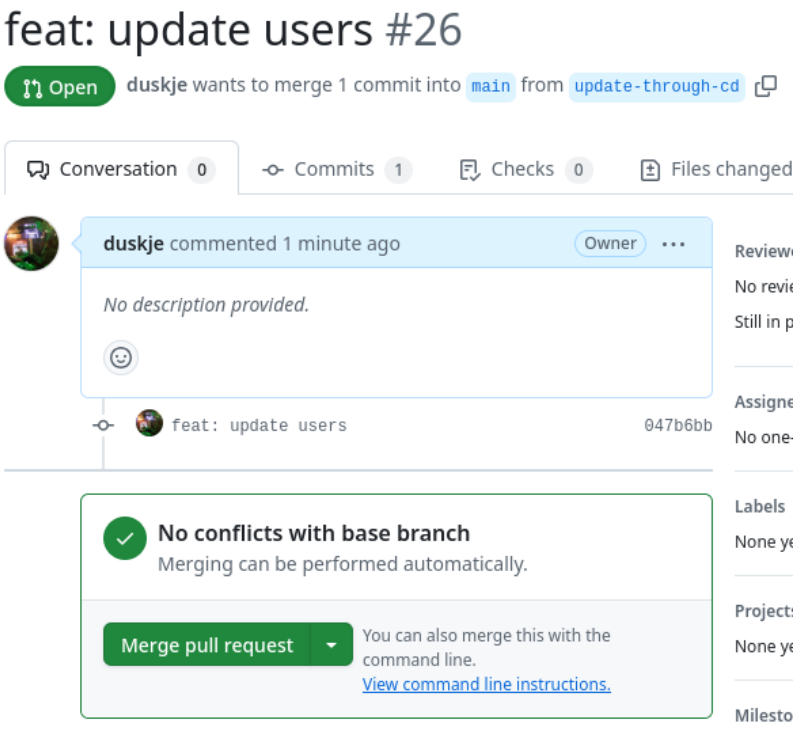

Figura 8.13: Pull request para introducir los cambios a la rama main

Después de hacer el merge, el cambio lanza un evento para cargar un ≪workflow≫; un archivo de

formato YAML procesado por GitHub para ejecutar un algoritmo definido por el usuario. El workflow usado para este proceso está tanto en el repositorio como en el anexo. Este paso puede observarse en la figura 8.14.

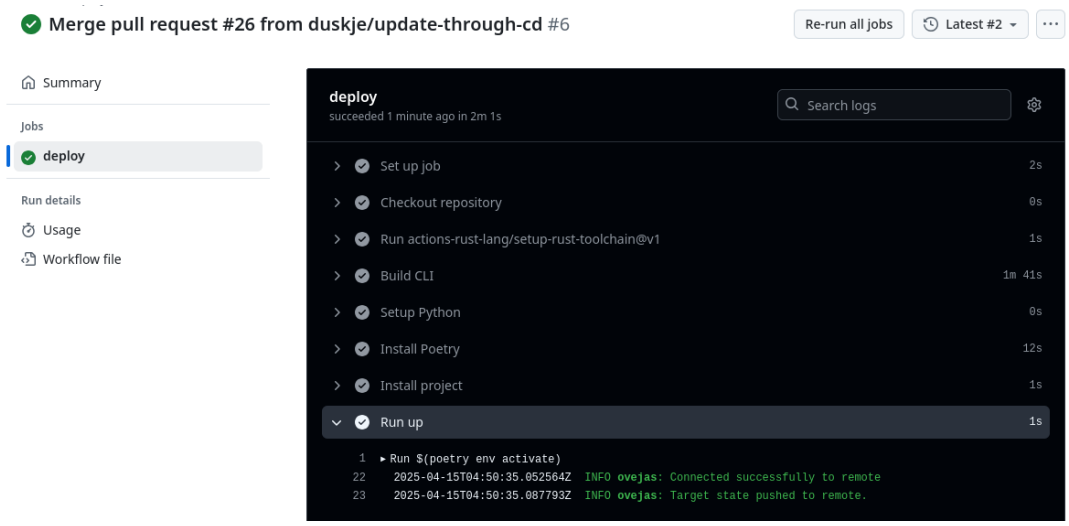

Figura 8.14: Ejecución de workflow

Finalmente, después de esta ejecución, la actualización exitosa de los recursos se puede observar en la figura 8.15.

```
2025-04-15T04:51:42.510394Z  INFO device: Remote requested current state
2025-04-15T04:51:42.522667Z  INFO device: local_state_hashes="{\"test-env\": [16
6, 158, 180, 11, 255, 86, 1, 0, 161, 196, 231, 4, 60, 189, 222, 42]}"
2025-04-15T04:51:42.695603Z  INFO device: Remote requested to update current sta
te
^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA
^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA
^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^[OA^C
ovejas@ovejas-0:~/ovejas_project/device $ cat /etc/passwd | grep user
hplip:x:111:7:HPLIP system user,,,:/run/hplip:/bin/false
ci-user:x:3998:0::/home/ci-user:/bin/bash
```

Figura 8.15: Actualización de estado por medio de un flujo de CI/CD

9. Conclusiones

En el desarrollo de este trabajo, se abordó la administración remota de sistemas embebidos a través del estudio de herramientas existentes de Infraestructura como Código (IaC). Por una parte, se estableció la importancia de estas herramientas en el ciclo de vida de las aplicaciones y su infraestructura. Por otra, se investigaron sus características y su funcionamiento interno con el objetivo de establecer puntos en común con las necesidades de la gestión de dispositivos IoT y así poder adaptarse a sus necesidades. Esto permitió considerar las ventajas y limitaciones de cada herramienta y escoger los elementos más adecuados para el proyecto.

Durante el proceso de diseño, se consideraron las particularidades de los dispositivos IoT conectados por redes inestables, así como consideraciones prácticas para realizar bibliotecas y proveedores en un ecosistema de código abierto. Adicionalmente, se tomaron en cuenta consideraciones prácticas para facilitar su uso en un contexto de cultura DevOps, en donde el sistema se integraría en procesos de CI/CD.

Entre los principales desafíos que pueden ser abordados en un trabajo futuro, están realizar mejoras en la seguridad de la herramienta y en la experiencia de uso. Por el lado de mejoras de seguridad, se pueden considerar medidas como la autenticación de los usuarios mediante proveedores de identidad, el registro de los dispositivos de forma segura y la gestión de secretos. Por el lado de experiencia de uso, un trabajo futuro puede abordar que las actualizaciones de estado sean transaccionales y que las actualizaciones de estado en el dispositivo se ejecuten de manera concurrente.

Finalmente, en este trabajo se demuestra el funcionamiento normal de la herramienta, incluyendo la administración de dispositivos y recursos, así como también la incorporación de la herramienta en un flujo de CI/CD por medio de GitHub Actions, así como también con herramientas como OpenTofu e infraestructura de la nube como Digital Ocean. Estos resultados validaron las decisiones tomadas durante el desarrollo de este documento y demuestran la viabilidad de la solución propuesta para gestionar la infraestructura de dispositivos IoT mediante IaC.

Referencias

[1] "What is a CI/CD pipeline?" [Online]. Available: https://www.redhat.com/en/topics/devops/what-cicd-pipeline

[2] H. Yasar and S. E. Teplov, "DevSecOps In Embedded Systems: An Empirical Study Of Past Literature," *ARES 22: Proceedings of the 17th International Conference on Availability, Reliability and Security*, 2022.

[3] DORA, "The State of DevOps report: Are you a software leader?" Tech. Rep., 2022.

[4] Amazon Web Services, "What is infrastructure as code?" [Online]. Available: https://aws.amazon.com/what-is/iac/

[5] Red Hat, "What is YAML?" 2023. [Online]. Available: https://www.redhat.com/en/topics/automation/what-is-yaml

[6] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, Tech. Rep., 2011.

[7] Michael Goodwin, "¿Qué es el aprovisionamiento?" [Online]. Available: https://www.ibm.com/es-es/think/topics/provisioning

[8] Pulumi, "What is Pulumi?" 2024. [Online]. Available: https://www.pulumi.com/docs/iac/concepts/#what-is-pulumi

[9] Terraform, "Configuration Syntax," 2024. [Online]. Available: https://developer.hashicorp.com/terraform/language/syntax/configuration

[10] A. Dorri, S. S. Kanhere, and R. Jurdak, "Multi-Agent Systems: A Survey," *IEEE Access*, 2018.

[11] NinjaOne Inc., "Everything you need to know about RMM agents: A complete overview." [Online]. Available: https://www.ninjaone.com/blog/everything-you-need-to-know-about-rmm-agents-a-complete-overview/

[12] Puppet Inc., "Install agents." [Online]. Available: https://www.puppet.com/docs/puppet/8/install_agents#install_agents

[13] Salt Project, "Salt minion." [Online]. Available: https://docs.saltproject.io/en/latest/ref/cli/salt-minion.html

[14] Red Hat, "Red Hat Ansible Automation Platform: A beginner's guide," Red Hat, Tech. Rep., 2022.

[15] Pulumi Security Team, "Pulumi Cloud Security Whitepaper," Pulumi, Tech. Rep., 2022.

[16] AWS Prescriptive Guidance, "Getting started with Terraform: Guidance for AWS CDK and AWS CloudFormation experts," Amazon Inc., Tech. Rep., 2024.

[17] Hashicorp, "Detecting and Managing Drift with Terraform." [Online]. Available: https://www.hashicorp.com/blog/detecting-and-managing-drift-with-terraform

[18] Terraform, "Publish providers," 2025. [Online]. Available: https://developer.hashicorp.com/terraform/registry/providers/publishing

[19] Amazon Web Services, "Working with cloudformation templates - aws cloudformation." [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-guide.html

[20] Pulumi, "Provisioner: local-exec — terraform — hashicorp developer." [Online]. Available: https://developer.hashicorp.com/terraform/language/resources/provisioners/local-exec

[21] ——, "Command." [Online]. Available: https://www.pulumi.com/registry/packages/command/api-docs/local/command/

[22] "Developing modules — ansible community documentation." [Online]. Available: https://docs.ansible.com/ansible/latest/dev_guide/developing_modules_general.html

[23] S. Project, "Install overview - salt install guide." [Online]. Available: https://docs.saltproject.io/salt/install-guide/en/latest/topics/overview.html#alternative-installations-and-configurations

[24] Amazon Web Services, "Managing AWS resources as a single unit with AWS CloudFormation stacks," 2025. [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/stacks.html

[25] A. Wiggins, "The twelve-factor app. 2017," *Saatavissa (viitattu 26.3. 2016) http://12factor. net*, 2018.

[26] Ansible Community Documentation, "Developing plugins." [Online]. Available: https://docs.ansible.com/ansible/latest/dev_guide/developing_plugins.html#developing-plugins

[27] Salt Project, "Execution architecture." [Online]. Available: https://docs.saltproject.io/salt/user-guide/en/latest/topics/execution-architecture.html

[28] ——, "Execution architecture." [Online]. Available: https://docs.saltproject.io/salt/user-guide/en/latest/topics/execution-architecture.html

[29] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[30] Erik De Bonte, "PEP 681 – Data Class Transforms," Python Enhancement Proposals, Tech. Rep., 2021.

[31] Pulumi, "Inputs & outputs," 2024. [Online]. Available: https://www.pulumi.com/docs/iac/concepts/inputs-outputs/#outputs

[32] Docker, "Control startup and shutdown order in Compose," 2024. [Online]. Available: https://docs.docker.com/compose/how-tos/startup-order/

[33] Pulumi, "Resource option: dependsOn," 2024. [Online]. Available: https://www.pulumi.com/docs/iac/concepts/options/dependson/#resource-option-dependson

[34] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, p. 558–562, Nov. 1962. [Online]. Available: https://doi.org/10.1145/368996.369025

[35] T. Preston-Werner, "Semantic versioning 2.0. 0," *línea]. Available: http://semver. org*, 2013.

[36] Pulumi, "Pulumi packages," 2025. [Online]. Available: https://www.pulumi.com/docs/iac/using-pulumi/pulumi-packages/

[37] Terraform, "Provider code walkthrough," 2025. [Online]. Available: https://developer.hashicorp.com/terraform/plugin/framework/getting-started/code-walkthrough

[38] I. Fette, A. Melnikov, "The WebSocket Protocol," RFC 6455, 2011.

[39] SQLite, "SQLite File IO Specification," 2024. [Online]. Available: https://www.sqlite.org/draft/fileio.html

[40] ——, "Appropriate Uses For SQLite," 2025. [Online]. Available: https://www.sqlite.org/whentouse.html

[41] GitLab, "What is a gitops workflow?" 2025. [Online]. Available: https://about.gitlab.com/topics/gitops/gitops-workflow/

10. Anexo: Workflow para proceso de despliegue continuo

```yaml
name: Run deploy
on:
  push:
    branches: main
    paths: 'python_example_project/**'
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4
      - uses: actions-rust-lang/setup-rust-toolchain@v1
        with:
          rustflags: ""
      - name: Build CLI
        working-directory: ./cli
        run: cargo install --path .
      - name: Setup Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.12'
      - name: Install Poetry
        uses: snok/install-poetry@v1
      - name: Install project
        working-directory: ./python_example_project
        run: poetry install --no-root
      - name: Run up
        working-directory: ./python_example_project
        run: |
          $(poetry env activate)
          ovejas up --env test-env
        env:
          ADDRESS: ${{ secrets.ADDRESS }}
          CLI_TOKEN: ${{ secrets.CLI_TOKEN }}
```

UNIVERSIDAD DE CONCEPCION – FACULTAD DE INGENIERIA
RESUMEN DE MEMORIA DE TITULO

Departamento	**:** Departamento de Ingeniería Eléctrica
Carrera	: Ingeniería Civil Electrónica
Nombre del memorista	: David Andrés Oscanoa Lara
Título de la memoria	**:** Infraestructura como código para IoT
Fecha de la presentación oral	: 03/09/2025
Profesor(es) guía	: Mario Medina Carrasco
Profesor(es) revisor(es)	: Sergio Sobarzo, Geoffrey Hecht
Concepto	**:**
Calificación	**:**

Resumen (máximo 200 palabras)

El crecimiento del Internet de las Cosas (IoT) ha impulsado el uso de sistemas embebidos basados en Linux en diversas aplicaciones. Sin embargo, su gestión a gran escala presenta desafíos debido a restricciones como conectividad limitada y escasos recursos. En este contexto, se propone una herramienta de Infraestructura como Código (IaC) diseñada específicamente para sistemas embebidos en dispositivos IoT, superando las limitaciones de herramientas actuales que no están adaptadas a estos entornos.

A diferencia de soluciones centradas en sistemas inmutables con actualizaciones A/B —más costosas y complejas—, esta propuesta opera sobre sistemas vivos, aplicando cambios en tiempo de ejecución de manera segura. Esto permite incorporar prácticas DevOps en el desarrollo de dispositivos IoT, mejorando la agilidad y acelerando los ciclos de entrega.

En este trabajo se realiza el diseño de una herramienta de IaC con tecnologías modernas como Rust, Diesel, Tokio y WebSockets, y se presenta una implementación de referencia que valida el diseño. Además, se demuestra su viabilidad en un flujo de trabajo CI/CD, integrándose en organizaciones que gestionan infraestructura IoT.