POSGRADOS

Maestría en

Software

RPC-SO-34-NO.778-2021

Opción de Titulación:

Proyecto de titulación con componentes de investigación aplicada y/o de desarrollo

Tema:

Implementación de un prototipo de herramienta en Go para la automatización de tareas y despliegue de aplicaciones en nodos controlados bajo una arquitectura cliente-servidor sin dependencias de intérpretes externos

Autor:

Pablo Gustavo Fuentes Espinoza

Director:

Jorge Oswaldo Loja Cajas

GUAYAQUIL – Ecuador

2025

Autor:

Pablo Gustavo Fuentes Espinoza
Ingeniero Electrónico en Control y Automatismo.
Candidato a Magíster en Software, Mención en Diseño de Arquitectura de Sistemas por la Universidad Politécnica Salesiana - Sede Guayaquil.
pfuentese@est.ups.edu.ec

Dirigido por:

Jorge Oswaldo Loja Cajas
Ingeniero de Sistemas.
Máster Universitario en Análisis y Visualización de Datos Masivos / Visual Analytics and Big Data.
jloja@ups.edu.ec

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra para fines comerciales, sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual. Se permite la libre difusión de este texto con fines académicos investigativos por cualquier medio, con la debida notificación a los autores.

DERECHOS RESERVADOS
©2025 Universidad Politécnica Salesiana.
GUAYAQUIL – ECUADOR – SUDAMÉRICA
PABLO GUSTAVO FUENTES ESPINOZA.

IMPLEMENTACIÓN DE UN PROTOTIPO DE HERRAMIENTA EN GO PARA LA AUTOMATIZACIÓN DE TAREAS Y DESPLIEGUE DE APLICACIONES EN NODOS CONTROLADOS BAJO UNA ARQUITECTURA CLIENTE-SERVIDOR SIN DEPENDENCIAS DE INTÉRPRETES EXTERNOS

Dedicatoria

A Dios, cuya guía, sabiduría y gracia han iluminado cada paso de este camino, pues sin su ayuda y su propósito, nada de esto habría sido posible.

A mis padres, Darwin y Karina, por su amor incondicional, su paciencia y sacrificio que han forjado la persona que soy hoy.

A mi hermana, Joselyn, cuyo afecto y aliento siempre me han recordado la importancia de la perseverancia.

A mi novia, Karelys, cuyo amor, entendimiento y fe en mi han sido una fuente constante de luz e inspiración a lo largo de este camino.

Y a todos quienes que, con sus palabras, acciones y ejemplo, me animaron a seguir aprendiendo, creciendo y avanzando.

Agradecimientos

En primer lugar, doy gracias a Dios por su guía, sabiduría y constante presencia a lo largo de este viaje. Su fuerza y gracia me han dado la perseverancia y claridad para completar este trabajo.

Me gustaría también expresar mi gratitud a mi tutor de tesis por su tiempo y su valiosa guía durante el desarrollo de este trabajo. Su experiencia y observaciones contribuyeron a mejorar la calidad e implementación del proyecto.

Mi más sincero agradecimiento al profesorado y al personal del programa de posgrado por su dedicación y por crear un entorno de aprendizaje virtual que fomentó el crecimiento académico y la innovación.

También quiero expresar mi agradecimiento a mis compañeros de clase por compartir este trayecto y por contribuir a una experiencia de aprendizaje colectivo.

Finalmente, estoy profundamente agradecido con mi familia y Karelys por su amor, paciencia y fe en mi. Su apoyo ha sido la base de mi motivación y la mayor recompensa de este recorrido.

Índice general

Índice de Figuras VIII

Índice de Tablas IX

Resumen XI

Abstract XII

1. Introducción 1

1.1. Descripción general del problema . . . 2
1.2. Objetivos . . . 2
1.2.1. Objetivo general . . . 2
1.2.2. Objetivos específicos . . . 2
1.3. Contribuciones . . . 3
1.4. Organización del manuscrito . . . 4

2. Marco Teórico Referencial 5

2.1. Estado del Arte . . . 6
2.2. Definiciones Previas . . . 7
2.2.1. Gestión de Configuración - CM . . . 7
2.2.2. Automatización de tareas . . . 8
2.2.3. Arquitectura Cliente-Servidor . . . 9
2.2.4. Modelos de configuración - Declarativos e Imperativos 10
2.2.5. Gestión de configuración basada en agentes y sin agentes 10
2.2.6. Intérprete . . . 11
2.2.7. Lenguaje Específico de Dominio - DSL . . . 12
2.3. Herramientas de CM existentes . . . 12
2.3.1. Ansible . . . 13
2.3.2. Puppet . . . 19
2.3.3. Chef . . . 25

2.3.4. Análisis comparativo de las herramientas 32
2.4. Tecnologías relevantes 33
2.4.1. Lenguaje de Programación Go 33
2.4.2. Formato YAML 34
2.4.3. Secure Shell - SSH 35

3. Prototipo: Diseño e implementación 38
3.1. Requisitos y criterios de diseño 39
3.1.1. Requisitos funcionales 39
3.1.2. Requisitos no funcionales 39
3.1.3. Criterios de diseño 40
3.2. Arquitectura del sistema 42
3.2.1. Vista General del Componente 42
3.2.2. Flujo de Ejecución Central 43
3.2.3. Dinámica de la Ejecución 46
3.2.4. Concurrencia y Separación de Responsabilidades . . . 46
3.2.5. Extensibilidad y Mantenibilidad 47
3.2.6. Consideraciones de Seguridad 48
3.3. Archivos de Configuración y DSL 49
3.3.1. Archivo Swimbook 49
3.3.2. Archivo ControllerPlan 50
3.3.3. Archivo Hosts 54
3.3.4. Sustitución de variables 54
3.4. Sistema de Módulos y Extensibilidad 59
3.4.1. API Pública de Módulo 59
3.4.2. Manifiesto del Módulo 62
3.4.3. Módulos integrados, oficiales y de terceros 62
3.4.4. Empaquetado e Instalación 64
3.4.5. Flujo de ejecución y resolución 66
3.4.6. Resumen 67
3.5. Diseño de la Interfaz de Línea de Comandos (CLI) 67
3.5.1. Jerarquía y estructura de comandos 68
3.5.2. shoal run 68
3.5.3. shoal ping 69
3.5.4. shoal mod 70
3.5.5. shoal init 70
3.5.6. shoal version 71
3.5.7. Flujo de trabajo 71
3.6. Validación y Resultados 72
3.6.1. Entorno de pruebas 72

3.6.2. Demostración Funcional 74
3.6.3. Salida de la CLI y Capturas de Pantalla 80
3.6.4. Resumen del cumplimiento de requisitos 83

4. Discusión y Conclusiones 87
4.1. Discusión de Resultados 88
4.2. Comparación con Herramientas Existentes 89
4.3. Limitaciones y Trabajo Futuro 91
4.4. Conclusiones 92

Índice de Figuras

2.1. Arquitectura Cliente-Servidor 9
2.2. Ansible - Arquitectura 14
2.3. Puppet - Arquitectura 20
2.4. Chef - Arquitectura 27

3.1. Trazabilidad (Objetivos - Req. Funcionales - Criterios de diseño) 41
3.2. Trazabilidad (Objetivos - Req. No Funcionales - Criterios de diseño) 42
3.3. Shoal - Arquitectura (Componentes) 43
3.4. Diagrama de secuencia (`shoal run`) 47
3.5. Shoal - Diagrama de clases 48
3.6. Shoal - Prueba RF1 81
3.7. Shoal - Prueba RF2 81
3.8. Archivos de salida - Prueba RF2 82
3.9. Shoal - Prueba RF3 - Swimbook inválido 82
3.10. Shoal - Prueba RF3 - Swimbook válido 82
3.11. Shoal - Prueba RF4 83

Índice de Tablas

2.1. Comparación entre herramientas CM 37

3.1. Componentes de Shoal 44
3.2. Campos que admiten interpolación 57
3.3. Variables - Sintaxis de Acceso 57
3.4. Campos en el Manifiesto de Módulo 63
3.5. Módulos integrados de `Shoal` 63
3.6. Comandos de la CLI de `Shoal` 68
3.7. Subcomandos de `shoal mod` 70
3.8. Entorno de pruebas 73
3.9. Validación de Requisitos Funcionales 84
3.10. Validación de Requisitos No Funcionales 85

4.1. Shoal - Comparación con herramientas existentes 93

IMPLEMENTACIÓN DE UN PROTOTIPO DE HERRAMIENTA EN GO PARA LA AUTOMATIZACIÓN DE TAREAS Y DESPLIEGUE DE APLICACIONES EN NODOS CONTROLADOS BAJO UNA ARQUITECTURA CLIENTE-SERVIDOR SIN DEPENDENCIAS DE INTÉRPRETES EXTERNOS

Autor:

PABLO GUSTAVO FUENTES ESPINOZA

Resumen

La creciente complejidad y heterogeneidad de las infraestructuras de TI han hecho de la automatización un aspecto esencial en las administración de sistemas modernos. Herramientas de gestión de configuración como Ansible, Puppet y Chef han simplificado enormemente los procesos de configuración y despliegue en sistemas distribuidos. Sin embargo, muchas de estas herramientas dependen de entornos de ejecución externos o agentes, limitando su aplicabilidad en entornos mínimos o controlados. Este trabajo aborda esta limitación a través del diseño e implementación de `Shoal`, un prototipo de herramienta para la automatización de tareas y despliegue de aplicaciones que opera sin depender de intérpretes externos.

Desarrollado en el lenguaje de programación Go, `Shoal` adopta una arquitectura cliente-servidor donde un nodo controlador se comunica con los nodos administrados de forma segura a través de SSH. La herramienta ejecuta tareas de automatización definidas en archivos de configuración en YAML usando una sintaxis declarativa diseñada para la claridad y facilidad de uso. Su motor interno soporta ejecución de tareas concurrentes en sistemas basados en Debian Linux y Windows, consiguiendo portabilidad e independencia de intérpretes a través de la compilación estática.

Este estudio siguió los siguientes objetivos principales: analizar el estado del arte las herramientas de automatización; comparar las soluciones existentes para identificar su usabilidad y limitaciones relacionadas a dependencias; diseñar el controlador de línea de comandos basado en Go capaz de gestionar tareas y artefactos a través de nodos en una red; y definir un modelo de configuración basado en YAML para mejorar la usabilidad y flexibilidad de la herramienta. `Shoal` integra estos elementos en una solución de automatización ligera, extensible y segura.

Los resultados confirman que la automatización de tareas puede ser conseguida a través de un ejecutable autocontenido, validando a `Shoal` como una alternativa viable y eficiente a las herramientas existentes, y sentando las bases para futuros estudios en la automatización libre de intérpretes.

Abstract

The increasing complexity and heterogeneity of the IT infrastructures have converted the automatization into a fundamental aspect in the modern system management. Configuration management tools like Ansible, Puppet and Chef have simplified a lot the configuration and deployment process in distributed systems. However, many of these tools depend on external runtimes or agents, limiting its use in minimal or controlled environments. This work addresses this limitation through the design and implementation of `Shoal`, a prototype tool for task automation and application deployment that works without depend on external interpreters.

Developed in Go programming language, `Shoal` adopts a client-server architecture where a controller node communicates securely with the managed nodes through SSH. The tool executes automation tasks defined in configuration files based on YAML using a declarative syntax designed for clarity and ease of use. Its internal engine supports concurrent task execution in Debian-based and Windows systems, achieving portability and independence of interpreters thanks to the static compilation.

This study pursued the following main objectives: analyze the state of art of the automation tools; compare the existing solutions to identify their usability and limitations related to their dependencies; design a command-line based controller in Go able to manage tasks and artifacts through the nodes in a network; define the configuration model based on YAML to improve the tool's usability and flexibility. `Shoal` integrates all these elements into a lightweight, extensible and secure automation solution.

The results confirm that task automation can be achieved through a self-contained executable, validating `Shoal` as a feasible and efficient alternative to the existing tools, and laying the foundation for future studies in interpreter-free automation.

Capítulo 1

Introducción

El manejo eficiente y escalable de sistemas es hoy en día un requisito fundamental para las operaciones de TI modernas. Herramientas de automatización como Ansible, Puppet y Chef han surgido para abordar esta necesidad, permitiendo optimizar tareas repetitivas de optimización y despliegue de aplicaciones. Sin embargo, estas herramientas suelen depender de intérpretes externos como Python o Ruby en los nodos controlados, lo que supone limitaciones en ambientes de recursos limitados y/o con políticas de dependencias estrictas.

En sistemas de alta seguridad o instalaciones mínimas, estas dependencias introducen una complejidad innecesaria, problemas de compatibilidad y violaciones a las restricciones del sistema. Además, la instalación y despliegue de estos intérpretes en cada nodo agrega carga administrativa e incrementa la superficie de un posible ataque.

Este trabajo propone la implementación de una herramienta prototipo desarrollada en Go, que permita la automatización de tareas y despliegue de aplicaciones mediante una arquitectura cliente-servidor. La principal ventaja reside en la eliminación de la necesidad de intérpretes externos en los nodos controlados. Con la compilación de binarios independientes, Go ofrece una alternativa ligera y sin dependencias.

La solución propuesta busca simplificar los flujos de trabajo de automatización, garantizando al mismo tiempo la compatibilidad en ambientes limitados. Se dirige a sistemas donde las herramientas tradicionales no son suficientes, ofreciendo un enfoque robusto y sostenible para la automatización de tareas y despliegues.

1.1. Descripción general del problema

Herramientas de automatización ampliamente usadas como Ansible, Puppet y Chef, requieren de intérpretes externos en cada uno de los nodos controlados. Si bien esto es eficaz en muchos escenarios, esta dependencia se convierte en una limitación en ambientes con políticas de seguridad estrictas, instalaciones mínimas o aprovisionamiento de software restrictivo.

La instalación y mantenimiento de estos intérpretes no se alinean al objetivo de simplificar la administración del sistema. Pues, se introduce pasos de configuración adicionales, aumentan la carga administrativa y pueden generar problemas de compatibilidad (versiones incompatibles de software).

Estos desafíos reducen la aplicación de la automatización en ambientes restrictivos. Por lo tanto, se necesita una solución que evite estas dependencias y, al mismo tiempo, permita la automatización de tareas y despliegue de aplicaciones de manera robusta y fiable.

1.2. Objetivos

1.2.1. Objetivo general

Implementar un prototipo de herramienta, mediante el diseño de una interfaz de línea de comandos funcional en Go, para la automatización de tareas y despliegue de aplicaciones en nodos controlados bajo una arquitectura cliente-servidor sin dependencias de intérpretes externos.

1.2.2. Objetivos específicos

- Analizar el estado del arte de las herramientas usadas para la automatización de tareas y despliegues de aplicaciones, mediante la revisión sistemática de la bibliografía, con la finalidad de obtener las últimas investigaciones realizadas en esta área de estudio.

- Comparar las diferentes herramientas existentes en esta área, mediante la revisión de sus documentaciones, para determinar sus características de usabilidad y sus limitaciones.

- Diseñar una interfaz de línea de comandos (controlador) compatible con Debian Linux o macOS, usando el lenguaje de programación Go y el protocolo de comunicación seguro SSH, para la configuración de tareas y envío de artefactos de despliegue en nodos controlados (Debian Linux o Windows Server) en una misma red.

- Diseñar la estructura de los archivos de configuración que usará el controlador, mediante la utilización del formato YAML, para mejorar la usabilidad de la herramienta.

- Implementar un prototipo de interfaz de línea de comandos configurable (controlador), mediante el uso de archivos YAML y el protocolo de comunicación SSH, para la automatización de tareas y envío de artefactos de despliegue de aplicaciones en nodos controlados (Debian Linux o Windows Server) bajo una arquitectura cliente-servidor sin dependencias de intérpretes externos.

1.3. Contribuciones

Este trabajo presenta las siguientes contribuciones principales:

- **Implementación de una herramienta automatización ligera en Go**
  Se desarrolló un prototipo de herramienta en el lenguaje de programción Go para automatizar tareas y desplegar aplicaciones en nodos controlados sin depender de intérpretes externos como Python o Ruby. Este enfoque evita problemas comunes de dependencia y compatibilidad de las herramientas tradicionales.

- **Arquitectura cliente-servidor con comunicación basada en SSH**
  El prototipo sigue un modelo cliente-servidor donde el controlador central se comunica con los nodos administrados mediante conexiones SSH seguras. Este diseño garantiza compatibilidad con infraestructuras existentes, eliminando la necesidad de agentes persistentes en los nodos.

- **Configuración usando archivos YAML**
  La herramienta utiliza YAML para definir tareas de automatización y despliegue, ofreciendo un formato legible y estructurado. Esto mejora la claridad y la facilidad de uso, permitiendo que la configuración sea accesible incluso para usuarios con poca experiencia en programación.

- **Soporte de variables y plantillas en configuraciones**
  La herramienta permite definir y reutilizar variables dentro archivos YAML, lo que facilita configuraciones modulares y fáciles de mantener. Esto reduce la duplicación y mejora la consistencia en las tareas de automatización.

- **Portabilidad a través de binarios estáticos**
  Gracias al soporte nativo de Go para la compilación estática y la compilación cruzada, la herramienta puede ejecutarse en varios sistemas operativos (SO), aunque en este trabajo, la herramienta tendrá como objetivo ejecutarse en Linux Debian y macOS (para el nodo controlador), puede extenderse para soportar otros SO.

- **Compatibilidad con ambientes restringidos o controlados**
  Al eliminar la necesidad de dependencias externas y confiar en la comunicación por SSH, esta herramienta es adecuada para entornos mínimos, aislados o estrictamente controlados donde las herramientas de automatización conocidas son poco prácticas.

1.4. Organización del manuscrito

Este manuscrito está organizado en cuatro capítulos, cada uno abordando una fase específica del proceso de investigación y desarrollo.

El Capítulo 1 introduce la motivación y el contexto del estudio, define el problema a investigar, los objetivos general y específicos, y las contribuciones del proyecto.

El Capítulo 2 proporciona los fundamentos teóricos del trabajo. Se revisa el estado del arte de las herramientas de gestión de configuración y automatización, se define conceptos claves como gestión de configuración, automatización de tareas y arquitectura cliente-servidor, y analiza herramientas existentes (como Ansible, Puppet y Chef) y sus limitaciones. Además, se discute las tecnologías relevantes usadas en la solución propuesta incluyendo Go, YAML y SSH.

El Capítulo 3 detalla el diseño e implementación de `Shoal`, la herramienta de automatización propuesta. Se define y se explica los requisitos del sistema, arquitectura, archivos de configuración, diseño de la interfaz de línea de comandos, con ayuda de diagramas y ejemplos. Este capítulo también presenta el proceso de validación, demostrando cómo `Shoal` cumple con sus requisitos funcionales y no funcionales.

El Capítulo 4 expone el análisis y discusión de los resultados. Compara a `Shoal` con las herramientas de automatización de tareas existentes, identifica sus limitaciones actuales, y propone futuras áreas de mejora. Finalmente, se presentan las conclusiones del trabajo, resume las principales contribuciones y confirma la viabilidad de una herramienta de automatización independiente de intérpretes externos.

Capítulo 2

Marco Teórico Referencial

Este capítulo revisa los fundamentos teóricos, las soluciones existentes y las tecnologías relevantes para el desarrollo de la herramienta propuesta. Comienza con un análisis del "Estado del Arte" de las herramientas para la gestión de la configuración y automatización de tareas, seguido de "Definiciones Previas" que aclaran conceptos claves como la gestión de la configuración, automatización de tareas, modelos de arquitectura y DSLs. La sección "Herramientas de CM existentes" examina Ansible, Puppet y Chef en detalle, resaltando su arquitectura, DSL, requerimientos de sistema, fortalezas y limitaciones, y concluye con un análisis comparativo de las mismas. Finalmente la sección "Tecnologías relevantes" destaca el lenguaje de programación Go, formato YAML y el protocolo SSH como las tecnologías claves para lograr portabilidad, dependencias mínimas y facilidad de uso en una herramienta de gestión de configuración.

2.1. Estado del Arte

La literatura acerca de las herramientas para la configuración y automatización de tareas revela una clara evolución de las mismas, partiendo desde primeros enfoques manuales hasta sistemas cada vez más sofisticados capaces de gestionar infraestructuras complejas y distribuidas. Esta progresión ha dado lugar a sistemas declarativos totalmente automatizados, cuyo objetivo radica en mejorar la consistencia, la escalabilidad y la tolerancia a fallos.

Las primeras herramientas para el manejo de configuraciones, como por ejemplo *CMT*, se enfocaban en entornos estáticos con estructuras de dependencia simples y una escalabilidad limitada, demostrando los desafíos que se han enfrentado en el manejo de infraestructuras en crecimiento con diversos tipos de nodos y sistemas operativos. Además, estas primeras herramientas utilizaban archivos de texto plano para describir las configuraciones, careciendo de la estructura y claridad de formatos más legibles como *YAML*, lo que evidencia cómo han evolucionado estas herramientas para ser más fáciles de mantener y entender [Arnault, 2000].

A medida que los requisitos de los sistemas fueron aumentando, surgieron nuevas herramientas con paradigmas declarativos y la misión de automatizar la lógica de configuración. Por ejemplo, herramientas como MetaConfig y Charon descritas en [Nielsen et al., 2011, Dolstra et al., 2013], adoptaron modelos que priorizan la modularidad, la reutilización de plantillas de configuración y la definición del estado del sistema, principios clave que siguen siendo fundamentales en la automatización moderna. Estas ideas se implementaron con mayor amplitud en sistemas como Engage [Fischer et al., 2012] que orquesta despliegues mediante la abstracción de tareas operativas en planes estructurados. Esta naturaleza declarativa es la base para la reproducibilidad y mantenimiento de tareas de configuración.

El auge de las prácticas DevOps aceleró la adopción de herramientas de automatización como Puppet, Chef y Ansible. Estas herramientas cuentan con un lenguaje específico de dominio (*Domain-Specific Language - DSL*) y capacidades poderosas de orquestación que soportan entornos híbridos y flujos de trabajo de integración continua [Kostromin, 2020, Likitha, 2022].

A pesar de estas capacidades, estudios comparativos han identificado algunos desafíos. Por ejemplo, en un estudio que analiza herramientas populares como Puppet, Chef, Ansible y SaltStack, los investigadores resaltaron que estas plataformas requerían la instalación de intérpretes o agentes adicionales en los nodos de destino, lo que complica su adopción en entornos limitados, restrictivos y/o de alta seguridad [Kostromin,

2020]. Un caso de estudio centrado en el manejo de la configuración de un sistema de control [Hardion et al., 2013] respalda estos desafíos, señalando cómo herramientas como Puppet y Chef introducen una curva de aprendizaje y requisitos adicionales (agentes), mientras que herramientas basadas en Python como Ansible, aunque más simples, aún asumían ciertas dependencias a nivel de sistema que podrían no estar disponibles de forma predeterminada.

Ansible, si bien ha sido una de las herramientas más seleccionadas por su diseño sin agentes y sus *playbooks* (archivos de configuración de tareas en Ansible) escritos en YAML, también ha presentado problemas relacionados con dependencias externas de Python en los nodos controlados [Rastenis, 2022]. Casos reales destacan fallos provocados por falta de módulos de Python en los nodos controlados como `python3-apt` y `libselinux-python3`, los cuales son requeridos en ciertos módulos de Ansible [realtebo, 2021, Vummadi, 2020, Semushin, 2017]. Esto apunta a la necesidad de soluciones de automatización más portables que funcionen sin la asunción de intérpretes y/o librerías preinstaladas de los nodos destino.

En resumen, el estado del arte revela un panorama maduro pero diverso de herramientas de automatización de configuración y despliegues. Las soluciones existentes ofrecen funcionalidades avanzadas pero pueden presentar restricciones operativas que limitan su uso en entornos mínimos. Estos estudios motivan el desarrollo de nuevas herramientas, como la propuesta en este trabajo, que combinen la simplicidad y portabilidad de interfaces de línea de comandos (*Command Line Interface - CLI*) con la flexibilidad de configuraciones en YAML, y que operen sobre protocolos de comunicación seguros, como SSH, reduciendo las dependencias de intérpretes o agentes preinstalados en los nodos controlados.

2.2. Definiciones Previas

2.2.1. Gestión de Configuración - CM

La Gestión de Configuración (*Configuration Management - CM*) es el proceso de definir, aplicar y garantizar consistentemente el estado deseado de sistemas informáticos y software a lo largo de su ciclo de vida. Esto asegura que todos los nodos en un entorno distribuido o a gran escala se ajusten a definiciones predefinidas reduciendo la desviación de la configuración prevista, que a menudo conduce a errores o riesgos de seguridad [Rand, 2021, Red Hat, 2023a].

Históricamente, las tareas de configuración se realizaban de manera

manual, lo que generaba errores humanos, inconsistencias y limitaciones de escalabilidad. Las herramientas de gestión de configuración surgieron para automatizar estas tareas mediante definiciones repetibles con control de versiones, garantizando la consistencia y auditabilidad de despliegues [Red Hat, 2023a].

Las primeras herramientas como Cfengine introdujeron nuevos enfoques para la gestión de estados de sistema y sentaron las bases para las prácticas de configuración modernas; demostrando cómo la automatización podía mejorar la resiliencia y reducir la complejidad operativa mediante la aplicación consistente de estados predefinidos [Burgess, 2005, Delaet et al., 2010].

Según [Delaet et al., 2010], las herramientas de CM permiten a los administradores gestionar la infraestructura de forma proactiva especificando configuraciones a un grupo de nodos en lugar de ejecutar comandos individuales. Este modelo mejora la tolerancia a fallos y permite una rápida adaptación a cambios en la infraestructura. Además, debe facilitar la trazabilidad, control de versiones y auditoría; funciones esenciales en entornos productivos.

Estas características mejoran significativamente la eficiencia y escalabilidad en el manejo de la configuración como una práctica fundamental en las operaciones de TI (Tecnología de la Información) modernas [Hintsch et al., 2016].

2.2.2. Automatización de tareas

La automatización de tareas es el proceso de aplicar tecnología para completar una tarea u optimizar un flujo de trabajo y así mejorar la productividad. Esto implica automatizar acciones rutinarias o que consumen mucho tiempo, eliminando la intervención humana y permitiendo que las personas se enfoquen en tareas de mayor valor al negocio [Diane and Stryker, 2024].

De acuerdo a [Heaton, 2025], algunos de los beneficios de la automatización de tareas son:

- Eficiencia: Al reducir el tiempo utilizado en tareas manuales, los operadores pueden completar las tareas mucho más rápido y dedicar menos tiempo a tareas repetitivas.
- Satisfacción de los operadores: Ayuda a reducir las cargas de trabajo estresantes, liberando el tiempo de los operadores para que se concentren en tareas más importantes.

- Menos errores: Cuando las herramientas de automatización se hacen cargo de las tareas manuales y repetitivas, se reduce el riesgo de error humano.
- Reducción de costos: El uso de la automatización reduce la necesidad de trabajo manual, lo que permite que las empresas puedan reasignar fondos en otras áreas.
- Escalabilidad: Ayuda a las empresas a optimizar flujos de trabajo complicados sin aumentar el gasto.

2.2.3. Arquitectura Cliente-Servidor

La arquitectura cliente-servidor se define como un modelo de computación donde un equipo llamado *"cliente"* solicita un recurso a otro equipo llamado *"servidor"* a través de una conexión de red. El servidor recibe la solicitud, la procesa y responde al cliente. En este modelo, puede incluir uno o más clientes que solicitan recursos o servicios a uno o más servidores que trabajan en conjunto para procesar una petición [Mbuguah et al., 2024].

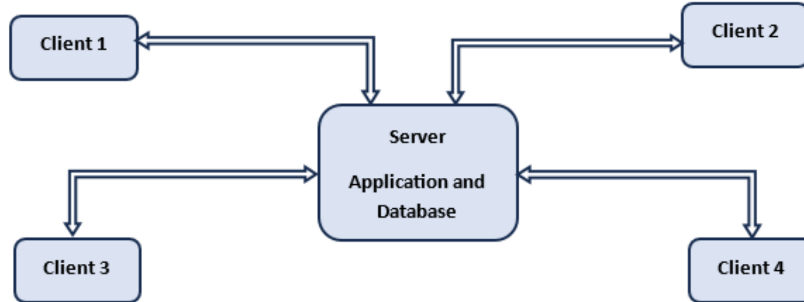

Figura 2.1: Arquitectura Cliente-Servidor
Fuente: [Mbuguah et al., 2024]

Una característica importante de esta arquitectura es su escalabilidad tanto vertical como horizontal, pues más y más servidores pueden conectarse para gestionar la carga de trabajar. Asimismo, se puede aumentar las características del servidor, como la RAM y el CPU. En este modelo, los equipos cliente y servidor pueden funcionar con recursos de hardware y software heterogéneos [Mbuguah et al., 2024].

2.2.4. Modelos de configuración - Declarativos e Imperativos

En el contexto de CM, un modelo de configuración es el enfoque utilizado para definir la configuración de un sistema. Entre los modelos más relevantes se encuentran los modelos declarativo e imperativo.

El modelo imperativo se enfoca en el *"cómo"*. En este modelo se describe cómo hacer la tarea con la expectativa de que se obtendrá el resultado deseado [Sen, 2025]. Por lo general, en este enfoque los administradores requieren un profundo conocimiento del entorno que están configurando [Cisco DevNet, 2025].

Por otro lado, el modelo declarativo se enfoca en el *"qué"* y describe qué necesita suceder (el estado deseado) dejando los detalles de cómo hacerlo al sistema o herramienta que se está usando [Sen, 2025]. En este caso, los administradores pueden solicitar un cambio utilizando una declaración general del resultado deseado, siendo la herramienta usada la responsable de traducir el resultado deseado a los equipos de la red [Cisco DevNet, 2025].

Es importante notar que tanto los modelos imperativos como declarativos son falibles: el primero requiere una compleja solución de problemas cuando las cosas fallas, mientras que el segundo requiere confiar en que se han alcanzado los estados deseados (con escasa verificación). Ambos modelos pueden ser problemáticos en ciertos escenarios; y por lo tanto, ninguno de éstos debe implementarse como la solución única para la gestión de la configuración [Sen, 2025].

2.2.5. Gestión de configuración basada en agentes y sin agentes

Una herramienta de gestión de configuración se puede clasificar en dos categorías: basada en agentes y sin agentes.

En la herramientas basadas en agentes, se despliega un software (agente) a lo largo de toda la arquitectura. Estos agentes recolectan datos usando diferentes APIs (*Application Programming Interface*) y llamadas de sistema. Luego un equipo central recibe la información de estos agentes, los tabula, provee resultados y/o envía acciones a ejecutar [Fashakin, 2025]. Algunas de las ventajas son:

- Monitoreo: Puesto que un agente se encuentra constantemente recolectando datos, se logra un monitoreo óptimo y profundo del sistema y su rendimiento.
- Menor ancho de banda: Reduce los requisitos de ancho de banda al

recopilar los datos localmente y transmitir únicamente los resultados procesados, reduciendo también así la carga de procesamiento del equipo central.

Por el contrario, las herramientas sin agentes, usan un único equipo/agente central para supervisar toda la infraestructura. Este equipo central supervisa los dispositivos de la infraestructura mediante APIs, SSH e interfaces del sistema para determinar el estado y la disponible como también para enviar acciones de ejecución [Fashakin, 2025]. Algunas ventajas son:

- Rendimiento mejorado: Esta arquitectura proporciona un mejor mantenimiento y eficiencia al utilizar menos recursos, menos memoria y no requieren instalación adicional.
- Despliegues rápidos: Mientras que en las soluciones basadas en agentes es necesario desplegar software en cada nodo (lo que puede tomar tiempo), esta solución puede desplegar y configurar aplicaciones de manera centralizada.
- Bajo costo: Tener un agente instalado en cada nodo, puede incurrir en costos adicionales al instalar actualizaciones. Al implementar una arquitectura sin agentes se ahorrará tiempo y costos en casos de escalamiento.

2.2.6. Intérprete

Un intérprete es un elemento activo que se encarga de traducir y ejecutar al mismo tiempo un programa. Convierte una sentencia de programa a lenguaje de máquina, la ejecuta y pasa a la siguiente. Esto difiere de los programa regulares que se presentan al sistema como instrucciones en código binario. Los programas interpretados permanecen en el lenguaje de programación que fueron escritos, siendo texto legible para el humano [PCMag, 2025].

Una ventaja importante de un lenguaje interpretado es que generalmente puede ejecutarse en más de una plataforma. El código fuente es el mismo, pues el intérprete se encarga de convertir este código en lenguaje de máquina. Sin embargo, el intérprete debe estar en el lenguaje nativo de la máquina donde se ejecuta, lo que implica que cambios en el lenguaje requieren intérpretes actualizados para cada plataforma [PCMag, 2025].

No obstante, los programas interpretados se ejecutan más lento que sus contrapartes compiladas. Mientras que el compilador traduce el programa completo antes de ejecutarlo, los intérpretes traducen línea por línea durante la ejecución [PCMag, 2025].

2.2.7. Lenguaje Específico de Dominio - DSL

Un Lenguaje Específico de Dominio (*Domain-Specific Language - DSL*) es un lenguaje de programación con alto nivel de abstracción optimizado para una clase de problema específico [JetBrains, 2025].

Este tipo de lenguaje es usualmente menos complejo que los de propósito general como C, Python o Java. Generalmente, los DSLs son desarrollados en coordinación con expertos en el campo en el que el DSL está siendo diseñado. En muchos casos, los DSLs están destinados a ser usados por personas no relacionadas al software sino con un fluido dominio de lo que aborda el DSL [JetBrains, 2025].

Un ejemplo conocido de DSL es *SQL* (*Structured Query Language*), un lenguaje declarativo para gestionar y hacer consultas a bases de datos. El enfoque de SQL, que especifica qué necesita hacerse y deja el cómo al motor de la base datos, resalta cómo los DSLs abstraen la complejidad de un dominio específico. Esto permite que cualquier persona pueda obtener datos de una base de datos sin necesidad de saber cómo se almacenan y particionan físicamente [Dagster Labs, 2024].

Otro DSL público conocido es Terraform, diseñado para infraestructura como código (*Infraestructure as Code - IaC*). Terraform utiliza un enfoque declarativo para definir y administrar la infraestructura, permitiendo a los usuarios especificar el estado final deseado de configuración de su infraestructura [Dagster Labs, 2024].

YAML (*YAML Ain't Markup Language*) es otro DSL muy popular y comúnmente elegido por las herramientas para definir configuraciones, gracias a su naturaleza legible para humanos [Dagster Labs, 2024].

2.3. Herramientas de CM existentes

Para comprender mejor el panorama actual en el contexto de la automatización de tareas y la gestión de configuración, es fundamental revisar herramientas ampliamente usadas que utilizan diferentes enfoques para abordar este problema. Herramientas como Ansible, Puppet y Chef se han utilizado extensamente en la industria y en el ámbito académico, ofreciendo soluciones maduras con diferentes arquitecturas y paradigmas operativos. En este apartado, se ofrece una visión general de estas herramientas destacando sus fortalezas y limitaciones, especialmente en relación a la dependencia de intérpretes y agentes, y su usabilidad en diferentes entornos.

2.3.1. Ansible

Visión General

Ansible es una herramienta de automatización de código abierto desarrollada en 2012 por Michael DeHaan, diseñada para simplificar tareas como las gestión de configuración, la implementación de aplicaciones a través de archivos de configuración escritos en YAML llamados "playbooks", y una arquitectura sin agentes [Courdent, 2024]. Esta herramienta se opera desde un nodo central (donde Ansible es instalado) y se comunica con los nodos controlados via SSH (Linux/Unix) o WinRM (Windows), prescindiendo de la necesidad de contar con agentes persistentes en cada sistema [PyNet Labs, 2025].

El nombre "Ansible" se inspiró en un dispositivo ficticio de comunicación instantánea que se menciona en la novela "Ender's Game" de Orson Scott, reflejando la ambición de la herramienta por proporcionar una automatización fácil e inmediata en toda una infraestructura. Ganó gran aceptación debido a su simplicidad y mínima sobrecarga operativa. En 2015, Red Hat adquirió Ansible, integrándolo a su ecosistema, manteniendo las versiones Community y Enterprise de Ansible [Kadima, 2023].

Desde su lanzamiento inicial, Ansible se ha mantenido en constante desarrollo y crecimiento de su comunidad, lo que le ha permitido contar con un gran ecosistema de módulos, integraciones con plataformas de la nube y soporte extendido para diversos sistemas. Ansible sigue principios de diseño fundamentales como dependencias mínimas, arquitectura sin agentes, simplicidad e idempotencia, los cuales permiten que la automatización sea accesible, confiable y escalable [Ansible, 2025a].

Arquitectura

Ansible está construido alrededor de una arquitectura cliente-servidor. Consta de un nodo de control (servidor), que es normalmente un sistema basado en Linux con Ansible instalado, el cual ejecuta tareas en nodos remotos (clientes) mediante conexiones seguras SSH sin la necesidad de agentes persistentes [Red Hat, 2025, Ansible, 2025a], como se puede ver en la Fig. 2.2.

Las tareas de automatización de Ansible son ejecutadas a través de "módulos", los cuales son pequeños programas (usualmente en Python) que son temporalmente transferidos y ejecutados en los nodos administrados. Una vez que el módulo completa su tarea, se elimina sin dejar procesos residuales. La comunicación con los módulos se realiza mediante las entrada y

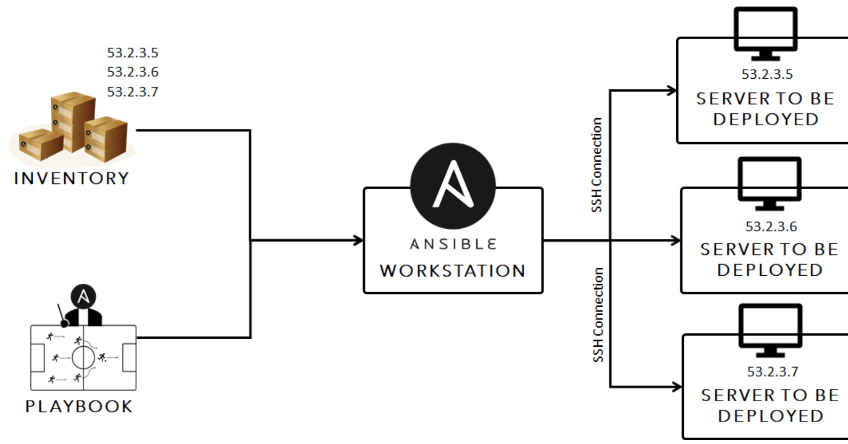

Figura 2.2: Ansible - Arquitectura
Fuente: [Prabhu, 2024]

salida estándar, y los resultados se devuelven en formato JSON. Este modelo garantiza un alto grado de transparencia y evita la necesidad de servicios persistentes en segundo plano en los nodos destino [Ansible, 2025b, Ninawe, 2025].

Ansible define algunos conceptos fundamentales para entender su funcionamiento [Ansible, 2025c]:

1. Nodo de control:
   La máquina donde se ejecuta el Ansible CLI, y desde donde se envían las tareas a los nodos administrados [Ansible, 2025c].

2. Nodos administrados:
   También llamados "hosts", son los dispositivos destino los cuales se pretenden gestionar con Ansible [Ansible, 2025c].

3. Inventario:
   Ansible utiliza un "inventario" para definir el conjunto de nodos a gestionar y sus agrupaciones. Estos "inventarios" se escriben en archivos de formato INI o YAML [Ansible, 2025d].

4. Playbooks:
   La lógica de automatización de Ansible se escribe en "playbooks", los cuales son archivos en formato YAML que describen las tareas,

los nodos destino y orden de ejecución de tareas. Los "playbooks" se componen de uno o varios "plays" (la unidad básica de ejecución de Ansible), cada uno de los cuales contienen tareas que invocan módulos apuntando a un grupo específico de nodos. Soporta variables, condicionales, mecanismos de manejo de errores, permitiendo a los operadores definir flujos de trabajo robustos y reutilizables [Ansible, 2025c,e,f].

5. Módulos:
   El código o los binarios que Ansible copia y ejecuta en cada nodo destino (cuando es necesario) para realizar la acción definida en cada tarea. Cada módulo tiene un uso particular y están agrupados en "colecciones" [Ansible, 2025c].

6. Plugins:
   Los plugins son fragmentos de código que amplían la funcionalidad de Ansible. Los plugins pueden controlar como te conectas a un nodo administrado o incluso controlar lo que se muestra en la consola [Ansible, 2025c].

7. Colecciones:
   Un formato en el que se distribuye contenido de Ansible que puede contener playbooks, módulos y plugins [Ansible, 2025c].

El flujo de trabajo de Ansible usualmente sigue los siguientes pasos [Ninawe, 2025]:

- El nodo de control procesa el playbook y el inventario.
- Inicializa sesiones SSH con los nodos destino.
- Las tareas son realizadas mediante la transferencia y ejecución de módulos en los nodos destino.
- Los resultados son recolectados en tiempo real.

Ansible también soporta ejecución paralela en múltiples nodos, lo que permite una automatización rápida y escalable. Su sistema modular, basado en plugins, ofrece extensibilidad, manteniendo la plataforma ligera y fácil de adoptar [Ansible, 2025g].

DSL

Ansible utiliza un DSL construido sobre el formato YAML para describir tareas de automatización de una manera estructurada y legible para humanos. Aunque YAML es un lenguaje de serialización de datos de propósito general, Ansible impone un esquema y una semántica específicos para su uso, convirtiéndolo en un DSL adecuado para el contexto de automatización de tareas [Red Hat, 2023]. Este enfoque permite a los administradores de sistemas expresar flujos de trabajo de configuración complejos y secuencias de tareas de forma clara y declarativa, sin necesidad de escribir instrucciones imperativas.

En el núcleo del DSL de Ansible se encuentran los playbooks, que definen uno o más "plays". Cada "play" se dirige a un grupo de nodos destino y especifica una secuencia de tareas a ejecutar, finalmente, estas tareas suelen invocar módulos de Ansible para alcanzar el estado deseado [Ansible, 2025e]. La estructura del DSL es concisa e intuitiva; por ejemplo, una tarea para instalar un servidor web "nginx" se puede escribir como se ve en el Listado 1.

```yaml
- name: Install Nginx
  apt:
    name: nginx
    state: present
```

Listado 1: Ejemplo de tarea en Ansible

Esta sintaxis facilita la comprensión y el mantenimiento de la lógica de automatización, incluso para usuarios que no son desarrolladores de software. Se pueden introducir variables en varios niveles del playbook, como de forma global, por host o por rol, y se pueden inyectar plantillas dinámicas mediante "Jinja2", un potente motor de plantillas integrado en Ansible. Este sistema de plantillas soporta condicionales, bucles y filtros, lo que permite configuraciones reutilizables y flexibles [Ansible, 2025h].

Otro pilar del DSL de Ansible son los "roles", que proporcionan una forma estandarizada de agrupar tareas, handlers, variables, archivos y plantillas relacionadas en componentes modulares. Estos "roles" promueven la reutilización y ayudan a reforzar la estructura en proyectos grandes. Cuando se aplica un "rol" en un playbook, su estructura de directorios determina automáticamente la lógica de ejecución, sin necesidad de incluir archivos manualmente [Ansible, 2025i].

El diseño del DSL de Ansible prioriza la idempotencia, garantizando, en la

mayor parte de los casos, que las ejecuciones repetidas del mismo playbook generen el mismo estado del sistema sin causar cambios imprevistos [Ansible, 2025e].

En resumen, el DSL de Ansible basado en YAML logra un equilibrio entre simplicidad y expresividad, permitiendo a los usuarios describir configuraciones complejas de sistema de forma declarativa, modular y reproducible.

Requisitos de sistema

Ansible es conocido por sus mínimos requisitos de sistema, especialmente en los nodos controlados. Su arquitectura sin agentes, basada en protocolos de comunicación estándar como SSH, reduce significativamente la necesidad de software preinstalado. Sin embargo, para funcionar correctamente, ciertos requisitos se deben cumplir tanto en el node de control como en los nodos a controlar.

Los requisitos en el nodo de control son [Ansible, 2025j]:

- Sistema operativo: Se requiere de sistema tipo Unix, normalmente Linux o macOS, o Windows pero usando WSL (*Windows Subsystem for Linux*).
- Python: Python 3.10 o superior es recomendado para usar las nuevas versiones de Ansible [Ansible, 2025k]. Python debe estar disponible en la máquina de control para ejecutar la CLI (*Console Line Interface*) de Ansible y los módulos principales.
- Cliente SSH: Debe contar con un cliente OpenSSH estándar para iniciar conexiones con los nodos controlados.

Los requisitos de los nodos administrados son [Ansible, 2025j]:

- Servidor SSH: Un servidor OpenSSH debe estar instalado y habilitado.
- Python: Una versión compatible de Python (2.7 o mayor dependiendo de la versión de Ansible instalada en el nodo de control) debe estar instalada en estos nodos [Ansible, 2025k].
- Powershell: En el caso de sistemas Windows se requiere Powershell 5.1 para su correcto funcionamiento con las nuevas versiones de Ansible [Ansible, 2025k].

Fortalezas

- **Arquitectura sin agentes**
  Una de las características más distintivas de Ansible es su diseño sin agentes, basándose únicamente en SSH para la comunicación [Red Hat, 2025].

- **Facilidad de uso**
  EL DSL basado en YAML de Ansible permite a los usuarios escribir tareas de automatización claras y legibles. Esto reduce la curva de aprendizaje y facilita la cooperación entre equipos de desarrolladores y operaciones [Ansible, 2025e].

- **Idempotencia y modelo declarativo**
  La mayoría de los módulos de Ansible son idempotentes, lo que garantiza que múltiples ejecuciones del mismo playbook conduzcan hacia un estado consistente del sistema sin producir efectos secundarios [Ansible, 2025a,e].

- **Modular y extensible**
  Ansible provee una amplia colección de módulos integrados y permite la creación de módulos y plugins personalizados, lo que permite a las organizaciones adaptar la automatización a sus necesidades específicas [Ansible, 2025b].

- **Integración con proveedores de infraestructura y nube**
  Ansible ofrece funcionalidades de inventarios dinámicos y módulos que se integran con proveedores de la nube como AWS y Azure, lo que lo hace ideal para administrar infraestructuras híbridas y nativas de la nube [Ansible, 2025l,m].

Limitaciones

- **Dependencia de Python en los nodos administrados**
  Aunque Ansible en sí se ejecuta en el nodo de control, requiere que Python esté instalado en los nodos a controlar para ejecutar la mayoría de sus módulos. Esto puede ser problemático en instalaciones mínimas de sistemas operativos donde Python o algunas librerías (por ejemplo `python3-apt`, `dbus-python` o `python3-selinux`) del mismo podrían no estar presentes por defecto [Ansible, 2025k, realtebo, 2021, Vummadi, 2020, Semushin, 2017].

- **Falta de un gestionamiento de estado continuo**
  A diferencia de otras herramientas como Puppet, Ansible se ejecuta a demanda, no de forma continua, lo que significa que los sistemas podrían desviarse del estado deseado a menos que se repita la ejecución de Ansible con regularidad [Arora, 2025].

2.3.2. Puppet

Visión General

Puppet es una herramienta de gestión de configuración creada por Luke Kanies en 2005, cuyo objetivo es automatizar el aprovisionamiento y mantenimiento constante de las configuraciones de sistemas a gran escala. Según Kanies, su trabajo en Puppet comenzó entre 2002 y 2003, pero no fue sino hasta 2005 que se dedicó por completo a él como una solución para los desafíos de la gestión de configuración a gran escala [Kanies, 2019, Brown and Wilson, 2012]. Puppet introdujo un DSL declarativo basado en Ruby que permite a los administradores definir estados de sistema deseados en lugar de escribir scripts de procedimiento para llegar a dicho estado [Puppet, 2025a].

Desde sus inicios, Puppet adoptó un modelo cliente-servidor: los nodos controlados, conocidos como agentes, se comunican periódicamente con un nodo maestro central para solicitar un catálogo de configuración y aplicar el estado especificado. Esta arquitectura contrasta con las herramientas basadas en modelo "push" al distribuir la responsabilidad de ejecución a los agentes, lo que mejora la escalabilidad y reduce la dependencia del servidor central [Brown and Wilson, 2012].

Puppet se centró inicialmente en entornos Unix, pero con la introducción de Puppet Enterprise en 2011, su soporte se extendió a Windows, dispositivos de red e infraestructura de nube [Cadman, 2021]. Puppet Enterprise también incorporó funciones como el control de acceso basado en roles (*Role-Based Access Control - RBAC*) y dashboards para reportes lo que le permitió consolidarse en sectores regulados y con un alto nivel de cumplimiento normativo [Puppet, 2022].

Hoy en día, Puppet sigue siendo una herramienta ampliamente usada para la configuración y automatización de sistemas. Su DSL robusto, su arquitectura basada en agentes y su amplio ecosistema de módulos lo han convertido en una opción confiable para organizaciones que gestionan infraestructuras complejas tanto en entornos locales como en la nube [Krause, 2024].

Arquitectura

Puppet emplea una arquitectura cliente-servidor (agente-servidor) para gestionar eficazmente las configuraciones de numerosos sistemas. Como se describe en la documentación oficial de Puppet Core, el servidor principal (Puppet Server) almacena la configuración deseada, que se compila en instrucciones específicas del sistema denominadas "catálogos". Los catálogos son documentos JSON que describen el estado deseado de un nodo agente específico [Puppet, 2025c]. Estos se transmiten mediante HTTPS con seguridad basada en SSL a los agentes de Puppet que se ejecutan en cada nodo administrado [Puppet, 2025b]. La Fig. 2.3 muestra visualmente la arquitectura de Puppet.

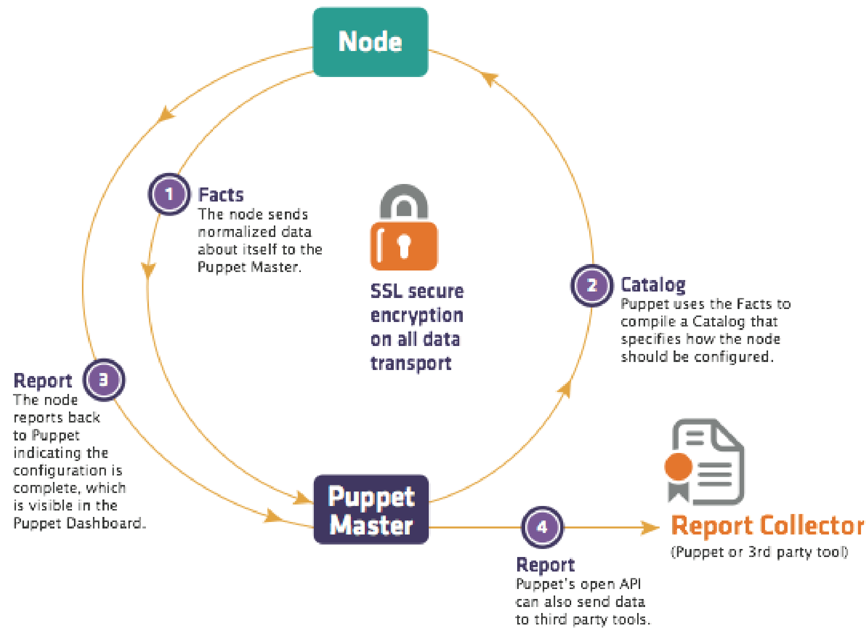

Figura 2.3: Puppet - Arquitectura
Fuente: [Brown and Wilson, 2012]

El primer paso esencial en el ciclo de ejecución de Puppet es la recopilación de la información del sistema de los nodos, denominada "facts". Cada agente de Puppet cuenta con una herramienta de inventario llamada "Facter", que es la encargada de recopilar los datos del sistema como el nombre de host, la dirección IP, la versión del sistema operativo y detalles de hardware. Estos

"facts" se envían al nodo maestro en forma de un archivo especial de código Puppet llamado "manifest". [Puppet, 2025c,d].

Del lado del servidor principal, se combinan estos archivos "manifests" con módulos para contruir un "catálogo", que define el estado deseado para cada nodo administrado. Este "catálogo" es solicitado y recibido por cada uno de los nodos agentes para aplicar y asegurar que el estado del nodo se mantenga en el estado definido. Al finalizar, el agente envía un reporte de los cambios realizados al servidor principal. Adicionalmente, todos los datos generados por Puppet, como "facts", "catalogs" y reportes, pueden ser almacenados en la base de datos de Puppet (PuppetDB), lo que permite que Puppet funcione más rápido y proporcione un API para que otras aplicaciones puedan acceder a los datos recopilados [Puppet, 2025c].

En versiones anteriores, Puppet ofrecía una arquitectura independiente, en la que los agentes compilaban sus propios "catálogos" usando la aplicación "Puppet apply". Sin embargo, Puppet ya no recomienda esta configuración ya que es difícil de mantener y proteger [Puppet, 2025b].

En resumen, Puppet ofrece una solución para la gestión de configuración robusta y escalable que admite flujos de trabajo centralizados y descentralizados. Su diseño separa los problemas de configuración entre agentes y servidor, lo que garantiza una comunicación segura, generación modular de catálogos y reportes completos.

DSL

Puppet utiliza un DSL declarativo para definir el estado deseado de los recursos del sistema. Este lenguaje permite a los usuarios expresar "qué" configuración se desea en el sistema, sin especificar cómo lograrla. Este modelo promueve la consistencia y reduce el error humano [Puppet, 2025e].

Los archivos de configuración de Puppet se escriben en archivos con extensión `.pp` y se componen de declaración de recursos (los elementos centrales de este DSL). Estos recursos pueden ser paquetes (*packages*), servicios (*services*) y archivos (*files*) [Puppet, 2025e,f]. Un ejemplo de archivo de configuración se puede ver en el Listado 2.

Cada recurso es un bloque que especifica un tipo de recurso (ej. `service`), el título del recurso (ej. `nginx`), y uno o más atributos (ej. `ensure` y `enable`) [Puppet, 2025g]. Puppet garantiza que estas declaraciones coincidan con el estado real del sistema, logrando así la idempotencia, donde aplicaciones repetidas de una configuración no resultan en cambios no deseados [Puppet, 2025h].

Además de las declaraciones básicas de recursos, el lenguaje soporta

```
package { 'nginx':
  ensure => installed,
}

service { 'nginx':
  ensure => running,
  enable => true,
}
```

Listado 2: Ejemplo de archivo de configuración en Puppet

[Puppet, 2025e]:

- Condicionales (`if`, `case`, `unless` y `selectors`) [Puppet, 2025i].
- Relaciones entre recursos (`before`, `require`, `notify` y `subscribe`) [Puppet, 2025j].
- Variables y tipos de datos.
- Reutilización de código a través de clases.
- Plantillas con Embedded Puppet (EPP) y Embedded Ruby (ERB) [Puppet, 2025k].

Aunque el lenguaje toma prestado elementos de la sintaxis de Ruby, Puppet ha evolucionado hasta tener un lenguaje de configuración fuertemente tipado, diseñado para facilitar la lectura y el mantenimiento [Reiher et al., 2023].

En resumen, este DSL permite la creación de definiciones de configuración reutilizables, mantenibles y portables, permitiendo a los administradores gestionar la infraestructura de forma consistente y eficiente.

Requisitos de sistema

Para operar de manera correcta, Puppet requiere requisitos específicos en software y hardware tanto en el nodo principal (*Puppet Server*) como en los nodos controlados (*Puppet Agents*).

El *Puppet Server* es el componente principal responsable de compilar los catálogos y entregarlos a los nodos agentes. Sus principales requisitos incluyen:

- Sistema operativo: Compatible con las principales distribuciones de Linux como Red Hat Enterprise Linux, Debian, Ubuntu y SUSE Linux Enterprise Server [Puppet, 2025r]. No es compatible con Windows ni macOS [Puppet, 2025s].
- Java: La aplicación de servidor de Puppet se ejecuta en la Máquina Virtual Java (*JVM - Java Virtual Machine*), por lo que, Java es requerido en el nodo principal. Las últimas versiones de Puppet Server requieren versions de Java 11 o 17 [Puppet, 2025r].
- Hardware: Las necesidades de recursos del nodo principal se ven afectadas por la cantidad de nodos agentes desplegados. Se recomienda mínimo un CPU de 2 núcleos y 4 GB de RAM para la administración máxima de 1000 nodos [Puppet, 2025t].
- Configuración de red: El nodo principal debe admitir conexiones entrantes en el puerto 8140 para su comunicación con los nodos agentes [Puppet, 2025u].

Por otro lado, los requerimientos de los nodos controlados son:

- Sistema operativo: Compatible con una amplia gama de sistemas operativos como Debian, Fedora, macOS, Microsoft Windows, Microsoft Windows Server, Red Hat Enterprise Linux, AmazonLinux, SUSE Linux Enterprise Server, Alma Linux, Rocky Linux, Oracle Linux, Scientific Linux y Ubuntu [Puppet, 2025v].
- Agente de Puppet: Todos los nodos administrados deben tener instalado el agente de Puppet.
- Hardware: Se recomienda mínimo un CPU de 1 núcleo y 512 MB de RAM [Puppet, 2025t].

El sistema en conjunto también requiere:

- Sincronización: Todos los nodos deben tener sus relojes sincronizados mediante NTP, pues Puppet utiliza certificados SSL para la autenticación en la comunicación de los nodos, los cuales (los certificados) son sensibles al tiempo [Puppet, 2025w].

Fortalezas

- **Madurez**
  Puppet ha estado en desarrollo activo desde 2005 y es ampliamente adoptado en entornos empresariales, demostrando alta estabilidad y soporte de la comunidad [Kanies, 2019].

- **Idempotencia**
  Los recursos de Puppet son aplicados de manera que ejecuciones repetidas del mismo no resultan en cambios no deseados. Esto garantiza la estabilidad y predictibilidad de ejecuciones consecutivas [Puppet, 2025h].

- **DSL declarativo**
  Su DSL permite a los usuarios definir el estado final deseado del sistema sin escribir explícitamente una lógica procedimental, mejorando la capacidad de mantenimiento y consistencia [Puppet, 2025a,e].

- **Agentes multiplataforma**
  Los agentes de Puppet se ejecutan en una amplia gama de sistemas operativos, incluidos Linux, Windows y macOS, lo que lo hace adecuado para infraestructuras heterogéneas [Puppet, 2025v].

- **Reportes y auditorías robustos**
  Puppet genera reportes detallados de los cambios realizados en cada ejecución, y puede integrarse con herramientas como PuppetDB, para el seguimiento del estado del sistema y auditorías [Puppet, 2025q].

- **Ecosistema modular**
  Puppet Forge ofrece una gran variedad de módulos reutilizables, lo que reduce la necesidad de crear definiciones desde cero [Puppet, 2025m].

- **Separación de datos y lógica**
  Con herramientas como Hiera, Puppet permite una configuración parametrizada que separa los datos del código, aumentando la flexibilidad y la reutilización del código [Puppet, 2025n].

- **Integración con herramientas externas**
  Puppet soporta integraciones externas para mejorar la flexibilidad y las capacidades de automatización. Como por ejemplo, su integración con Hiera y Bolt. Este último permite la ejecución de tareas según se requiera sin necesidad de un agente de Puppet [Puppet, 2025o].

Limitaciones

- **Curva de aprendizaje pronunciada**
  El DSL de Puppet y el uso de Hiera pueden ser un desafío para los principiantes, especialmente para aquellos que no están familiarizados con los patrones de programación funcional [Nolle, 2019].

- **Modelo persistente agente/servidor**
  El modelo agente-servidor de Puppet introduce una complejidad adicional. Requiere la configuración y mantenimiento de un servidor principal, certificados SSL y registros periódicos de agentes [Puppet, 2025a].

- **Instalación de agentes en cada nodo**
  A diferencia de las herramientas sin agentes, Puppet requiere que cada nodo a controlar ejecute un agente de Puppet. Esto añade una sobrecarga operativa y puede resultar inadecuado para ciertos entornos.

- **Sin ejecución ad-hoc nativa**
  La herramienta principal de Puppet no permite la ejecución de comandos ad-hoc por defecto. Aunque Puppet Bolt, otra herramienta desarrollada por Puppet, permite la operación sin agentes, trabajando de forma independiente al flujo de trabajo principal agente-servidor de Puppet [Puppet, 2025o].

- **Compatibilidad limitada del servidor principal**
  El servidor principal de Puppet, donde se centraliza todas las configuraciones, solo se puede instalar en algunas distribuciones principales de Linux. No es compatible con Windows ni macOS [Puppet, 2025r,s].

2.3.3. Chef

Visión General

Chef es una herramienta de gestión de configuración de código abierto diseñada para automatizar el aprovisionamiento y administración de la infraestructura. Fue desarrollada por Opscode, empresa que posteriormente se renombró como Chef Software. La herramienta se lanzó por primera vez en enero de 2009, lo que marcó un avance significativo en la transición hacia la Infraestructura como Código (IaC), al permitir a los usuarios definir configuraciones del sistema mediante código [Steinglass, 2010, Jacob, 2009,

Miller, 2013]. En septiembre de 2020, Chef Software fue adquirido por Progress Software, que continúa con el soporte de la plataforma bajo la marca Progress Chef [Crist, 2020].

Chef surgió para satisfacer las necesidades de las organizaciones que enfrentan una creciente complejidad y gran escala, ofreciendo un enfoque centrado en el desarrollo para la automatización de la infraestructura que cuenta con control de versiones, metodologías de pruebas y diseño modular para una configuración de sistema confiable y repetible [Arora, 2024, Diklic, 2020].

Chef utiliza un lenguaje específico de dominio basado en Ruby, que encapsula unidades reutilizables conocidas como “cookbooks” y “recipes”, para describir la lógica de configuración de sistema. Estos “recipes” permiten a los administradores expresar el estado requerido del sistema de forma imperativa y declarativa (en cierto grado), ofreciendo flexibilidad para muchos equipos de trabajo [Diklic, 2020].

A diferencia de herramientas basadas en un modelo “push” como Ansible, Chef opera con un modelo “pull”, donde los nodos son los responsables de solicitar al servidor el estado deseado y de ejecutar las acciones necesarias para alcanzarlo de forma independiente. Este modelo se adapta bien en entornos que manejan un gran volumen de nodos, sin embargo, requiere de la instalación de clientes en cada uno de ellos [Progress, 2025a].

Chef ha sido ampliamente adoptado en entornos empresariales debido a su escalabilidad, sólido soporte de la comunidad y un ecosistema de herramientas para la realización de pruebas antes de los despliegues [Progress, 2025a].

Arquitectura

Chef opera en una arquitectura cliente-servidor que permite la automatización escalable en diversos entornos de TI. Esta arquitectura se basa en tres componentes principales, como se puede observar en la Fig. 2.4.

- **Chef Workstation** es el entorno de desarrollo de Chef. Éste incluye todas las herramientas necesarias crear código de infraestructura utilizando el DSL basado en Ruby de Chef, administrar los “cookbooks”, definir políticas y pruebas de despliegues. Incluye herramientas para pruebas como Cookstyle, Test Kitchen, ChefSpec y Chef InSpec, como también una herramienta llamada Knife para la interacción con el “Chef Infra Server” [Progress, 2025a].

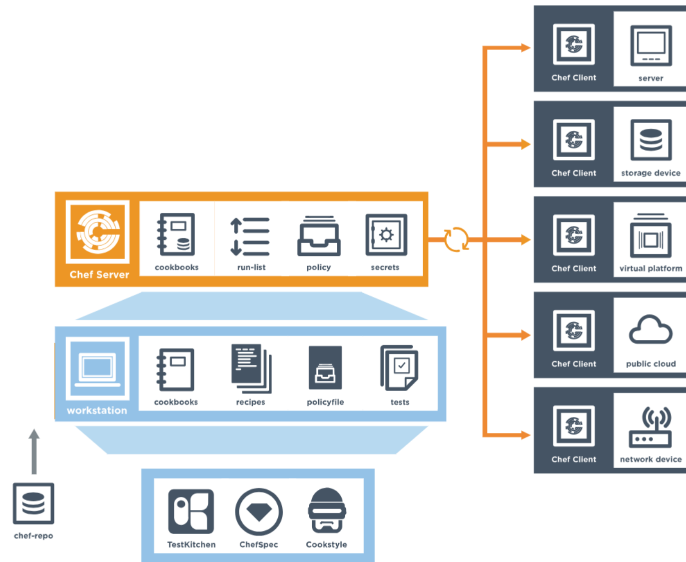

Figura 2.4: Chef - Arquitectura
Fuente: [Progress, 2025b]

- **Chef Infra Server** actúa como el almacén central y punto de distribución de los datos de configuración. Almacena los "cookbooks" y políticas aplicadas a los nodos, y la metadatos que describe cada nodo registrado bajo la administración de "Chef Infra Client". Cuando un cliente se conecta, se autentica mediante certificados SSL y recibe las últimas instrucciones de políticas y configuraciones. El servidor utiliza un control de accesos basado en roles (*Role-Based Access Control* - *RBAC*), que permite restringir el acceso a los objetos de Chef (nodos, ambientes, "cookbooks", etc) [Progress, 2025c,d,e].

- **Chef Infra Client** es un agente que se instala y se ejecuta periódicamente en cada uno de los nodos a administrar. Cuando se ejecuta, "Chef Infra Client" solicita las políticas de configuración a "Chef Infra Server" y las aplica para llevar al sistema al estado deseado. Esto se realiza de forma idempotente, de manera que el mismo "recipe" puede ser ejecutado múltiples veces obteniendo resultados consistentes [Progress, 2025f].

El flujo de la operación sigue estos pasos de manera general [Progress,

2025a]:

1. El administrador crea o modifica "cookbooks" en el "Chef Workstation".
2. Estos "cookbooks" son subidos al "Chef Infra Server" usando el comando `knife`.
3. El agente "Chef Infra Client" en cada uno de los nodos periódicamente realiza consultas al servidor.
4. Cada cliente obtiene los "cookbooks" relevantes para su operación y ejecuta la definición de recursos presentes para llevar al nodo al estado declarado. Este proceso es llamado "convergencia" [Progress, 2025f].
5. Después de la convergencia, el cliente reporta al servidor el éxito o fallo en la ejecución, incluyendo detalles de logs y recursos actualizados.

Chef utiliza, por defecto, un enfoque de orquestación "pull", donde los clientes solicitan sus propias configuraciones a un servidor, promoviendo la escalabilidad y la descentralización [Progress, 2025a].

La comunicación entre estos componentes está protegida mediante certificados SSL, lo que garantiza que solo nodos registrados y confiables puedan comunicarse con el servidor [Progress, 2025d].

DSL

Chef utiliza un lenguaje específico de dominio poderoso basado en Ruby, lo que permite a los usuarios definir configuraciones de sistema mediante estructuras de programación habituales como condicionales, bucles y variables. Este enfoque proporciona flexibilidad y expresividad, distinguiéndolo de otras herramientas que se basan únicamente en formatos YAML o JSON [Progress, 2025g].

Aunque el DSL de Chef se basa en la sintaxis de Ruby, incorpora recursos y estructuras personalizadas definidas por Chef. Estos recursos como `packages`, `users` y `services`, extienden el lenguaje Ruby con una semántica centrada en la configuración específica para la automatización de la infraestructura. El DSL está diseñado para ser expresivo, legible y orientado a tareas, lo que permite a los usuarios describir el estado deseado del sistema sin necesidad de conocimientos profundos de programación en Ruby [Progress, 2025g].

La unidad principal de configuración en Chef es el "recipe", un archivo que contiene declaraciones de recursos que especifican la configuración deseada

para un sistema. Estos “recipes” se agrupan en “cookbooks”, los cuales pueden incluir plantillas, atributos, librerías y metadatos para formar unidades de automatización modulares y reutilizables [Progress, 2025h].

Los “recipes” se escriben en archivos con extensión `.rb`, lo que muestra su sintaxis basada en Ruby. Sin embargo, es importante notar que, aunque los “recipes” son en realidad códigos escritos en Ruby, su comportamiento lo determina el entorno de ejecución de Chef y no un intérprete de Ruby de propósito general. [Progress, 2025h]. Un ejemplo de un “recipe” simple para instalar `nginx`, usando el recurso `package`, se puede observar en el Listado 3.

```
package 'nginx' do
  action :install
end
```

Listado 3: Ejemplo de un Chef Recipe

Además de los recursos por defecto, Chef permite la creación de recursos personalizados, lo que permite encapsular configuraciones complejas o específicas de una organización en componentes reutilizables. Esta extensibilidad hace que Chef sea adecuado para diversos entornos con diversas necesidades de infraestructura [Progress, 2025i].

Requisitos de sistema

Para operar correctamente, Chef requiere requisitos específicos para cada uno de sus tres componentes principales Chef Workstation, Chef Infra Server y Chef Infra Client.

Chef Workstation es el entorno usado por los administradores para crear los “cookbooks”, realizar pruebas e interactuar con *Chef Infra Servers*. Los principales requisitos para su instalación son [Progress, 2025o]:

- Sistema operativo: Compatible con Amazon Linux, macOS, Debian, Red Hat Enterprise Linux / CentOS, Ubuntu y Windows de 64 bits.
- Git: Aunque no es obligatorio, se recomienda tener Git instalado para el versionamiento y manejo de “cookbooks”.
- Hardware: Se recomienda mínimo 4 GB de RAM y 8 GB de espacio disponible para su instalación.

El componente *Chef Infra Server* es el encargado de almacenar y distribuir los “cookbooks”, políticas y manejar la autenticación. Sus requisitos son [Progress, 2025p]:

- Sistema operativo: Compatible con Amazon Linux, Red Hat Enterprise Linux, Rocky Linux, SUSE Linux Enterprise Server y Ubuntu. No es compatible con Windows ni con macOS.
- Hostname: Debe tener configurado un hostname de manera adecuada y debe ser FDQN (*Fully Qualified Domain Name*).
- NTP: Una conexión a NTP (*Network Time Protocol*) es requerido pues Chef Infra Server es sensible a desviaciones del reloj interno.
- Configuración de red: El servidor debe asegurarse de mantener los puertos 80 y 443 abiertos.
- Agente de correo local: Un agente de transferencia de correo local que permita al servidor enviar notificaciones por correo electrónico.
- cron: Cron debe estar instalado y habilitado, pues las tareas de mantenimiento periódicas son realizadas a través de él.
- git: Git debe estar instalado para que varios de los servicios internos puedan confirmar sus revisiones.
- Hardware: El CPU debe tener una arquitectura de 64 bits y un mínimo de 4 núcleos, mínimo 8 GB de RAM y 15 GB de espacio disponible.

El componente *Chef Infra Client* es instalado en cada uno de los nodos a controlar, y es el encargado de solicitar las últimas configuraciones al servidor central y aplicarlas en caso de que el estado actual del nodo no sea el deseado. Sus principales requisitos son [Progress, 2025r]:

- Sistema operativo: Compatible con AIX, Amazon Linux, CentOS, Debian, FreeBSD, macOS, Oracle Enterprise Linux, Red Hat Enterprise Linux, Rocky Linux, Solaris, SUSE Linux Enterprise Server, Ubuntu (LTS releases), Windows [Progress, 2025q].
- RAM: Durante ejecución de Chef Infra Client es recomendable tener disponible al menos 512 MB de memoria RAM.

Fortalezas

- **Madurez**
Chef tiene una presencia importante en el ecosistema de gestión de configuración y se integra bien con otras herramientas DevOps y pipelines de integración y entrega continua [Kumar, 2022].

- **Extensible y modular**
Chef permite a los usuarios crear sus propios recursos personalizados, extendiendo así la capacidad del sistema [Progress, 2025i]. Además, Chef organiza la lógica de configuración en unidades modulares llamadas "cookbooks", lo que facilita la reutilización, el mantenimiento y el control de versiones [Progress, 2025k].

- **Alta escalabilidad**
Chef puede gestionar grandes infraestructuras de forma eficiente, siempre que los componentes servidor y agentes estén dimensionados y optimizados adecuadamente. Su arquitectura le permite escalar a miles de nodos con recursos suficientes [Velimirovic, 2024].

- **DSL poderoso para control granular**
El uso de un DSL basado en Ruby permite a los usuarios avanzados definir una lógica de infraestructura altamente personalizada y compleja (soportando un modelo híbrido imperativo-declarativo). Esto resulta especialmente útil en entornos que requieren de un control granular del comportamiento de los recursos [Progress, 2025g,j].

- **Fuerte apoyo de la comunidad y empresas**
Chef cuenta con el respaldo de una gran comunidad de usuarios (Chef Supermarket alberga una vasta colección de "cookbooks" desarrollados por la comunidad) y ofrece un sólido soporte empresarial a través de *Enterprise Chef*, garantizando actualizaciones oportunas, software con soporte a largo plazo (*Long-Term Support - LTS*) e integración con proveedores de la nube [Progress, 2025t,s,l,n].

- **Idempotencia y Testabilidad**
Los recursos de Chef están diseñados para ser idempotentes, y la integración con herramientas como Test Kitchen y Chef InSpec permite a los usuarios probar el código de configuración antes del despliegue [Progress, 2025j,a].

- **Integración con proveedores de la nube**
Chef ofrece herramientas como plugins de Knife para la nube dentro

del Chef Workstation, para automatizar el aprovisionamiento en plataformas como AWS, Azure y Google Cloud [Progress, 2025l].

- **Capacidades de auditoría y compliance**
  Con Chef InSpec, los usuarios pueden definir perfiles para auditar y monitorear continuamente la infraestructura para garantizar el cumplimiento de políticas regulatorias o internas [Progress, 2025m].

Limitaciones

- **Curva de aprendizaje pronunciada**
  El DSL basado en Ruby, si bien es poderoso, requiere que los desarrolladores estén familiarizados con las convenciones de Ruby. Esto supone una barrera de aprendizaje para los equipos que aún no dominan este lenguaje [Progress, 2025g].

- **Instalación de agentes en cada nodo**
  Chef Infra Client debe ser instalado en cada uno de los nodos a administrar, lo que agrega una carga operativa, especialmente en entornos con muchos nodos efímeros o soporte de automatización limitado [Progress, 2025f].

- **Compatibilidad limitada del servidor principal**
  El componente *Chef Infra Server* solo es compatible con distribuciones de Linux, no se puede ejecutar en Windows ni en macOS, lo que limita su flexibilidad en entornos heterogéneos [Progress, 2025p].

2.3.4. Análisis comparativo de las herramientas

Después de analizar las herramientas Ansible, Puppet y Chef en las subsecciones anteriores, queda en evidencia que cada una de éstas presenta filosofías, arquitecturas y modelos operativos distintos. Si bien las tres buscan facilitar y automatizar la gestión y despliegue de procesos de configuración, sus decisiones de diseño influyen en su usabilidad, escalabilidad e idoneidad para diferentes entornos.

Ansible, por ejemplo, destaca por su simplicidad y arquitectura sin agentes. El uso de SSH para la comunicación y YAML para la definición de la configuración, permite a los usuarios definir tareas de automatización sin necesidad de conocimientos avanzados de programación. Esto reduce la barrera de entrada, permitiendo una integración y adopción más rápida en escenarios donde se prefiere una configuración sencilla y mínima del lado

de los nodos controlados. Sin embargo, su dependencia de Python en estos nodos, puede causar problemas de compatibilidad cuando faltan módulos necesarios para ciertas tareas o cuando no está disponible la versión de Python esperada. Además, Ansible requiere intervención manual para aplicar el estado deseado, lo que puede ser una limitación en infraestructuras más complejas.

Puppet, por otro lado, utiliza un modelo agente-servidor con un DSL declarativo que abstrae los detalles de ejecución. Su énfasis en la aplicación constante del estado deseado y la generación de reportes lo hace adecuado para entornos a gran escala con estrictas necesidades de cumplimiento. Sin embargo, el requerimiento de instalar y administrar agentes, sumado a la limitada compatibilidad de plataformas del servidor principal, puede presentar obstáculos en despliegues heterogéneos o nativas de la nube.

Chef ofrece una gran flexibilidad y control granular gracias a su DSL basado en Ruby. Proporciona un ecosistema robusto con herramientas adicionales para la realización de pruebas, auditorías y visualización de información. Sin embargo, su curva de aprendizaje es pronunciada, y al igual que Puppet, requiere la instalación de agentes en cada uno de sus nodos administrados.

La Tabla 2.1 muestra la comparación entre las características de cada herramienta.

Este resumen comparativo destaca las ventajas y desventajas de cada herramienta. La elección de la herramienta adecuada suele depender de las necesidades específicas del entorno, ya sea la simplicidad, compatibilidad o extensibilidad como prioridad. En este trabajo, estas comparaciones ayudan a ilustrar la motivación de diseñar e implementar un prototipo de herramienta que combine la usabilidad con mínimas dependencias en tiempo de ejecución.

2.4. Tecnologías relevantes

2.4.1. Lenguaje de Programación Go

Go (también conocido como Golang) es un lenguaje de código abierto desarrollado en Google, diseñado para ser simple, eficiente y con un soporte sólido para la concurrencia. Desde su lanzamiento en 2009, Go ha sido ampliamente adoptado para la programación a nivel de sistema, herramientas de infraestructura en la nube e interfaces de línea de comandos gracias a su rendimiento y facilidad de uso [The Go Authors, 2025a].

Una de las ventajas más atractivas de Go es su capacidad para producir binarios estáticos. Estos binarios contienen todas las dependencias necesarias, eliminando la necesidad de entornos de ejecución o intérpretes en el sistema destino. Esta propiedad hace que Go sea especialmente adecuado para ambientes donde instalaciones con mínima sobrecarga y máxima portabilidad son esenciales, lo que se alinea con el objetivo de desarrollar herramientas de automatización ligeras que eviten dependencias de intérpretes como Python o Ruby [The Go Authors, 2025a].

La compilación cruzada es otra ventaja de Go, pues permite a los desarrolladores compilar código para múltiples sistemas operativos y arquitecturas desde un único entorno de desarrollo. Esto lo hace ideal para el despliegue de herramientas de automatización en sistemas heterogéneos como Linux, Windows Server y macOS [The Go Authors, 2025b].

La librería estándar de Go es extensa e incluye un robusto soporte para la programación de redes, criptografía y procesamiento de texto. En particular, los paquetes `text/template` y `text/html` proporcionan un poderoso procesador de plantillas de texto que permiten construir lenguajes específico de dominio (DSL) flexibles en archivos de configuración. Esta capacidad se aprovecha en la herramienta propuesta en esta tesis, donde los archivos de configuración YAML se analizan y se procesan utilizando la sintaxis de plantillas de Go para inyectar lógica, variables y expresiones que faciliten la definición de las instrucciones de configuración [The Go Authors, 2025c].

Adicionalmente, el fuerte enfoque de Go en la simplicidad y la legibilidad, junto con rápida compilación y primitivas de concurrencia integradas (como `goroutines` y `channels`), lo hacen ideal crear interfaces de líneas de comandos escalables y con alta capacidad de respuesta. Su ecosistema dinámico y herramientas para la gestión de dependencias (`go mod`) y pruebas (`go test`), también contribuyen a un desarrollo eficiente y mantenible [The Go Authors, 2025a].

En general, la filosofía de diseño y las características técnicas de Go lo convierten en una opción natural para crear herramientas de automatización ligeras, portables y fáciles de mantener con el objetivo de mejorar la usabilidad sin depender de intérpretes ni entornos de ejecución especiales.

2.4.2. Formato YAML

YAML (*YAML Ain't Markup Language*) es un formato de serialización de datos, comúnmente utilizado para archivos de configuración e intercambio de datos entre lenguajes de programación. Está diseñado para ser intuitivo para

los humanos, manteniendo la característica de ser analizable por máquina, lo que lo convierte en una opción preferida por herramientas modernas de automatización y gestión de configuración como Ansible, Kubernetes y Docker Compose [YAML Language Development Team, 2021].

La principal ventaja de YAML reside en su simplicidad y legibilidad en comparación con formatos como JSON o XML. YAML utiliza sangría para representar sus estructuras, eliminando la necesidad de utilizar llaves o paréntesis angulares. Esto reduce el ruido sintáctico y facilita la creación, revisión y mantenimiento de los archivos de configuración, un factor importante cuando las configuraciones se gestionan de forma colaborativa [Amazon Web Services Inc., 2025].

YAML admite varios tipos de datos, incluyendo escalares, listas y mapas. También ofrece funciones avanzadas como "anchors" y "aliases" para reutilizar contenido y fusión de campos para combinar múltiples estructuras de datos. Estas características lo transforman en un formato potente, claro y reutilizable [YAML Language Development Team, 2021].

En el contexto del presente trabajo, YAML se utiliza como base para un lenguaje específico de dominio ligero. Este enfoque aprovecha la estructura declarativa de YAML a la vez que integra nuevas funcionalidades con ayuda del paquete `text/template` de Go, siendo posible la declaración de variables, bucles y lógica condicional dentro del DSL. Esta combinación mejora la flexibilidad, manteniendo la legibilidad y simpleza de YAML.

Además, la amplia adopción de YAML en un gran número de herramientas de automatización, implica que muchos usuarios ya están familiarizados con su sintaxis y estructura, lo que resulta en un aprendizaje más rápido, permitiéndoles ser productivos en menos tiempo. Su naturaleza de texto plano lo hace compatible con sistemas de control de versiones, facilitando el seguimiento de cambios y la gestión colaborativa.

2.4.3. Secure Shell - SSH

Secure Shell (*SSH*) es un protocolo de red criptográfico diseñado para operar servicios de forma segura en una red no segura. Se lo utiliza comúnmente para el inicio de sesión de forma remota por línea de comandos y para la ejecución de comandos. Reemplaza protocolo antiguos no seguros como "Telnet" y "rlogin", proporcionando una autenticación robusta, confidencialidad e integridad a través de la encriptación de datos [Lonvick and Ylonen, 2006].

En el contexto de las herramientas de gestión y automatización de la configuración, SSH desempeña un papel fundamental al permitir una

comunicación segura con los nodos controlados sin el requisito de agentes preinstalados. Al usar SSH, los sistemas pueden pueden ejecutar comandos, transferir archivos y orquestar tareas sin la necesidad de agentes o entornos de ejecución especializados en los nodos remotos. Esto reduce la sobrecarga operacional inicial y simplifica el despliegue en infraestructuras heterogéneas [OpenSSH Team, 2025].

La autenticación usando una llave pública de SSH permite conexiones seguras sin contraseña, lo que mejora la escalabilidad al gestionar un gran número de nodos. Este enfoque no solo reduce la carga administrativa de la gestión de credenciales, sino que también minimiza el riesgo de ataques de fuerza bruta, mejorando así la seguridad general del sistema [Painter Lee, 2023, Portnox, 2025].

En el caso de herramientas escritas en Go, como la propuesta en esta tesis, es posible aprovechar el paquete oficial `golang.org/x/crypto/ssh` para implementar una funcionalidad nativa de cliente SSH o invocar directamente el ejecutable de OpenSSH instalado en el sistema. Ambas opciones permiten la creación segura de sesiones, la ejecución de comandos y transferencia de archivos sin requerir de agentes adicionales [The Go Authors, 2025d].

El papel consolidado del protocolo SSH en la administración de sistemas, ha provocado que la mayoría de los sistemas tipo Unix incluyan un cliente SSH por defecto, reduciendo así significativamente el tiempo de configuración inicial, haciéndolo ideal para su uso en herramientas de gestión de configuración [SSH, 2025].

Tabla 2.1: Comparación entre herramientas CM

Característica	**Ansible**	**Puppet**	**Chef**
Arquitectura	Sin agentes, modelo push	Basado en agentes, modelo pull	Basado en agentes, modelo pull
DSL	YAML	Puppet DSL	DSL basado en Ruby
Aplicación de estado	Manual	Automático	Automático
Compatibilidad Servidor	Linux / macOS / Windows	Sólo Linux	Sólo Linux
Compatibilidad nodos clientes	Linux / macOS / Windows	Linux / macOS / Windows	Linux / macOS / Windows
Escalabilidad	Moderada	Alta	Alta
Curva de aprendizaje	Baja	Media	Alta
Formato de configuración	.yaml	.pp	.rb
Protocolo de comunicación	SSH	HTTPS/SSL	HTTPS vía Chef Infra Server
Dependencias en nodos	Python	Puppet Agent	Chef Infra Client

Capítulo 3

Prototipo: Diseño e implementación

Este capítulo presenta el diseño y la implementación de `Shoal`, un prototipo de herramienta desarrollada para ejecutar tareas de configuración y despliegue en los nodos controlados utilizando un modelo de comunicación cliente-servidor sin dependencias de intérpretes externos. Este capítulo comienza definiendo los requisitos funcionales y no funcionales del sistema, seguido de un análisis de trazabilidad que los relaciona con los criterios de diseño y objetivos de la herramienta. La arquitectura propuesta es descrita en detalle, enfatizando su estructura modular, modelo de concurrencia, y la separación de responsabilidades entre componentes principales. Se da especial atención a cómo `Shoal` logra conseguir operación sin agentes independiente de la plataforma a través de una comunicación basada en SSH y configuraciones en YAML.

Las subsiguientes secciones se centran en el DSL basado en YAML que define el flujo de trabajo de automatización, el sistema de módulos extensible y la interfaz de línea de comandos que proporciona un acceso unificado a las funcionalidades principales de `Shoal`. Este capítulo concluye con la sección de validación y resultados, la cual demuestra la funcionalidad del prototipo a través de ejecuciones prácticas, capturas de pantalla, y verificación de requisitos.

Todos estos componentes juntos demuestran cómo `Shoal` integra las fortalezas de Go (binarios estáticos, concurrencia y plantillas) para entregar una plataforma de automatización ligera y extensible alineada con los objetivos del proyecto.

3.1. Requisitos y criterios de diseño

El prototipo desarrollado, llamado `Shoal`, fue diseñado para automatizar tareas de sistema y despliegues de aplicaciones usando un modelo cliente-servidor que evita dependencias en intérpretes externos como Python o Ruby, en los nodos administrados. A diferencia de herramientas tradicionales como Ansible, Puppet o Chef, `Shoal` está implementado en Go, permitiendo la generación de binarios compilados ligeros capaces de ejecutarse nativamente en diferentes sistemas operativos sin requerir entornos de ejecución adicionales y/o especiales.

El diseño de `Shoal` sigue una arquitectura hexagonal, la cual separa la lógica de dominio de las dependencias de infraestructura, asegurando flexibilidad y testabilidad. Esta estructura permite que diferentes componentes puedan ser reemplazados o extendidos de manera independiente sin modificar la lógica central de funcionamiento.

3.1.1. Requisitos funcionales

- **RF1 - Ejecución remota de tareas vía SSH**
  El controlador debe ejecutar un conjunto de tareas predefinidas en uno o más nodos destino a través de conexiones SSH.

- **RF2 - Transferencia de artefactos**
  El prototipo debe soportar la transferencia de archivos o binarios a los nodos administrados usando la misma conexión SSH.

- **RF3 - Interpretar archivos de configuración en YAML**
  El controlador debe leer, validar e interpretar los archivos de configuración (llamados `swimbooks`) escritos en YAML, los cuales definen variables y las tareas a ejecutar.

- **RF4 - Sustitución de variables y plantillas**
  La herramienta debe soportar sustitución dinámica de variables y plantillas para la interpolación de valores dentro de los archivos de configuración.

3.1.2. Requisitos no funcionales

- **RNF1 - Operación sin agentes**
  Shoal debe ejecutar todas las tareas sin la instalación de agentes persistentes o entornos de ejecución especiales en los nodos administrados.

- **RNF2 - Independencia de intérpretes externos**
  Los nodos controlados no deben requerir intérpretes externos como Python o Ruby. Todas las operaciones deben ejecutarse usando SSH.

- **RNF3 - Compatibilidad multiplataforma**
  Shoal debe administrar nodos remotos basados en Debian Linux y Windows Server, mientras el controlador debe operar en sistemas Debian Linux o macOS.

- **RNF4 - Seguridad**
  Las conexiones SSH deben soportar autenticación por clave y por llave pública. Variables sensibles y credenciales no deben ser guardados ni registradas en texto plano.

- **RNF5 - Rendimiento y escalabilidad (orientado al diseño)**
  La arquitectura debe soportar la ejecución de tareas concurrentes y la habilidad de manejar múltiples nodos eficientemente. Esta característica se aborda a nivel arquitectónico.

- **RNF6 - Usabilidad**
  La sintaxis de configuración debe ser simple, de fácil lectura, y basada en YAML para reducir la curva de aprendizaje de usuarios familiarizados con herramientas similares.

- **RNF7 - Extensibilidad y modularidad**
  La arquitectura debe permite a desarrolladores extender la funcionalidad del prototipo a través de componentes modulares como nuevas implementaciones de ejecutores, nuevas formas de transporte y módulos reusables.

3.1.3. Criterios de diseño

Para la implementación, se establecieron los siguientes criterios de diseño:

- **Arquitectura sin agentes vía SSH:** `Shoal` usa SSH para comunicarse directamente con los nodos destino, eliminando la necesidad de instalar agentes.

- **Arquitectura hexagonal:** La lógica de dominio central está aislada de adaptadores específicos dependientes de infraestructura, facilitando la compatibilidad multi-plataforma y futuras extensiones.

- **Abstracción de la forma de ejecución:** `Shoal` define una interfaz unificada llamada `Executor`, la cual abstrae los detalles de ejecución de tareas, permitiendo delegar las operaciones a un `LocalExecutor` (para tareas en el controlador) y a un `RemoteExecutor` (para tareas en los nodos remotos), desacoplando así la lógica de orquestación de la lógica de ejecución. Mediante esta separación de responsabilidades, la arquitectura permanece extensible y eficiente aún si el número de nodos incrementa.

- **DSL basado en YAML con plantillas:** Los archivos de configuración YAML son declarativos, de fácil lectura y enriquecidos con el sistema de plantillas de Go.

- **CLI implementada con Cobra:** La interfaz de línea de comandos está construida con la librería Cobra (misma que es utilizada en herramientas populares como `kubectl`), asegurando la usabilidad, consistencia y fácil integración de nuevos comandos.

La relación entre los objetivos del trabajo, los requisitos funcionales y no funcionales, y los criterios de diseño, se puede observar en las Fig. 3.1 y Fig. 3.2.

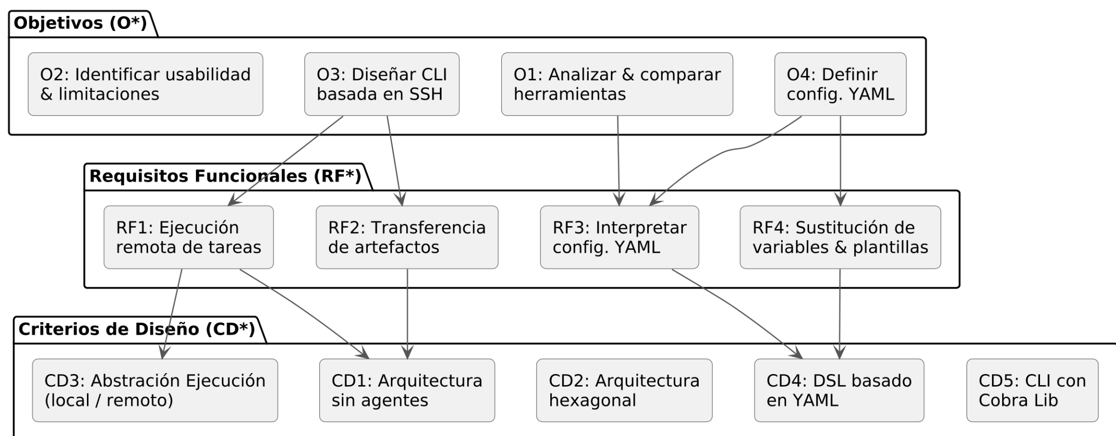

Figura 3.1: Trazabilidad (Objetivos - Req. Funcionales - Criterios de diseño)

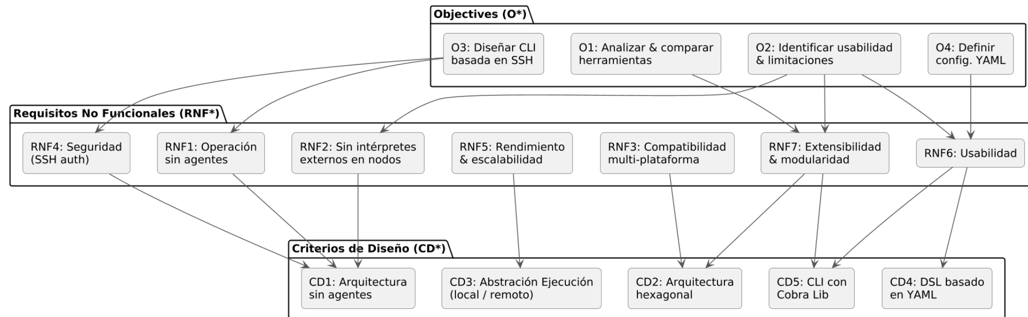

Figura 3.2: Trazabilidad (Objetivos - Req. No Funcionales - Criterios de diseño)

3.2. Arquitectura del sistema

`Shoal` sigue un arquitectura hexagonal (puertos y adaptadores) para conseguir una separación de responsabilidades entre la lógica de dominio y dependencias externas como la lectura de archivos de configuración, reportes y comunicación con los nodos remotos.

Este enfoque promueve la modularidad, testabilidad y extensibilidad, garantizando que nuevas funcionalidades o implementaciones puedan ser añadidas sin alterar la lógica de orquestación central del sistema.

En el centro del sistema se encuentra el **Engine**, cuyo método principal *RunSwimbook* coordina todos los aspectos de una ejecución de automatización: sustitución de variables y plantillas, enrutamiento de tareas y manejo de ejecuciones concurrentes en los nodos.

La capa CLI prepara el entorno de ejecución, cargando las configuraciones de tareas e inventarios de hosts, para luego delegar la ejecución al `Engine`.

Esta clara división de responsabilidades asegura que el `Engine` permanezca desacoplado de las responsabilidades de E/S y formatos de archivos externos.

3.2.1. Vista General del Componente

La arquitectura de `Shoal` divide las responsabilidades entre la lógica de dominio central, puertos (interfaces) y adaptadores (implementaciones).

Un **Swimbook** es el archivo de configuración declarativo basado en YAML que define variables y un grupo ordenado de tareas.

El término *Swimbook* se alinea con la metáfora marina del proyecto: de la misma manera que *shoal* representa un grupo coordinado de peces, un *swimbook* simboliza el conjunto sincronizado de acciones que guían su movimiento. Una descripción más detallada de su estructura y esquema YAML se presenta en la Sección 3.3.

Los diferentes componentes del sistema se pueden visualizar en la Tabla 3.1 y en la Fig. 3.3:

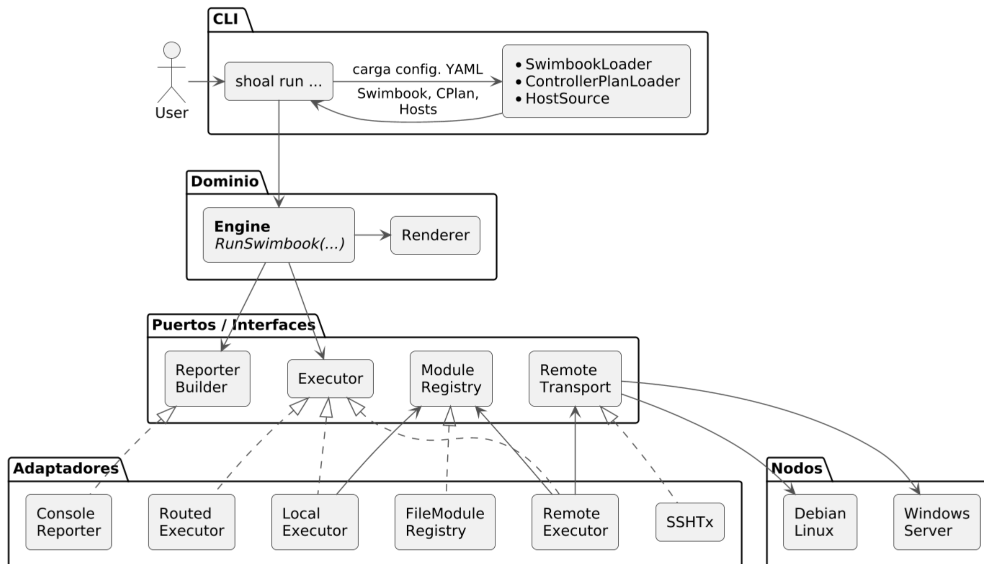

Figura 3.3: Shoal - Arquitectura (Componentes)

Las interacciones entre estos componentes durante la ejecución se describen en la siguiente subsección.

3.2.2. Flujo de Ejecución Central

El punto de entrada principal de la orquestación en Shoal es:

```
func RunSwimbook(ctx context.Context, sb *Swimbook, cp *ControllerPlan,
                 hosts []Host) error
```

Listado 4: Método RunSwimbook

Tabla 3.1: Componentes de Shoal

Componente	**Tipo**	**Rol**	**Implementación**
Engine	Núcleo	Orquestador central que controla la ejecución de tareas y la concurrencia	-
Executor	Interfaz (Puerto)	Define el contrato de ejecución de tareas	`LocalExecutor`, `RemoteExecutor`, `RoutedExecutor`
RemoteTransport	Interfaz (Puerto)	Maneja la comunicación remota con los nodos	`SSHTx`
ModuleRegistry	Interfaz (Puerto)	Provee el acceso a los módulos disponibles y sus metadatos	`FileModuleRegistry`
Renderer	Comp. concreto	Sustituye variables en plantillas usando `text/template` de Go	-
ReporterBuilder	Interfaz (Puerto)	Genera reportes de ejecución al usuario	`ConsoleReporter`
SwimbookLoader	Interfaz (Puerto)	Analizador de swimbooks (invocado por la CLI)	`SwimYAMLLoader`
ControllerPlan Loader	Interfaz (Puerto)	Analizador de configuraciones a nivel de controlador (invocado por la CLI)	`CPlanYAMLLoader`
HostSource	Interfaz (Puerto)	Resuelve inventarios de hosts y detalles de conexión (invocado por la CLI)	`HostYAMLSource`

Este método ejecuta todas las tareas definidas en un *Swimbook* en los hosts remoto especificados, como también las tareas a nivel de controlador definidas en el `ControllerPlan`.

Etapas de ejecución

1. **Obtención de datos de entrada:**
   La capa CLI instancia las implementaciones basadas en YAML de `SwimbookLoader`, `ControllerPlanLoader` y `HostSource`. Luego, llama los respectivos métodos para obtener todos los datos de entrada requeridos para la ejecución. Los objetos resultantes (`Swimbook`, `ControllerPlan` y `[]Host`) son, posteriormente, pasados al método `RunSwimbook` del `Engine`, el cual comienza la orquestación.

2. **Sustitución de variables:**
   Una vez proporcionado los datos de entrada, el componente `Renderer` fusiona e interpola variables de los archivos de configuración utilizando el paquete `text/template` de Go. Esto asegura que todas las expresiones de estos archivos sean valores concretos antes de la ejecución.

3. **Enrutamiento de tareas y ejecución:**
   El `Engine` itera sobre cada tarea definida en el `Swimbook` y determina cómo debe ejecutarse basado en el contexto:

   - **LocalExecutor:** Ejecuta tareas en el mismo controlador (por ejemplo, preparación de los artefactos de despliegue).
   - **RemoteExecutor:** Ejecuta tareas en los nodos controlados a través de la interfaz `RemoteTransport` (el adaptador por defecto, `SSHTx`, utiliza SSH para la comunicación).
   - **RoutedExecutor:** Actúa como una fachada que delega a ejecutores Local o Remoto dependiendo del tipo de host objetivo.

   El `Engine` es el responsable de la concurrencia: crea `goroutines` para cada host remoto, limitadas por un límite de concurrencia configurable. Los ejecutores permanecen sin estado y se centran únicamente en la ejecución de la tarea asignada.

4. **Generación de informe:**
   Una vez finalizada cada tarea, el `Engine` agrega los resultados usando la interfaz `ReportBuilder`. El adaptador por defecto, `ConsoleReporter`,

construye y presenta reportes que contiene registros de salida, códigos de error y un resumen de ejecución por host.

3.2.3. Dinámica de la Ejecución

1. La CLI invoca el método `RunSwimbook` del `Engine`, pasando el contexto, el `Swimbook` y `ControllerPlan` cargados, y la lista de hosts remotos que se van a afectar.

2. El `Engine` realiza la sustitución de variables usando el `Renderer` e itera sobre cada tarea de forma secuencial como se encuentren definidas en el `Swimbook`.

3. Cada tarea es enrutada por el `Engine` hacia su correspondiente implementación `Executor` a través del `RoutedExecutor`.

4. Para las tareas remotas, el `Engine` genera `goroutines` para cada host objetivo. El `RemoteExecutor` llama al `RemoteTransport` (ej. SSH) configurado para ejecutar comandos y transferir artefactos.

5. El `Engine` agrega los resultados y los envía al `ReporterBuilder` para que se produzca la salida formateada que visualiza el usuario.

La Fig. 3.4 muestra el diagrama de secuencia durante la ejecución del comando principal `shoal run`.

Por otro lado, la Fig. 3.5 resume la estructura de los elementos principales de `Shoal`, resaltando como el `Engine` depende únicamente de interfaces (`Executor` y `ReporterBuilder`) y una implementación concreta de `Renderer`, mientras la CLI instancia los analizadores y el inventario de hosts antes de invocar el método `RunSwimbook`. El diagrama también muestra que `RoutedExecutor`, `LocalExecutor` y `RemoteExecutor` implementan la interfaz `Executor`, con `RemoteExecutor` dependiendo de un `RemoteTransport` y ambos ejecutores resolviendo módulos a través de `ModuleRegistry`.

3.2.4. Concurrencia y Separación de Responsabilidades

La concurrencia es completamente controlada por el `Engine`, el cual genera y sincroniza `goroutines` para cada host remoto.
Los `Executors` permanecen sin estado y se centran únicamente en el *cómo* de la ejecución. Esta separación garantiza la escalabilidad, mientras se mantiene la lógica de ejecución simple y aislada. El modelo de concurrencia puede

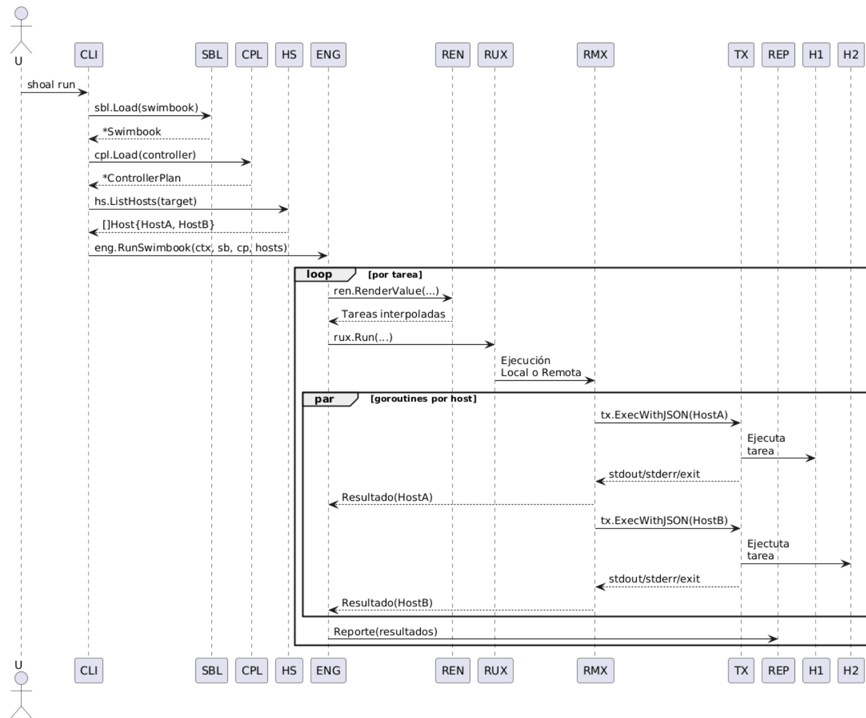

Figura 3.4: Diagrama de secuencia (`shoal run`)

ser ajustado usando el parámetro `max_parallel_hosts` especificado en el `Swimbook`.

3.2.5. Extensibilidad y Mantenibilidad

Cada adaptador o implementación es reemplazable de forma independiente. Por ejemplo:

- Una nueva implementación de `RemoteTransport` (ej. WinRM) puede ser añadida sin modificar el `Engine`.
- `Reporters` adicionales (ej. JSONReporter) pueden ser introducidos implementando la interfaz `ReporterBuilder`.
- Analizadores de configuración personalizados (ej. JSONLoader) pueden extender la funcionalidad de Shoal, manteniendo la misma lógica de orquestación central.

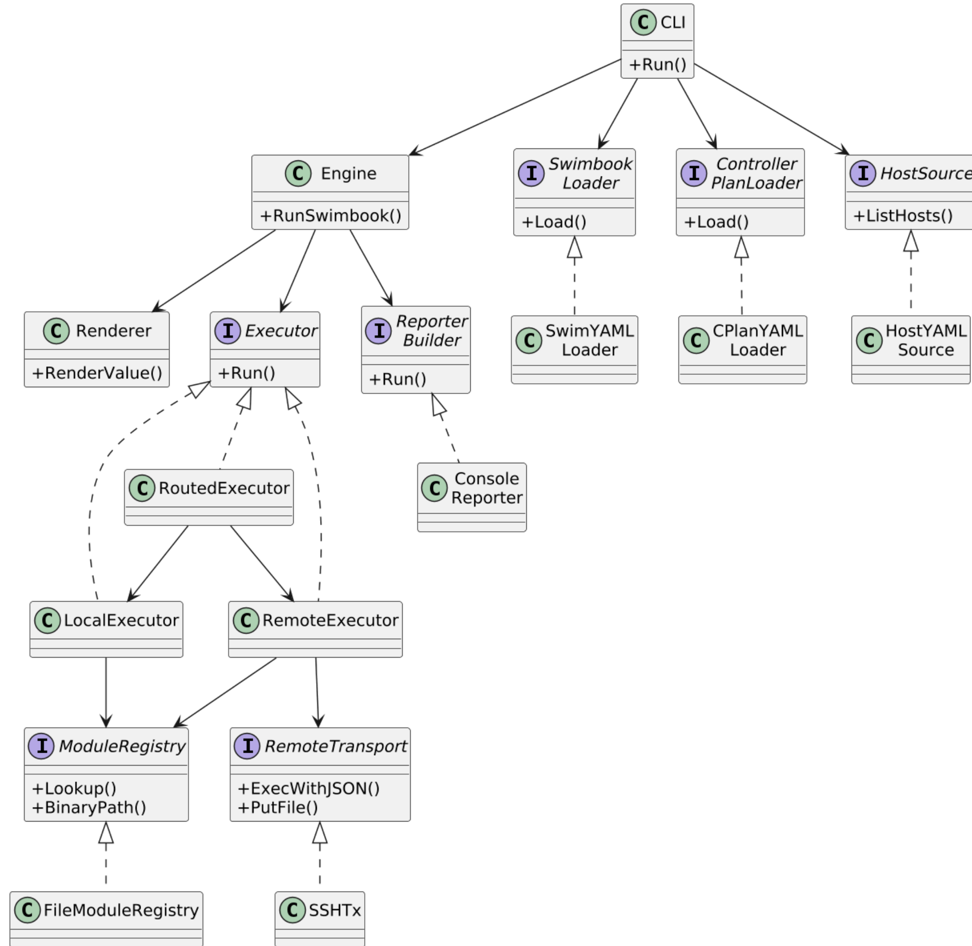

Figura 3.5: Shoal - Diagrama de clases

Este diseño modular simplifica las pruebas y la evolución futura de la herramienta, manteniendo las dependencias aisladas y las interfaces estables.

3.2.6. Consideraciones de Seguridad

- **Ejecución sin agentes:** No se instalan agentes persistentes de software en los nodos controlados.
- **Comunicación encriptada:** El adaptador `SSHTx` por defecto, use

canales SSH seguros para todas las operaciones remotas.

- **Autenticación:** Se soporta la autenticación basada en clave como por llave pública.
- **Manejo de secretos:** Los datos sensibles son sustituidos sólo en memoria y nunca son almacenados ni registrados en la salida del reporte.
- **Escalada de privilegios:** Algunas tareas pueden especificar privilegios elevados, las cuales `Shoal` ejecuta de manera segura utilizando los parámetros `become` y `become_user`.

3.3. Archivos de Configuración y DSL

`Shoal` define su comportamiento a través de archivos de configuración YAML, formando un Lenguaje de Dominio Específico (DSL) ligero que expresa tanto las tareas a ejecutar como la estrategia de ejecución.

Tres tipos de archivos principales componen este DSL:

1. **Swimbook:** Define *qué* ejecutar (las tareas, variables y estrategia de concurrencia).
2. **Controller Plan:** Define las tareas de lado del controlador antes y/o después de las tareas remotas.
3. **Hosts:** Define donde las tareas se ejecutaron (inventario de hosts).

Esta estructura se asemeja a la de sistemas de gestión de configuración establecidos, como los `playbooks` e inventarios de Ansible.

Cada archivo es validado de forma independiente y cargado a través de interfaces específicas (`SwimbookLoader`, `ControllerPlanLoader` y `HostSource`) instanciadas por el CLI antes de invocar al `Engine`.

3.3.1. Archivo Swimbook

Swimbook es el archivo de configuración principal que describe la secuencia de operaciones (`steps`) a ejecutar en uno o más hosts objetivos.

Este archivo puede referenciar un `ControllerPlan` (para tareas de pre-ejecución y pos-ejecución), define variables reusables, y configura la concurrencia y la tolerancia de error a través del campo `strategy`.

La sintaxis de `Swimbook` forma la capa central del DSL de `Shoal`.

```
name: string # nombre del swimbook
controller:
    file: string # path del archivo ControllerPlan (relativo al Swimbook)
vars: map[string]any # variables literales (no se interpolan)
steps:
    - name: string # nombre del paso/tarea
      if: bool | template # (opcional) condición si el paso se debe ejecutar
                          # o no (por defecto: true)
      module: string # nombre del módulo a ejecutar
      args: map[string](any | template) # argumentos de entrada del módulo
      bind_output: string # (opcional) nombre de variable donde se almacena
                          # las variables de salida del módulo
      become: boolean | template # true para tener permisos elevados
                                 # (por defecto: false)
      become_user: string | template # usuario para permisos elevados
      become_pass: string | template # (opcional) contraseña del usuario
                                     # para permisos elevados
strategy:
    max_parallel_hosts: int # (opcional) cantidad de hosts a procesar de
                            # forma paralela (por defecto: 5)
    continue_on_error: boolean # (opcional) true para continuar incluso si
                               # se presenta un error en algún paso
```

Listado 5: Swimbook - YAML Schema

El Listado 5 muestra el esquema YAML del archivo y el Listado 6 presenta un ejemplo de `Swimbook`.

3.3.2. Archivo ControllerPlan

`ControllerPlan` especifica los pasos que deben ejecutarse en el controlador, ya sea antes o después de las tareas remotas descritas en un `Swimbook`.

Casos comunes de uso incluyen la preparación de artefactos de despliegues y validaciones de tareas.

El Listado 7 muestra el esquema YAML del archivo y el Listado 8 presenta un ejemplo de `ControllerPlan`.

```yaml
name: "Deploy site"
controller:
    file: "./controller.yaml"
vars:
    app: "shoal"
    release_tag: "r42"
strategy:
    max_parallel_hosts: 5
    continue_on_error: false
steps:
    - name: "Ensure release directory"
      module: file.manage
      args:
        path: "/var/www/{{ .Vars.app }}/releases/{{ .Vars.release_tag }}"
        type: dir
        state: present
        mode: "0755"
    - name: "Push bundle to remote host"
      module: transfer.push
      args:
        src: "./artifacts/site-{{ .Vars.release_tag }}.zip"
        dest: "/tmp/site-{{ .Vars.release_tag }}.zip"
    - name: "Unzip into release directory"
      module: shell.posix
      args:
        command: >
          unzip -o /tmp/site-{{ .Vars.release_tag }}.zip -d
          /var/www/{{ .Vars.app }}/releases/{{ .Vars.release_tag }}
    - name: "Remove temporary file"
      module: file.manage
      args:
        path: "/tmp/site-{{ .Vars.release_tag }}.zip"
        state: absent
    - name: "Restart web service"
      module: shell.posix
      args:
        command: "systemctl restart nginx"
      become: true
      become: root
```

Listado 6: Swimbook - Ejemplo

```yaml
before: # definen las tareas a ejecutarse en el controlador antes de las
        # tareas remotas
    - name: string # nombre del paso/tarea
      if: bool | template # (opcional) condición si el paso se debe ejecutar
                          # o no (por defecto: true)
      module: string # nombre del módulo a ejecutar
      args: map[string](any | template) # argumentos de entrada del módulo
      bind_output: string # (opcional) nombre de variable donde se almacena
                          # las variables de salida del módulo
      become: boolean | template # true para tener permisos elevados
                                 # (por defecto: false)
      become_user: string | template # usuario para permisos elevados
      become_pass: string | template # (opcional) contraseña del usuario
                                     # para permisos elevados
after: # definen las tareas a ejecutarse en el controlador después de las
       # tareas remotas
    - name: string # nombre del paso/tarea
      if: bool | template # (opcional) condición si el paso se debe ejecutar
                          # o no (por defecto: true)
      module: string # nombre del módulo a ejecutar
      args: map[string](any | template) # argumentos de entrada del módulo
      bind_output: string # (opcional) nombre de variable donde se almacena
                          # las variables de salida del módulo
      become: boolean | template # true para tener permisos elevados
                                 # (por defecto: false)
      become_user: string | template # usuario para permisos elevados
      become_pass: string | template # (opcional) contraseña del usuario
                                     # para permisos elevados
```

Listado 7: ControllerPlan - YAML Schema

```yaml
before:
    - name: "Prepare artifacts directory"
      module: file.manage
      args:
        path: "./artifacts"
        state: present
        mode: "0755"
    - name: "Package web assets"
      module: shell.posix
      args:
        command: >
            zip -r ./artifacts/site-{{ .Vars.release_tag }}.zip ./dist
after:
    - name: "Pull logs from remote"
      module: transfer.pull
      args:
        src: "/var/log/nginx/error.log"
        dest: "./logs/{{ .Host.Name }}-error.log"
```

Listado 8: ControllerPlan - Ejemplo

3.3.3. Archivo Hosts

El archivo `hosts.yaml` define los hosts objetivos y los parámetros de conexión por defecto.

Los hosts objetivo son seleccionados en tiempo de ejecución a través de la CLI (`--groups`, `--hosts`), en vez de ser declarados dentro del `Swimbook`.

Esta separación asegura la reutilización de `Swimbooks` a través de diferentes entornos.

El Listado 9 muestra el esquema YAML del archivo y el Listado 10 presenta un ejemplo de `hosts.yaml`.

Reglas

- Cada host hereda campos no especificados del campo `defaults`.
- Los grupos deben referenciar nombres de hosts existentes.
- La CLI determina cuales hosts o grupos están activos durante la ejecución de un `Swimbook`.

3.3.4. Sustitución de variables

Una vez que los archivos `Swimbook`, `ControllerPlan` y `Hosts` son cargados, el `Engine` de `Shoal` prepara el contexto de ejecución de cada host.

Este contexto combina la configuración estática con datos en tiempos de ejecución, para ser luego pasados al `Renderer`, el cual interpola las plantillas en los campos permitidos de cada paso.

El `Renderer` usa el paquete `text/template` de Go, asegurando un modelo determinístico y tipado.

A diferencia de sistemas de plantillas más permisivos como Jinja2 usado en Ansible, el paquete `text/template` de Go asegura reglas de evaluación estrictas. Variables no definidas o expresiones malformadas generan errores explícitos en vez de silenciosamente resolver a un `string` vacío. Esto garantiza que todas las sustituciones sean validadas contra el contexto proporcionado antes de la ejecución. En consecuencia, el uso de las plantillas de Go aumenta la confiabilidad y la trazabilidad en el procesamiento de las configuraciones.

```
defaults:
    os_arch: string # OSArch común entre hosts
    env: map[string]string # variables de entorno común entre hosts
    address: string # dirección IP común entre hosts
    user: string # usuario de conexión común entre hosts
    password: string # (opcional) contraseña de usuario de conexión común
                     # entre hosts
    ssh_port: string # (opcional) puerto ssh común entre hosts
                     # (por defecto: 22)
    ssh_private_key_file: string # (opcional) llave para autenticación
                                 # común entre hosts
hosts:
    - name: string # nombre del host
      os_arch: string # <OperatingSystem>/<Architecture> - ej. linux/amd64
      env: map[string]string # variables de entorno específicas del host
      address: string # dirección IP del host
      user: string # usuario de conexión
      password: string # (opcional) contraseña de usuario de conexión
      ssh_port: string # (opcional) puerto ssh (por defecto: 22)
      ssh_private_key_file: string # (opcional) llave para autenticación

groups: map[string][]string # agrupa los hosts por grupos
```

Listado 9: Hosts - YAML Schema

```yaml
defaults:
    os_arch: "linux/amd64"
    env:
        API_URL: myapi.com
    user: deploy
    ssh_port: 22
    ssh_private_key_file: ~/.ssh/id_rsa
hosts:
    - name: "web-1"
      address: "10.0.1.11"
    - name: "web-2"
      address: "10.0.1.12"
groups:
    web: ["web-1", "web-2"]
```

Listado 10: Hosts - Ejemplo

Reglas de sustitución de variables

`Shoal` interpola las plantillas que se encuentran únicamente en los campos de `steps` que se muestran en la Tabla 3.2 (los `steps` se definen tanto en el `Swimbook` como en el `ControllerPlan`).

Todos los demás campos (como `name`, `module`, `bind_output`) permanecen de manera literal y no están sujetos a una sustitución dinámica.

Variables y su Sintaxis de Acceso

Las plantillas pueden acceder a tres fuentes de datos principales en tiempo de ejecución, como se puede observar en la Tabla 3.3.

Notar que la definición de variables es evaluada como literales en tiempo de carga. Luego, en tiempo de ejecución, el módulo `vars.set` puede modificar estas variables por host, lo que implica que actualizaciones de variables aplican únicamente al mapa de variables del host que se está ejecutando y no se propaga a las demás ejecuciones paralelas de hosts.

Tabla 3.2: Campos que admiten interpolación

Campo	**Descripción**
`if`	Condición lógica que controla si el `step` se ejecuta o no. Debe resolver un valor booleano después de la interpolación.
`args`	Argumentos que se pasan al módulo. Las plantillas dentro de los valores se expanden y se resuelven.
`become`	Expresión booleana que determina la escalada de privilegios.
`become_user`	Nombre de usuario para la escalada de privilegios.
`become_pass`	Contraseña opcional para la escalada de privilegios.

Tabla 3.3: Variables - Sintaxis de Acceso

Fuente	**Sintaxis**	**Ejemplo**	**Descripción**
Variables Swimbook	`{{.Vars.<name>}}`	`{{.Vars.app}}`	Valores literales definidos en el bloque `vars` de un `Swimbook` o mutada dinámicamente con el módulo `vars.set`.
Variables Entorno	`{{.Env.<name>}}`	`{{.Env.API_URL}}`	Valores proporcionados del entorno del sistema o por CLI.
Host Metadata	`{{.Host.<field>}}`	`{{.Host.Address}}`	Metadatos para el host objetivo actual.

```
steps:
  - name: "Define service name per OS"
    module: vars.set
    args:
      key: "svc"
      value: "{{ if eq .Host.os_arch `linux/amd64` }}myapp_64{{ else }}myapp{{ end }}"

  - name: "Restart computed service"
    module: shell.posix
    args:
      cmd: "systemctl restart {{ .Vars.svc }}"
    become: true
```

Listado 11: Mutación de variables y contexto de host aislado - Ejemplo

Precedencia de Variables

Cuando se realiza la sustitución de variables en un `step`, Shoal construye un contexto por host con los valores en el siguiente orden (de menor a mayor prioridad):

1. Variables de entorno globales (.Env).
2. Variables definidas en `Swimbook` (.Vars).
3. Variables de entorno específicas del host (.Env).
4. Propiedades del host (.Host).
5. Las variables `bind_output` de los pasos completados (.Vars).

Este enfoque permite que las definiciones de menor prioridad sean sobre-escritas de manera segura sin perder la claridad y reproducibilidad.

Por ejemplo, cuando el `Swimbook` del Listado 11 es ejecutado en múltiples hosts en paralelo:

- Cada host obtiene su propio contexto de ejecución con un mapa independiente de `.Vars` y `.Host`.
- El módulo `vars.set` modifica `.Vars.svc` para ese único host.
- Los contextos de los demás hosts permanecen sin cambios.

Este diseño de variables con alcance de host permite conseguir una ejecución de tareas en paralelo sin condiciones de carrera, garantizando resultados determinísticos y reproducibles a través de entornos heterogéneos.

3.4. Sistema de Módulos y Extensibilidad

El modelo de ejecución de `Shoal` está centrado alrededor de módulos, los cuales son componentes autocontenidos que realizan acciones concretas cuando son referenciados en un `Swimbook` o `ControllerPlan`.

Los módulos proporcionan el vocabulario funcional del DSL, permitiendo a los usuarios componer operaciones complejas (ej.: manipulación de archivos, instalación de paquetes) de forma declarativa a través de YAML.

A diferencia de los sistemas de plugins tradicionales que dependen de una interfaz a nivel de lenguaje, `Shoal` expone una API pública de Módulo que simplifica la creación de módulos, manteniendo al sistema de módulos agnóstico a un lenguaje y libre de dependencias.

Esta API define estructuras auxiliares y funciones para estandarizar la comunicación entre el `Engine` y un módulo externo.

3.4.1. API Pública de Módulo

Cada módulo externo es un ejecutable independiente que intercambia datos estructurados con `Shoal` a través de mensajes `JSON` en `stdin` y `stdout`.

`Shoal` expone objetos auxiliares y utilidades (ver Listado 12) para manejar este protocolo de manera transparente, a través del paquete público llamado `shoalmod` que forma parte del repositorio principal de `Shoal`.

Un módulo comúnmente utiliza esta API para:

1. Analizar la solicitud entrante con `ReadRequest()`.
2. Decodificar el campo `args` de la solicitud en una estructura tipada con `DecodeArgs(...)`.
3. Ejecuta la lógica.
4. Retorna el resultado a `Shoal` utilizando `WriteResult(...)`.

El Listado 13 muestra un ejemplo de módulo utilizando el paquete `shoalmod`.

Este protocolo ligero permite que cualquier lenguaje compilado pueda ser utilizado, siempre y cuando escriba y lea datos en JSON en `stdin/stdout`.

```go
// github.com/fpabl0/shoal/pkg/module
package shoalmod

import (...)

// Request module request input.
type Request struct {
	ProtocolVersion string         `json:"protocolVersion"`
	Args            map[string]any `json:"args"`
	TimeoutSeconds  int            `json:"timeoutSeconds"`
}
// Result module result output.
type Result struct {
	OK       bool           `json:"ok"`
	Changed  bool           `json:"changed"`
	ExitCode int            `json:"exitCode,omitempty"`
	Stdout   string         `json:"stdout,omitempty"`
	Stderr   string         `json:"stderr,omitempty"`
	Outputs  map[string]any `json:"outputs,omitempty"`
}
// ReadRequest reads JSON from stdin and returns the request + context
// with timeout.
func ReadRequest() (context.Context, context.CancelFunc, *Request, error)
// DecodeArgs decodes the Args field from the Request into a typed struct.
func DecodeArgs(req *Request, out any) error
// EncodeOutput encodes an output object into a map[string]any.
func EncodeOutput(output any) (map[string]any, error)
// WriteResult writes JSON to os.Stdout
func WriteResult(res Result)
// Fatal is a helper for fatal errors.
func Fatal(err error)
```

Listado 12: API Pública de Módulo

```
package main

import (
  ...
  shoalmod "github.com/fpabl0/shoal/pkg/module"
)

type Args struct {
    Path  string `json:"path"`
}

func main() {
    ctx, cancel, req, err := shoalmod.ReadRequest()
    if err != nil {
        shoalmod.Fatal(err)
    }
    defer cancel()

    var args Args
    if err := shoalmod.DecodeArgs(req, &args); err != nil {
        shoalmod.Fatal(fmt.Errorf("invalid args: %w", err))
    }

    // ... perform actions

    shoalmod.WriteResult(shoalmod.Result{
        OK:       true,
        Changed:  true,
        ExitCode: 0,
        Outputs:  map[string]any{
           "msg": fmt.Sprintf("Created %s", args.Path),
        },
    })
}
```

Listado 13: Ejemplo de Módulo con `shoalmod`

```yaml
name: "com.example.fs.audit"
author: "Example Labs"
description: "Scans directories for file integrity changes."
version: "1.0.0"
scope: "remote" # remote / local / both
os_archs:
  - "linux/amd64"
  - "windows/amd64"
```

Listado 14: Ejemplo de Manifiesto de Módulo

3.4.2. Manifiesto del Módulo

Cada módulo externo u oficial incluye un archivo de manifiesto llamado `module.yaml` que define sus metadatos y el alcance de ejecución.

Este manifiesto permite que el `FileModuleRegistry` de `Shoal` registre módulo dinámicamente, verifique la compatibilidad y garantice una nomenclatura predecible.

El Listado 13 y la Tabla 3.4 muestran el esquema y los campos utilizados en el manifiesto `module.yaml`.

3.4.3. Módulos integrados, oficiales y de terceros

`Shoal` distingue tres categorías de módulos:

Módulos integrados

Son implementados directamente en el `Engine` o `RemoteExecutor`. Estos módulos no son binarios externos, sino que forman parte de la lógica central de `Shoal`.

Esos módulos siempre están disponibles y no pueden ser reemplazados ni desinstalados. La Tabla 3.5 muestra los módulos integrados de `Shoal`:

Módulos oficiales

Estos módulos se distribuyen dentro del binario de la CLI de `Shoal` usando la directiva `go:embed` de Go.

Durante la primera llamada a `shoal run`, la CLI verifica si estos módulos se encuentran instalados en el directorio de módulos del usuario

Tabla 3.4: Campos en el Manifiesto de Módulo

Campo	**Descripción**
`name`	Identificador único del módulo. Los módulos oficiales usan nombre cortos (ej.: `file.manage`); módulos de terceros deben seguir el formato FDQN (*Fully Qualified Domain Name*) reverso conteniendo al menos tres puntos (ej.: `com.example.fs.audit`).
`author`	El nombre del creador u organización.
`description`	Pequeña descripción del propósito del módulo.
`version`	Número de versión del módulo.
`scope`	Declara el ambiente de ejecución (`remote`, `local` o `both`).
`os_archs`	Parejas de sistemas operativos y arquitecturas soportadas.

Tabla 3.5: Módulos integrados de `Shoal`

Módulo	**Capa**	**Propósito**
`vars.set`	`Engine`	Modifica variables en tiempo de ejecución en el contexto de host actual.
`debug.print`	`Engine`	Imprime un mensaje de depuración (acepta variables), el cual es previamente sujeto a la sustitución de variables y enviado al `ReporterBuilder` configurado.
`transfer.push`	`RemoteExecutor`	Copia archivos desde el controlador a nodos remotos usando `RemoteTransport`.
`transfer.pull`	`RemoteExecutor`	Copia archivos desde el nodo remoto al controlador usando `RemoteTransport`.

(`~/.shoal/modules`). Si no se encuentran, `Shoal` los extrae e instala de manera offline los archivos .zip embebidos en la CLI.

Esto permite que la herramienta sea autocontenida y pueda operar sin acceso a internet.

Actualmente, `Shoal` cuenta con los siguiente módulos oficiales:

- `file.manage`
- `file.copy`
- `file.move`
- `pkg.apt`
- `pkg.choco`
- `shell.posix`
- `shell.powershell`
- `shell.run`

Módulos de terceros

Contribuidores externos o usuarios puede desarrollar sus propios módulos utilizando la API pública descrita anteriormente. Estos módulos difieren de los oficiales por su convención de nomenclatura:

- Debe contener al menos tres puntos siguiendo el formato FDQN reverso (ej.: `com.example.fs.audit`).
- Nombres con menos puntos están reservados para el ecosistema oficial de `Shoal`.

Esta regla de nomenclatura obligatoria previene colisiones de nombres y permite a Shoal gestionar de forma segura la precedencia de módulos.

3.4.4. Empaquetado e Instalación

Los módulos son distribuidos en archivos *.zip* que contiene los binarios del módulo compilados para cada par de sistema operativo y arquitectura que éste soporta, y el manifiesto del módulo `module.yaml`. La CLI de `Shoal` proporciona comandos para ayudar al desarrollo de módulos de terceros.

Creación de un módulo

Para crear un nuevo módulo, `Shoal` CLI dispone del comando `shoal mod create`.

Por ejemplo, al ejecutar el comando del Listado 15, el usuario obtendrá una estructura de archivos (ver Listado 16) lista para comenzar el desarrollo de un nuevo módulo.

```
shoal mod create com.example.fs.audit
```

Listado 15: Ejemplo de creación de módulo

```
com.example.fs.audit/
|-- module.yaml
|-- go.mod
|-- main.go
|-- README.md
```

Listado 16: Estructura base para desarrollo de un módulo

Empaquetado

Cuando el desarrollo del módulo ha concluido, es necesario preparar el archivo .zip para la distribución. `Shoal` CLI proporciona el comando del Listado 17 para esta finalidad, el cual ejecuta las siguientes acciones:

1. Valida el manifiesto del módulo (`module.yaml`).
2. Compila el módulo para obtener los binarios para cada uno de las parejas sistema operativo y arquitectura definidas en el manifiesto del módulo.
3. Genera un archivo `zip` con el nombre del módulo.

```
shoal mod package
```

Listado 17: Empaquetado de módulo

Instalación

El comando del Listado 18 es utilizado para la instalación de un módulo a partir de un archivo `zip`. Durante la instalación, `Shoal`:

- Valida que el módulo respete la convención de nomenclatura en formato FDQN reverso.
- Verifica que exista un binario para cada una de los OSArchs definidos en el manifiesto.
- Registra el módulo en el directorio `~/.shoal/modules`.

```
shoal mod install <module>.zip
```

Listado 18: Instalación de un módulo

3.4.5. Flujo de ejecución y resolución

En tiempo de ejecución, cuando un `step` referencia un módulo (ej.: `module: file.manage`), `Shoal` realiza lo siguiente:

1. El `Engine` recibe la información `step`, donde se encuentra también el nombre del módulo a ejecutar.
2. Si el nombre del módulo es exactamente igual a `vars.set` o `debug.print`, la lógica de modificación de variables o impresión de mensaje de depuración se realiza, respectivamente, de lo contrario, el `Engine` delega la ejecución al `Executor`.
3. El `Executor` llama a `ModuleRegistry.Lookup(...)` para obtener la información del módulo.
4. El `Executor` valida en base a esta información si el módulo puede ejecutarse o no en el host actual.

5. Verifica si el módulo especificado es uno de los módulos integrados (`transfer.push` o `transfer.pull`), de ser así, su lógica es manejada internamente.

6. Verifica si el módulo existe, y obtiene el binario correspondiente para el OSArch del host objetivo con ayuda de `ModuleRegistry.BinaryPath(...)`.

7. Ejecuta el binario en el nodo remoto con ayuda de `RemoteTransport`.

8. Pasa los argumentos de entrada del módulo, serializados en formato JSON a través de `stdin`.

9. Lee el resultado de `stdout` y lo propaga de vuelta al `Engine`.

Esta separación de responsabilidad de registro, transporte y ejecución, mantiene a `Shoal` modular y agnóstico a lenguajes de programación.

3.4.6. Resumen

El sistema de módulos de `Shoal` provee un claro límite entra la configuración declarativa y lógica de ejecución.

A través de una API pública y registro de módulos, y un esquema de nombres controlado, se consigue:

- Extensibilidad segura sin interpretadores en tiempo de ejecución.
- Distribución offline de módulos oficiales.
- Clara separación entre módulos integrados y módulos desarrollados por el usuario.

Todas estas características forman un ecosistema de automatización autocontenido y extensible que permanece consistente con la meta de diseño de `Shoal`: ejecución ligera, determinista y libre de dependencias.

3.5. Diseño de la Interfaz de Línea de Comandos (CLI)

`Shoal` proporciona una CLI unificada a través de la cual los usuarios interactúan con las funcionalidades centrales del sistema, como la ejecución de `swimbooks`, verificación de conectividad y manejo de módulos.

La CLI está implementada con la librería Cobra de Go, la cual permite definir jerárquicamente comandos y subcomandos, análisis automático de flags, y generación de ayuda integrada.

Este diseño asegura extensibilidad y una experiencia de usuario consistente en todos los comandos.

La CLI sirve como la capa de control principal que coordina la carga de la configuración, selección de hosts y `swimbook` a ejecutar. Actúa como el adaptador de nivel superior en la arquitectura de `Shoal`, llamando a servicios internos como `Engine`, `SwimbookLoader`, `ControllerPlanLoader` y `HostSource` según los argumentos y flags especificados por el usuario.

3.5.1. Jerarquía y estructura de comandos

Cada comando de la CLI de `Shoal` se corresponde con un caso de uso de alto nivel, mientras los subcomandos y flags permiten un control más granular. La Tabla 3.7 resume los comandos disponibles en la CLI.

Tabla 3.6: Comandos de la CLI de `Shoal`

Comando	Descripción
`shoal run`	Ejecuta un `swimbook` en los hosts especificados
`shoal ping`	Verifica la conexión SSH y la autenticación con los hosts objetivos
`shoal mod`	Maneja los módulos (create, package, install, list, uninstall)
`shoal init`	Inicializa un nuevo directorio para un proyecto de `Shoal`
`shoal version`	Muestra la versión de `Shoal`

3.5.2. shoal run

El comando `shoal run` es el punto de entrada principal para la ejecución de tareas de automatización. Este comando requiere como argumento el path del archivo `Swimbook` y opcionalmente acepta filtros de hosts y archivos de variables de entorno a través de flags. El Listado 19 muestra un ejemplo.

```
shoal run -H web1,web2 deploy_app.yaml
```

Listado 19: Instalación de un módulo

El proceso de ejecución es el siguiente:

1. **Carga de archivos de configuración:**
   La CLI carga el `Swimbook` especificado, el `ControllerPlan` referenciado y el archivo de hosts (`hosts.yaml`).

2. **Resuelve hosts objetivos:**
   Filtros aplicados por medio de los flags `-H` y `-G`, seleccionan el subconjunto de hosts en donde se ejecutará el `Swimbook`.

3. **Delega ejecución:**
   El método `Engine.RunSwimbook(...)` orquesta todos los `steps` definidos en los hosts objetivos usando `goroutines` para concurrencia.

4. **Resultados de salida:**
   Los resultados son recolectados y mostrados usando el `Reporter` configurado, por defecto `ConsoleReporter`.

Este comando abstrae toda la complejidad interna, permitiendo a los usuarios ejecutar una automatización distribuida usando un único comando simple.

3.5.3. shoal ping

El comando `shoal ping` (ver Listado 20) permite a los usuarios probar la conectividad de los hosts objetivos antes de la ejecución. El comando intenta establecer una sesión SSH usando las credenciales definidas en `hosts.yaml`.

```
shoal ping -G linux_nodes
```

Listado 20: Ejemplo de comando shoal ping

El comando regresa un resumen indicando si cada nodo aceptó la conexión, falló la autenticación o no pudo ser alcanzado. Esta funcionalidad de diagnóstico ayuda a los usuarios a verificar el acceso SSH y la correcta configuración sin ejecutar ninguna tarea.

3.5.4. shoal mod

La familia de comandos `shoal mod ...` maneja el ecosistema de módulos de `Shoal`. Permite a los desarrolladores y usuarios crear, compilar, empaquetar, instalar e inspeccionar los módulos instalados. En la Tabla se pueden observar todos los subcomandos disponibles.

Tabla 3.7: Subcomandos de `shoal mod`

Subcomando	**Descripción**
`shoal mod create <name>`	Crea la estructura base de un módulo, con un archivo fuente y manifiesto `module.yaml` por defecto.
`shoal mod package`	Compila y empaqueta el módulo en un archivo `.zip` para la instalación.
`shoal mod install <zip>`	Instala un módulo desde un archivo zip.
`shoal mod list`	Lista todos los módulos instalados
`shoal mod uninstall <name>`	Elimina el módulo del sistema.

3.5.5. shoal init

El comando `shoal init` inicializa un nuevo proyecto de `Shoal`, creando una estructura de archivos base que contiene todo lo necesario para empezar a usar la herramienta.

Esto ayuda a los usuarios a crear rápidamente nuevos proyectos sin necesidad de crear manualmente los archivos requeridos.

Después los usuarios pueden editar las plantillas generadas para ajustarla a sus necesidades, permitiendo el uso inmediato de `shoal run`.

Por ejemplo, el comando del Listado 21, produce la estructura descrita en el Listado 22.

```
shoal init myproject
```

Listado 21: Ejemplo de comando `shoal init`

```
myproject/
|-- .env
|-- hosts.yaml
|-- swimbook.yaml
|-- controller_plan.yaml
```

Listado 22: Estructura de archivos creada por `shoal init`

3.5.6. shoal version

Muestra la versión actual de la herramienta instalada. Ver Listado 23.

```
shoal version
CLI Version: v0.1.0
```

Listado 23: Ejemplo de `shoal version`

3.5.7. Flujo de trabajo

El diseño de la CLI sigue un flujo de trabajo consistente:

- **Configuración declarativa, control imperativo:** Los usuarios describen tareas de automatización en `Swimbooks`, pero el control de ejecución es imperativo a través de la CLI.
- **Operación offline:** Todos los módulos oficiales y plantillas están embebidos dentro del binario de la herramienta, permitiendo que la ejecución no requiera de acceso a internet.
- **Sistema de comandos extensible:** Nuevos comandos y funcionalidades pueden añadirse a la CLI sin necesidad de modificar los ya existentes, gracias a la estructura de comandos de Cobra.

Un flujo de trabajo usual combina los comandos mostrados en el Listado 24.

```
shoal init myproject
cd myproject/
shoal ping -G linux_nodes
shoal run -G linux_nodes swimbook.yaml
```

Listado 24: Shoal CLI - Flujo de trabajo usual

3.6. Validación y Resultados

El propósito de esta sección es validar la implementación de `Shoal` y demostrar que los requisitos funcionales y no funcionales han sido conseguidos satisfactoriamente. La validación fue realizada con la ayuda de pruebas funcionales controladas usando escenarios representativos para la automatización de tareas y despliegue de artefactos a nodos remotos. En lugar de centrarse en la evaluación del rendimiento, el objetivo de estas pruebas fue verificar la correcta ejecución y usabilidad del prototipo y su independencia de interpretadores externos.

3.6.1. Entorno de pruebas

Todos los procedimientos de validación se ejecutaron en un entorno mixto de plataformas. La configuración incluyen un host controller con macOS, donde es ejecutado `Shoal CLI` y dos nodos controlados (una instancia de Ubuntu 24.04 y otra instancia de Windows Server) ambos alcanzables a través de SSH. La Tabla 3.8 muestra los componentes usados para las pruebas.

Esta configuración permite la verificación de las capacidades sin agentes y multi-plataforma de `Shoal`, confirmando que no se requirió entornos de ejecución adicionales (como Python o Ruby) en los nodos administrados.

Archivos de Configuración

Para asegurar la reproducibilidad y la trazabilidad de la configuración de validación, un conjunto de archivos de configuración se usaron para definir los hosts objetivos, variables de entorno y la lógica de aprovisionamiento para ambos nodos administrados.

Todos estos archivos se encuentran el directorio `examples` del repositorio principal de `Shoal`.

Tabla 3.8: Entorno de pruebas

Componente	Especificación
Controlador	macOS 13.7 (64-bit, amd64)
Versión de Shoal	0.1.0 (prototipo)
Nodo controlado 1	Ubuntu 24.04 (64-bit), OpenSSH habilitado
Nodo controlado 2	Windows Server 2019 (64-bit), OpenSSH habilitado
Red	VLAN privada (VirtualBox NAT)
Aprovisionamiento	Vagrant 2.4.3
Método de conexión	Autenticación por llave y contraseña SSH

El archivo `hosts.yaml` utilizado se puede observar en el Listado 25. Este archivo define los nodos a controlar y sus credenciales de acceso.

```yaml
hosts:
  - name: linux1
    address: 127.0.0.1
    os_arch: linux/amd64
    user: vagrant
    ssh_port: 2222
    ssh_private_key_file: ./.vagrant/machines/linux1/virtualbox/private_key
    env:
      PRINT_CMD: "echo"
  - name: win1
    address: 127.0.0.1
    os_arch: windows/amd64
    user: vagrant
    ssh_port: 2223
    password: vagrant # sólo para demostrar conectividad con ambos métodos
    env:
      PRINT_CMD: "Write-Output"
```

Listado 25: Pruebas - hosts.yaml

El archivo `.env` se muestra en el Listado 26. Este archivo provee de variables de entorno accesibles desde el `renderer` de `Shoal` como `{{ .Env.CONTROLLER_OS }}` . Esta variable fue usada para verificar la

interpolación de plantillas dentro de los argumentos del módulo especificado en una tarea.

```
CONTROLLER_OS=macOS
```

Listado 26: Pruebas - .env

El archivo `Vagrantfile` (ver Listado 27) aprovisiona ambos nodos en un ambiente controlado, permitiendo pruebas consistentes y reproducibles.

Reproducibilidad

Estos archivos de configuración, junto al binario de `Shoal` y el conjunto de módulos oficiales, constituyen un banco de pruebas reproducible usado para la validación. Cada prueba descrita en la Sección 3.6.2 utilizó este mismo ambiente para asegurar la consistencia de resultados.

3.6.2. Demostración Funcional

La validación funcional fue estructurada alrededor de cuatro escenarios de pruebas, cada uno correspondiente a uno de los requisitos funcionales definidos en la Sección 3.1.1.

Todos los experimentos fueron ejecutados desde una máquina con macOS (controlador), que controló dos nodos (Ubuntu 24.04 y Windows Server 2019).

Ambos nodos fueron alcanzables por SSH y no requirieron de algún intérprete adicional como Python o Ruby.

RF1 - Ejecución remota de tareas vía SSH

Objetivo:
Verificar que `Shoal` puede establecer sesiones SSH en ambos hosts (Ubuntu y Windows) y ejecutar comando remotos de forma concurrente sin requerir de ningún agente de software.

Procedimiento:
Un `Swimbook` llamado `sb_rf1.yaml` (Ver Listado 28) fue creado para ejecutar un simple comando para visualizar la información del sistema de cada host.

```
Vagrant.configure("2") do |config|
  config.vm.define "linux1" do |linux1|
    linux1.vm.box = "hashicorp-education/ubuntu-24-04"
    linux1.vm.box_version = "0.1.0"
    linux1.vm.network "forwarded_port", guest: 22, host: 2222, id: "ssh"
  end
  config.vm.define "win1" do |win1|
    win1.vm.box = "StefanScherer/windows_2019_docker"
    win1.vm.box_version = "2021.05.15"
    win1.vm.network "forwarded_port", guest: 22, host: 2223, id: "ssh"
    win1.vm.provision "shell", inline: <<-SHELL
      netsh advfirewall set allprofiles state off
      netsh advfirewall firewall add rule name="Allow ICMPv4-In"
       protocol=icmpv4 dir=in action=allow
      Start-Process PowerShell -Verb RunAs
      Get-WindowsCapability -Online | Where-Object Name -like 'OpenSSH*'
      # Install the OpenSSH Server
      Add-WindowsCapability -Online -Name OpenSSH.Server~~~~0.0.1.0
      # Start the sshd service
      Start-Service sshd
      Set-Service -Name sshd -StartupType 'Automatic'
      if (!(Get-NetFirewallRule -Name "OpenSSH-Server-In-TCP"
          -ErrorAction SilentlyContinue |
          Select-Object Name, Enabled)) {
          Write-Output "Firewall Rule 'OpenSSH-Server-In-TCP' does
           not exist, creating it..."
          New-NetFirewallRule -Name 'OpenSSH-Server-In-TCP'
           -DisplayName 'OpenSSH Server (sshd)' -Enabled True
           -Direction Inbound -Protocol TCP -Action Allow -LocalPort 22
      } else {
          Write-Output "Firewall rule 'OpenSSH-Server-In-TCP' has been
           created and exists."
      }
    SHELL
  end
end
```

Listado 27: Pruebas - Vagrantfile

```yaml
name: "RF1"
steps:
  - name: "Get system info"
    module: "shell.run"
    args:
      command: >
        {{ if eq .Host.OSArch `windows/amd64` }}
          $os=(Get-CimInstance Win32_OperatingSystem).Caption;
          $arch=$env:PROCESSOR_ARCHITECTURE;
          Write-Output ($os + ' ' + $arch)
        {{ else }}
          uname -a
        {{ end }}
    bind_output: out
  - name: "Show system info"
    module: "debug.print"
    args:
      msg: '{{ .Vars.out.stdout }}'
```

Listado 28: Prueba RF1 - `sb_rf1.yaml`

Ejecución:

```
shoal run sb_rf1.yaml
```

Resultado:

`Shoal` se conectó exitosamente en ambos hosts a través SSH. El nodo de Ubuntu regresó la información del kernel y la arquitectura, mientras que el nodo de Windows regresó el nombre del sistema operativo y arquitectura obtenida desde Powershell. Los registros de salida confirman la ejecución paralela a través de `goroutines` concurrentes iniciadas por el `Engine`, validando la orquestación de tareas a múltiples nodos a través de SSH.

RF2 - Transferencia de artefactos

Objetivo:
Demostrar que `Shoal` puede transferir archivos entre el controlador y los nodos administrados, usando los módulo integrados e independientes de

utilidades externas como scp.

Procedimiento:
El Swimbook sb_rf2.yaml fue utilizado para esta prueba (Ver Listado 29). Un archivo de texto rf2_welcome.txt localizado en el controlador fue subido a ambos nodos y luego es obtenido de vuelta para su verificación usando los módulos transfer.push transfer.pull.

```
name: "RF2 - File Transfer Test"
steps:
  - name: "Upload message"
    module: "transfer.push"
    args:
      src: "./rf2_welcome.txt"
      dest: >
        {{ if eq .Host.OSArch `windows/amd64` }}
          C:\\Temp\\welcome.txt
        {{ else }}
          /tmp/welcome.txt
        {{ end }}
  - name: "Retrieve confirmation"
    module: "transfer.pull"
    args:
      src: >
        {{ if eq .Host.OSArch `windows/amd64` }}
          C:\\Temp\\welcome.txt
        {{ else }}
          /tmp/welcome.txt
        {{ end }}
      dest: "./rf2_results/{{ .Host.Name }}_welcome.txt"
```

Listado 29: Prueba RF2 - sb_rf2.yaml

Ejecución:

```
shoal run sb_rf2.yaml
```

Resultado:

La transferencia resultó exitosa en ambos hosts. Ningún comando auxiliar SSH fue utilizado, las interfaces `Executor` y `RemoteTransport` manejan todas las transmisiones de manera interna.

RF3 - Interpretar archivos de configuración en YAML

Objetivo:
Validar que `Shoal` analiza correctamente los archivos de configuración basados en YAML, detectando errores de sintaxis.

Procedimiento:
Un archivo `Swimbook` (ver Listado 30) intencionalmente malformado fue usado primero para confirmar errores de validación, seguido de una versión corregida ejecutada exitosamente.

```yaml
name: "RF3 - Invalid Syntax"
vars: "my var"
steps:
  - name: "Print hello"
    module: "debug.print"
    args:
      msg: 'Hello'
```

Listado 30: Prueba RF3 - Swimbook Malformado - `sb_rf3_invalid.yaml`

Al ejecutarse:

```
shoal run sb_rf3_invalid.yaml
```

`Shoal` retornó un error al analizar el `Swimbook`, señalando que el campo `vars` debe ser un mapa, no un simple `string`.
Luego se corrigió el `Swimbook` al mostrado en el Listado 31.
Este último se ejecutó sin problemas en ambos nodos.

Resultado:

Los analizadores YAML internos de `Shoal` realizaron una validación estricta de sintaxis y de esquema previo a la ejecución, previniendo malas configuraciones en tiempo de ejecución.

```yaml
name: "RF3 - Valid Syntax"
vars:
  name: "Pablo"
steps:
  - name: "Print hello"
    module: "debug.print"
    args:
      msg: "Hello {{ .Vars.name }} from {{ .Host.Name }}"
```

Listado 31: Prueba RF3 - Swimbook Válido - `sb_rf3_valid.yaml`

RF4 - Sustitución de variables y plantillas

Objetivo:
Asegurar que `Shoal` realice correctamente la sustitución de variables y valores de entorno in los `steps`, aplicando la sintaxis del paquete `text/template` de Go de forma segura y consistente en todos los nodos.

Procedimiento:
Se dispuso de un archivo `Swimbook` (ver Listado 32) que usa internamente variables propias del `Swimbook` y variables de entorno (globales y por host).

Ejecución:

```
shoal run sb_rf4.yaml
```

Resultado:

`Shoal` realizó la sustitución de variables en cada comando de manera correcta, produciendo:

- En Ubuntu: `Hello macOS from linux1`
- En Windows: `Hello macOS from win1`

Las variables no definidas desencadenan errores explícitos de plantilla en lugar de fallos silenciosos, confirmando que el sistema de plantillas de Go aplica validaciones estrictas, garantizando un comportamiento determinista en ambos nodos.

```yaml
name: "RF4 - Variable & Template Rendering"

vars:
  greeting: "Hello"
  msg: ""

steps:
  - name: "Prepare message"
    module: vars.set
    args:
      key: msg
      value: >
        {{ .Vars.greeting }} {{ .Env.CONTROLLER_OS }} from {{ .Host.Name }}
  - name: "Echo variable"
    module: "shell.run"
    args:
      command: "{{ .Env.PRINT_CMD }} '{{ .Vars.msg }}'"
    bind_output: out
  - name: "Display output"
    module: "debug.print"
    args:
      msg: '{{ .Vars.out.stdout }}'
```

Listado 32: Prueba RF4 - `sb_rf4.yaml`

Resumen

A lo largo de los cuatro requisitos funcionales, `Shoal` ejecutó tareas, analizó archivos de configuración, transfirió artefactos y sustituyó variables en plantillas de forma consistente utilizando Go.

Cada escenario validó la interoperabilidad entre los componentes `Engine`, `Executors`, `Renderer` y `RemoteTranport` en entornos heterogéneos.

Todas las pruebas se completaron sin conectividad de red más allá de SSH y sin la necesidad de ningún intérprete externo, confirmando la independencia de `Shoal` y los objetivos de diseño.

3.6.3. Salida de la CLI y Capturas de Pantalla

Esta sección presenta salidas de terminal representativas recopiladas durante la validación de cada requisito funcional (ver Sección 3.6.2).

RF1 - Ejecución remota de tareas vía SSH

```
pablofuentes:~/fpabl0/shoal/example % shoal run sb_rf1.yaml

SWIMBOOK: RF1  (targets: 2)

------------------------------------------[ REMOTE ]------------------------------------------
→ Step 1/2: Get system info        (shell.run)
    • linux1     OK (409ms)
    • win1       OK (1.89s)
  ✓ 1.89s

→ Step 2/2: Show system info       (debug.print)
    • win1       OK (0s)
      DebugPrint:
-----------------------------------------------
Microsoft Windows Server 2019 Datacenter Evaluation AMD64

-----------------------------------------------
    • linux1     OK (0s)
      DebugPrint:
-----------------------------------------------
Linux vagrant-ubuntu 6.8.0-51-generic #52-Ubuntu SMP PREEMPT_DYNAMIC Thu Dec  5 13:09:44 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux

-----------------------------------------------
  ✓ 0s

... completed in 1.891s ✓
----------------------------------------------------------------------------------------------

SWIMBOOK RESULT: RF1 ✓ success in 1.891s
```

Figura 3.6: Shoal - Prueba RF1

La Figura 3.6 demuestra conexión SSH exitosa y la ejecución concurrente de comandos. Cada nodo reporta correctamente su correspondiente información de sistema operativo.

RF2 - Transferencia de artefactos

```
pablofuentes:~/fpabl0/shoal/example % shoal run sb_rf2.yaml

SWIMBOOK: RF2 - File Transfer Test  (targets: 2)

------------------------------------------[ REMOTE ]------------------------------------------
→ Step 1/2: Upload message          (transfer.push)
    • win1       OK (481ms)
    • linux1     OK (1.16s)
  ✓ 1.16s

→ Step 2/2: Retrieve confirmation  (transfer.pull)
    • linux1     OK (17ms)
    • win1       OK (67ms)
  ✓ 67ms

... completed in 1.227s ✓
----------------------------------------------------------------------------------------------

SWIMBOOK RESULT: RF2 - File Transfer Test ✓ success in 1.227s
```

Figura 3.7: Shoal - Prueba RF2

Las Figuras 3.7 y 3.8 verifican la transferencia de archivos bidireccional entre el controlador y ambos nodos. `Shoal` envió y recibió exitosamente archivos usando los módulos integrados de transferencia.

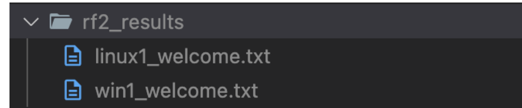

Figura 3.8: Archivos de salida - Prueba RF2

RF3 - Interpretar archivos de configuración en YAML

```
pablofuentes:~/fpabl0/shoal/example % shoal run sb_rf3_invalid.yaml
Error: swimbook loader: unmarshal: yaml: unmarshal errors:
  line 2: cannot unmarshal !!str `my var` into map[string]interface {}
```

Figura 3.9: Shoal - Prueba RF3 - Swimbook inválido

```
pablofuentes:~/fpabl0/shoal/example % shoal run sb_rf3_valid.yaml

SWIMBOOK: RF3 – Valid Syntax  (targets: 2)

------------------------------------------[ REMOTE ]------------------------------------------
→ Step 1/1: Print hello            (debug.print)
    • win1        OK (0s)
      DebugPrint:
-----------------------------------------------
Hello Pablo from win1
-----------------------------------------------
    • linux1      OK (0s)
      DebugPrint:
-----------------------------------------------
Hello Pablo from linux1
-----------------------------------------------
  ✓ 0s

... completed in 0s ✓
----------------------------------------------------------------------------------------------

SWIMBOOK RESULT: RF3 – Valid Syntax ✓ success in 0s
```

Figura 3.10: Shoal - Prueba RF3 - Swimbook válido

En las Figura 3.9 se aprecia que el analizador YAML de `Shoal` identificó correctamente un error en el esquema YAML y proporcionó información de cómo resolver el problema.

Tras la corrección (ver Figura 3.10), el `Swimbook` fue analizado y ejecutado correctamente.

RF4 - Sustitución de variables y plantillas

En la Figura 3.11 se ilustra que las variables de `Swimbook` y las variables de entorno (globales y por host) definidas fueron correctamente sustituidas en tiempo de ejecución. Ambos nodos produjeron una salida específica por host,

```
pablofuentes:~/fpabl0/shoal/example % shoal run sb_rf4.yaml

SWIMBOOK: RF4 — Variable & Template Rendering  (targets: 2)

-----------------------------------------[ REMOTE ]-----------------------------------------
→ Step 1/3: Prepare message        (vars.set)
    • win1       OK (0s)
    • linux1     OK (0s)
  ✓ 0s

→ Step 2/3: Echo variable          (shell.run)
    • linux1     OK (1.4s)
    • win1       OK (1.83s)
  ✓ 1.83s

→ Step 3/3: Display output         (debug.print)
    • win1       OK (0s)
      DebugPrint:
----------------------------------------------
Hello macOS from win1

----------------------------------------------
    • linux1     OK (0s)
      DebugPrint:
----------------------------------------------
Hello macOS from linux1

----------------------------------------------
  ✓ 0s

... completed in 1.831s ✓
--------------------------------------------------------------------------------------------

SWIMBOOK RESULT: RF4 — Variable & Template Rendering ✓ success in 1.831s
```

Figura 3.11: Shoal - Prueba RF4

demostrando que el motor de plantillas de Go fue aplicado consistentemente en cada paso.

3.6.4. Resumen del cumplimiento de requisitos

Esta subsección resume cómo cada requisito funcional y no funcional definidos en la Sección 3.1 fueron verificados durante la validación.

Las Tabla 3.9 y 3.10 proporcionan una visión general de la trazabilidad que vincula cada requisito con un caso de pruebas, evidencias y resultados observados correspondientes.

Discusión

Todos los requisitos funcionales (RF1-RF4) se validaron a través de ejecuciones multi-plataforma, transferencia de artefactos, análisis de configuración y sustitución de variables. Estas pruebas confirmaron que `Shoal` cumple con el objetivo de automatizar entornos heterogéneos usando únicamente comunicación segura por SSH y componentes nativos de Go.

En cuanto a los requisitos no funcionales (RNF1-RNF7), las validaciones confirmaron una operación sin agentes, independencia de agentes e

Tabla 3.9: Validación de Requisitos Funcionales

ID	**Descripción**	**Método de Validación**	**Evidencia / Resultado**
RF1	Ejecución de tareas vía SSH	Ejecución del Swimbook `sb_rf1.yaml`	`Shoal` ejecutó correctamente los comandos específicos de cada SO en ambos hosts de manera concurrentes sin agentes intérpretes (Fig. 3.6)
RF2	Transferencia de artefactos	Ejecución del Swimbook `sb_rf2.yaml`	Los archivos fueron subidos y obtenidos de forma correcta. (Fig. 3.7)
RF3	Análisis de archivos de configuración	Ejecución de los Swimbooks `sb_rf3_invalid.yaml` y `sb_rf3_valid.yaml`	Errores en la validación del esquema fueron detectados con mensajes explícitos, Swimbooks válidos fueron ejecutados correctamente. (Fig. 3.9 y Fig. 3.10)
RF4	Sustitución de variables y plantillas	Ejecución del Swimbook `sb_rf4.yaml`	Las Variables del Swimbook y de entorno fueron sustituidas correctamente por host. (Fig. 3.11)

Tabla 3.10: Validación de Requisitos No Funcionales

ID	**Descripción**	**Método de Validación**	**Evidencia / Resultado**
RNF1	Operación sin agentes	No se instaló software adicional en los nodos a parte de OpenSSH	Todas las tareas se ejecutaron únicamente a través de SSH.
RNF2	Independencia de intérpretes externos	Revisión de las dependencias en tiempo de ejecución y las configuraciones de los nodos	No se requirió instalación de Python ni Ruby.
RNF3	Compatibilidad multi-plataforma	Las pruebas fueron realizadas en nodos Ubuntu 24.04 y Windows Server 2019	Todos los pasos del Swimbook fueron ejecutados exitosamente en ambos SO, confirmando la independencia de plataforma.
RNF4	Seguridad	Examinación de sesiones SSH y manejo de credenciales	Todas las comunicaciones fueron encriptadas a través de SSH.
RNF5	Rendimiento y escalabilidad (orientado al diseño)	Revisión arquitectónica del modelo de concurrencia del `Engine`	`Engine` soporta la ejecución de tareas en paralelo a través del parámetro `max_parallel_hosts`.
RNF6	Usabilidad	Observación de la sintaxis YAML y el flujo de trabajo de la CLI	Un DSL basado en YAML y una CLI intuitiva redujo la curva de aprendizaje de la herramienta.
RNF7	Extensibilidad y modularidad	Evaluación del manejo de módulos	Shoal carga módulos oficiales y de terceros dinámicamente sin necesidad de recompilación.

intérpretes externos, y una comunicación segura y encriptada.

`Shoal` demostró ser fácil de usar gracias a su DSL basado en YAML, su CLI intuitiva y su sistema de módulos extensible.

Aunque las métricas de rendimiento y escalabilidad no fueron medidas cuantitativamente debido al alcance del trabajo, el diseño concurrente interno ofrece una base sólida para futuras optimizaciones.

En resumen, esta evaluación demuestra que `Shoal` satisface todos los objetivos definidos en la Sección 1.2, confirmando su viabilidad como una solución de automatización ligera, independiente de intérpretes y agentes, para entornos de despliegue multi-plataforma.

Capítulo 4

Discusión y Conclusiones

Este capítulo presenta un análisis de los resultados obtenidos de la implementación de `Shoal`.

Discute las implicaciones de los resultados de la validación, compara el prototipo desarrollado con herramientas existentes, e identifica las principales limitaciones como también las área potenciales de mejora.

La discusión enfatiza cómo `Shoal` cumple con los objetivos previstos, particularmente habilitando las automatización sin dependencia de intérpretes externos.

Finalmente, el capítulo concluye resumiendo todas las contribuciones de este trabajo y delineando desarrollos futuros que podrían extender la funcionalidad de `Shoal` y fortalecer su posición como una alternativa ligera a las herramientas de gestión de configuración tradicionales.

4.1. Discusión de Resultados

Los resultados obtenidos de la validación de `Shoal` demuestran que es técnicamente viable implementar una herramienta de automatización multi-plataforma que opera enteramente sin intérpretes externos o agentes en los nodos. Los experimentos descritos al final del Capítulo 3 confirmaron la ejecución exitosa de tareas de configuración y despliegue en nodos basados en Debian Linux y Windows Server a través de SSH, validando así los principios de diseño fundamentales de simplicidad, seguridad e independencia de plataforma.

Un factor clave para alcanzar estos resultados, reside en la decisión de implementar `Shoal` en el lenguaje de programación Go. La capacidad de Go de producir binarios estáticos demostró ser crítica para los objetivos del proyecto de conseguir ejecutables autónomos que no dependen de dependencias externas. Esta propiedad simplifica la distribución de la herramienta como de los módulos. Además, las funciones de concurrencia integradas de Go permitieron al `Engine` manejar ejecución paralela de tareas eficientemente a través de múltiples nodos, soportando escalabilidad de tareas sin la necesidad de orquestadores externos.

La decisión de adoptar SSH como la única capa de comunicación también trajo implicaciones positivas relacionadas al diseño y seguridad. Al depender exclusivamente de OpenSSH (usualmente preinstalado en la mayoría de sistemas modernos), `Shoal` evitó la necesidad de agentes persistentes en los nodos administrados. Esta decisión no solo redujo la superficie de ataque del sistema, sino también simplificó la integración con infraestructuras de TI existentes. Los resultados de validación mostraron que ambos modos de autenticación SSH (por contraseña o por llave) operaron sin problemas, demostrando que el diseño pudo equilibrar la facilidad de configuración con la seguridad.

Desde el punto de vista de la usabilidad, la inclusión de un DSL basado en YAML proporcionó grandes ventajas. A través de una sintaxis declarativa construida sobre YAML y las plantillas de Go, `Shoal` ofreció un modelo de configuración que fue legible y fácil de entender. Las pruebas de validación confirmaron que los usuarios pueden definir variables, lógica condicional y parámetros de ejecución con un mínimo esfuerzo de aprendizaje. Esto reforzó la idea que los archivos de configuración pueden servir como una forma de documentación, disminuyendo la carga cognitiva a los administradores y facilitando la revisión y reproducibilidad. Comparado con enfoques tradicionales de scripts, el DSL basado en YAML minimizó la complejidad manteniendo un comportamiento determinístico a través de las

reglas estrictas de evaluación de plantillas de Go.

La arquitectura de módulos introdujo una capa adicional de extensibilidad. Los módulos integrados y los externamente empaquetados demostraron que nuevas funcionalidades pueden ser integradas sin recompilar el controlador (CLI). Este enfoque modular se alinea con los principios de diseño de software, promoviendo la separación de responsabilidades y el soporte de un ecosistema de componentes de automatización reutilizables. La clara distinción entre módulos integrados, oficiales y de terceros también establece una base para el control de versiones y el origen de paquetes, la cual es esencial para una automatización segura en ambientes de producción.

Finalmente, el proceso de validación resaltó la efectividad del flujo de ejecución y visualización de resultados. El modelo de concurrencia del `Engine` aseguró que múltiples nodos puedan ser manejados en paralelo, mientras el componente `Reporter` proporcionó una clara visibilidad de los resultados de las tareas. Las validaciones confirmaron que el diseño sin agentes no comprometió la granularidad de la retroalimentación ni la trazabilidad. Aunque las pruebas de escalabilidad quedaron fuera del alcance del proyecto, las decisiones arquitectónicas demostraron que futuras mejoras y/o funcionalidades pueden ser implementadas sin alterar la lógica central de la herramienta.

En resumen, los resultados obtenidos validan la viabilidad técnica y la solidez conceptual del diseño de `Shoal`. Combinando un motor de ejecución concurrente basado en Go, un modelo de configuración basado en YAML y una comunicación segura a través de SSH, `Shoal` demuestra que las herramientas de automatización pueden ser ligeras y aún soportar extensibilidad y operación multi-plataforma; emergiendo como una alternativa práctica y adaptable para entornos donde las herramientas tradicionales enfrentan desafíos debido a sus dependencias en intérpretes o agentes externos.

4.2. Comparación con Herramientas Existentes

El contraste de `Shoal` versus las herramientas de gestión de configuración existentes reside en una filosofía de diseño distinta centrada en la simplicidad y la autonomía. A diferencia de herramientas maduras como Ansible, Puppet y Chef, las cuales dependen de entornos de ejecución o agentes para operar, `Shoal` fue concebido para funcionar de manera autónoma e independiente de intérpretes. (Ver Tabla 4.1)

Esta decisión de diseño representa un giro intencional hacia la

minimización de dependencias en entornos, manteniendo la compatibilidad multi-plataforma y la facilidad de despliegue.

Ansible, por ejemplo, consigue una operación sin agentes a través de SSH, pero depende de la presencia de Python y algunos de sus módulos de sistema. Estas dependencias pueden introducir problemas de compatibilidad en entornos controlados con instalaciones mínimas, donde Python podría no estar disponible. Por el contrario, `Shoal` elimina las dependencias de intérpretes para operar. Su CLI y sus módulos compilados estáticamente con Go integran toda la funcionalidad necesaria para ejecutar tareas directamente por SSH, simplificando el despliegue y la complejidad de mantenimiento.

Esta elección de diseño mejora la portabilidad, pues, los nodos administrados no requieren de un entorno de ejecución o librerías preinstaladas para trabajar.

Puppet y Chef emplean una arquitectura cliente-servidor en la cual un controlador central controla los nodos usando agentes preinstalados para mantener la sincronización y la aplicación continua del estado deseado. Este modelo provee alta consistencia y control centralizado pero incrementa la complejidad de configuración e introduce puntos de mantenimiento adicionales. `Shoal` sigue el mismo principio arquitectónico de un controlador comunicándose con sus nodos administrados; sin embargo, prescinde de agentes por completo. Todas las operaciones son ejecutadas a través de sesiones SSH seguras iniciadas por el controlador, consiguiendo la ejecución de tareas coordinadas sin servicios persistentes en segundo plano. Este enfoque sin agentes simplifica el despliegue y es particularmente ventajoso para entornos de pequeña y mediana escala donde se busca reducir la sobrecarga administrativa sobre la aplicación continua del estado.

Desde la perspectiva del diseño del lenguaje de configuración, ambos Puppet y Chef usan sus propios DSLs derivados de Ruby, permitiendo el modelado de estados complejos, con la desventaja de una pronunciada curva de aprendizaje para nuevos usuarios. El DSL de `Shoal`, por el contrario, se enfoca en la claridad y la fácil adopción. Su sintaxis declarativa permite a los usuarios definir tareas, variables y lógicas condicionales de una manera concisa y legible, alineándose con la simplicidad de Ansible pero manteniendo la independencia de intérpretes externos.

En términos de extensibilidad, `Shoal` emplea una arquitectura modular que distingue entre módulos integrados, oficiales y de terceros. A diferencia del sistema de plugin basados en Python de Ansible y el de Puppet basado en Ruby, los módulos de `Shoal` son distribuidos como binarios independientes y registrados dinámicamente a través de la CLI. Este modelo minimiza las

interdependencias y soporta el desarrollo de módulos en cualquier lenguaje compilado, con la única restricción de que sean compatibles con la API pública de módulos de `Shoal`. Este diseño promueve la mantenibilidad, flexibilidad y la escalabilidad a largo plazo separando la lógica de módulos del motor central de la herramienta.

Finalmente, el objetivo de `Shoal` no es reemplazar estas herramientas que han demostrado estabilidad y madurez en entornos empresariales, sino complementarlas ofreciendo una alternativa minimalista enfocada en la ejecución directa de tareas, independencia de intérpretes y comunicación segura.

4.3. Limitaciones y Trabajo Futuro

Aunque las pruebas de validación demostraron que `Shoal` cumple con los objetivos de diseño, varias limitaciones existen aún en el prototipo, dejando abierta la posibilidad de desarrollo futuro.

La primera limitación está relacionada con las pruebas de escalabilidad. El prototipo fue probado en un ambiente controlado con únicamente dos nodos, de manera que el rendimiento bajo despliegues a gran escala no fue medido. Un trabajo futuro debería incluir pruebas de estrés para evaluar la eficiencia de la concurrencia, la utilización de recursos y el manejo de las conexiones cuando se opera con un gran número de nodos.

Una segunda limitación involucra el alcance de operaciones que `Shoal` puede manejar. La herramienta actualmente puede realizar tareas de automatización ejecutando un conjunto definido de pasos secuenciales en los nodos remotos y soporta lógica condicional básica a través del campo `if`, permitiendo que estos pasos se ejecuten o no basado en valores de variables o resultados previos. Sin embargo, `Shoal` aún no maneja interdependencias complejas entre múltiples nodos o servicios. Futuras versiones podrían expandir este alcance hacia capacidades completas de orquestación, como coordinación entre nodos, y mecanismos de reversión para gestionar errores de una mejor manera.

El ecosistema de módulos también se encuentra limitado. A pesar que los módulos integrados y oficiales probaron ser funcionales, expandir la biblioteca de módulos e introducir un registro verificado de módulos de terceros fortalecería la extensibilidad y la adopción en la comunidad.

Desde la perspectiva de la usabilidad, el DSL basado en YAML y la CLI fueron validados únicamente por el autor, sin pruebas de usuario más amplias. En un futuro, podría incluirse un estudio empírico de

usabilidad para obtener retroalimentación de administradores de sistemas y profesionales de DevOps, y así mejorar la experiencia general del usuario.

Finalmente, trabajos posteriores podrían enfocarse en mejorar las características de seguridad e integración. Algunas posibilidades incluyen conectar a `Shoal` con sistemas de gestión de secretos para manejar credenciales de manera más segura, implementar registros de auditoría detallados para el seguimiento de la ejecución y añadir la verificación de firmas digitales para asegurar la integridad de operaciones críticas.

En síntesis, aunque `Shoal` demuestra la viabilidad de una herramienta de automatización libre de intérpretes, su desarrollo futuro debería priorizar pruebas de escalabilidad, expansión de módulos, evaluaciones de usabilidad e integración con sistemas de seguridad, para convertirse en una solución lista para producción.

4.4. Conclusiones

Este trabajo presentó el diseño e implementación de `Shoal`, un prototipo de herramienta para la automatización de tareas y despliegue de aplicaciones usando una arquitectura cliente-servidor sin dependencias de intérpretes externos.

`Shoal` fue construido en Go y usa archivos de configuración basados en YAML y comunicación por SSH para ejecutar tareas de automatización eficientemente en nodos con sistemas basados en Debian Linux y Windows.

El estudio primero analizó las herramientas existentes como Ansible, Puppet y Chef, identificando su dependencia en entornos de ejecución externos o agentes como una limitación para entornos mínimos. Este análisis guió el diseño de una alternativa autónoma capaz de conseguir una funcionalidad similar a través de binarios compilados.

La implementación del prototipo cumplió con los objetivos propuestos: ejecución de tareas remotas concurrentes en Go, una interfaz de línea de comandos para configuración y control, y una estructura basada en YAML para definir tareas, variables y lógica de ejecución. Las pruebas de validación comprobaron la factibilidad de un enfoque independiente de intérpretes, resaltando su simplicidad, portabilidad y mantenibilidad.

Trabajos futuros deberían incluir pruebas de escalabilidad, evaluaciones de usabilidad más amplias y un ecosistema de módulos expandido para convertir a `Shoal` en una herramienta de automatización lista para entornos productivos.

Tabla 4.1: Shoal - Comparación con herramientas existentes

Criterio	**Ansible**	**Puppet**	**Chef**	**Shoal**
Arquit.	Sin agentes (SSH/ WinRM)	Basado en agentes	Basado en agentes	Sin agentes (SSH)
DSL	YAML Playbook	Puppet DSL	DSL basado en Ruby	YAML Swimbook
Aplicación estado	Manual	Automático	Automático	Manual
Compat. Servidor	Linux / macOS / Windows	Sólo Linux	Sólo Linux	Linux / macOS
Compat. nodos clientes	Linux / macOS / Windows	Linux / macOS / Windows	Linux / macOS / Windows	Linux / Windows
Curva de aprendizaje	Baja	Media	Alta	Baja
Protocolo comunic.	SSH/ WinRM	HTTPS/ SSL	HTTPS vía Chef Infra Server	SSH
Deps. en nodos	Python	Puppet Agent	Chef Infra Client	Ninguna
Módulos	Módulos en Python	Módulos usando Puppet Lang.	Ruby Cookbooks	Módulos en Go u otro lenguaje compilado

Bibliografía

Amazon Web Services Inc. YAML vs JSON - Difference Between Data Serialization Formats, 2025. URL https://aws.amazon.com/compare/the-difference-between-yaml-and-json/.

Ansible. Introduction to Ansible, 2025a. URL https://docs.ansible.com/ansible/latest/getting_started/introduction.html.

Ansible. Developing modules, 2025b. URL https://docs.ansible.com/ansible/latest/dev_guide/developing_modules_general.html.

Ansible. Ansible concepts, 2025c. URL https://docs.ansible.com/ansible/latest/getting_started/basic_concepts.html.

Ansible. How to build your inventory, 2025d. URL https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html#intro-inventory.

Ansible. Ansible playbooks, 2025e. URL https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html.

Ansible. Working with playbooks, 2025f. URL https://docs.ansible.com/ansible/latest/playbook_guide/playbooks.html.

Ansible. Controlling playbook execution: strategies and more, 2025g. URL https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_strategies.html.

Ansible. Templating (Jinja2), 2025h. URL https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_templating.html.

Ansible. Roles, 2025i. URL https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_reuse_roles.html.

Ansible. Installing Ansible, 2025j. URL https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html.

Ansible. Releases and maintenance, 2025k. URL https://docs.ansible.com/ansible/latest/reference_appendices/release_and_maintenance.html.

Ansible. Working with dynamic inventory, 2025l. URL https://docs.ansible.com/ansible/latest/inventory_guide/intro_dynamic_inventory.html.

Ansible. Inventory plugins, 2025m. URL https://docs.ansible.com/ansible/latest/plugins/inventory.html.

C. Arnault. CMT: A software configuration management tool. In *Prepared for International Conference on Computing in High-Energy Physics and Nuclear Physics (CHEP 2000), Padova, Italy*, pages 7–11, 2000. URL https://chep2000.pd.infn.it/short_p/spa_f033.pdf.

S. Arora. Chef in DevOps: Crafting Efficiency through Infrastructure Alchemy, 2024. URL https://medium.com/@sugam.arora23/chef-in-devops-crafting-efficiency-through-infrastructure-alchemy-cf8831652e49.

S. Arora. Ansible vs. Puppet: Know the Key Differences, 2025. URL https://www.simplilearn.com/ansible-vs-puppet-the-key-differences-to-know-article.

A. Brown and G. Wilson. *Puppet*, volume 2. Lulu, 2012. URL https://aosabook.org/en/v2/puppet.html.

M. Burgess. A Tiny Overview of Cfengine: Convergent Maintenance Agent, 2005. URL https://markburgess.org/papers/tiny_intro.pdf.

C. Cadman. Top Questions and Answers for Puppet on Windows, 2021. URL https://www.puppet.com/blog/puppet-on-windows.

Cisco DevNet. Models: Imperative vs. Declarative - Open NX-OS, 2025. URL https://developer.cisco.com/docs/nx-os/models-imperative-vs-declarative/.

H. Courdent. What is Ansible? A brief history, 2024. URL https://www.windmill.dev/blog/ansible-history.

B. Crist. The Fourth Chapter of Chef Has Arrived: Progress to Purchase Chef, 2020. URL https://www.chef.io/blog/the-fourth-chapter-of-chef-has-arrived-progress-to-purchase-chef.

Dagster Labs. Standardize Pipelines with Domain-Specific Languages, 2024. URL https://medium.com/@dagster-io/standardize-pipelines-with-domain-specific-languages-1f5729fc0f65.

T. Delaet, W. Joosen, and B. V. Distrinet. A survey of system configuration tools, 2010. URL https://www.usenix.org/legacy/event/lisa10/tech/full_papers/Delaet.pdf.

R. Diane and C. Stryker. What is task automation?, 2024. URL https://www.ibm.com/think/topics/task-automation.

A. Diklic. Introduction to Chef - Semaphore Tutorial, 2020. URL https://semaphore.io/community/tutorials/introduction-to-chef.

E. Dolstra, R. Vermaas, and S. Levy. Charon: Declarative provisioning and deployment. In *1st International Workshop on Release Engineering*, pages 17–20, 5 2013. doi: 10.1109/RELENG.2013.6607691. URL https://edolstra.github.io/pubs/charon-releng2013-final.pdf.

A. Fashakin. Agent Vs Agentless Configuration Management, 2025. URL https://attuneops.io/agent-vs-agentless-configuration-management/.

J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: A Deployment Management System. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 263–274. ACM, 2012. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254096. URL http://doi.acm.org/10.1145/2254064.2254096https://data-ken.org/papers/pldi12_engage.pdf.

V. Hardion, D. P. Spruce, M. Lindberg, A. M. Otero, J. Lidon-Simon, J. J. Jamroz, and A. Persson. Configuration management of the control system. *THPPC013*, 2013. URL https://accelconf.web.cern.ch/ICALEPCS2013/papers/thppc013.pdf.

R. Heaton. What is task automation?, 2025. URL https://www.techtarget.com/searchitoperations/definition/task-automation.

J. Hintsch, C. Görling, and K. Turowski. A review of the literature on configuration management tools. *CONF-IRM 2016 Proceedings*, 2016. URL https://core.ac.uk/download/pdf/301369226.pdf.

A. Jacob. Chef 0.6.0 Release, 2009. URL https://www.chef.io/blog/chef-0-6-0-release.

JetBrains. What are Domain-Specific Languages (DSL), 2025. URL https://www.jetbrains.com/mps/concepts/domain-specific-languages/.

S. Kadima. Ansible: History Of Ansible, 2023. URL https://medium.com/%40kadimasam/ansible-history-of-ansible-4b16208c9188.

L. Kanies. Episode 22: Puppet – DevOps, Security, and Cloud Automation with Luke Kanies, 2019. URL https://opensourceunderdogs.com/episode-22-puppet-devops-security-and-cloud-automation-with-luke-kanies.

R. O. Kostromin. Survey of software configuration management tools of nodes in heterogeneous distributed computing environment. In *International Workshop on Information, Computation, and Control Systems for Distributed Environments*, 2020. URL https://www.researchgate.net/publication/344967752_Survey_of_software_configuration_management_tools_of_nodes_in_heterogeneous_distributed_computing_environment.

M. Krause. Puppet Configuration Management: Exploring a Powerful Solution, 2024. URL https://datascientest.com/en/puppet-configuration-management-exploring-a-powerful-solution.

R. Kumar. What is Chef and How it works? An Overview and Its Use Cases, 2022. URL https://www.devopsschool.com/blog/what-is-chef-and-how-it-works-an-overview-and-its-use-cases/.

S. Likitha. Automation of Server Configuration Using Ansible. *International Journal for Research in Applied Science and Engineering Technology (IJRASET)*, 10:4109–4113, 2022. URL https://www.academia.edu/download/89100391/Automation_of_Server_Configuration_Using_Ansible.pdf.

C. M. Lonvick and T. Ylonen. The Secure Shell (SSH) Protocol Architecture. RFC 4251, 2006. URL https://www.rfc-editor.org/info/rfc4251.

S. Mbuguah, V. Mony, and G. Nyabuto. Architectural review of client-server models. *International Journal of Scientific Research and Engineering Trends*, 10:139–143, 01 2024. doi: 10.47679/ijasca.v3i1.48. URL https://www.researchgate.net/publication/376512127_Architectural_Review_of_Client-Server_Models.

R. Miller. Opscode Rebrands as Chef, Raises $32 Million, 2013. URL https://www.datacenterknowledge.com/investing/opscode-rebrands-as-chef-raises-32-million.

T. D. Nielsen, C. Iversen, and P. Bonnet. Private Cloud Configuration with MetaConfig. In *IEEE International Conference on Cloud Computing*, pages 508–515, 7 2011. doi: 10.1109/CLOUD.2011.63. URL https://www.researchgate.net/profile/Philippe-Bonnet-2/publication/221400024_Private_Cloud_Configuration_with_MetaConfig/links/00463517c3c55b0d7c000000/Private-Cloud-Configuration-with-MetaConfig.pdf.

S. Ninawe. Ansible Architecture: Key Components Overview, 2025. URL https://spacelift.io/blog/ansible-architecture.

T. Nolle. Ansible vs. puppet: Declarative devops tools square off, 2019. URL https://www.techtarget.com/searchsoftwarequality/tip/Ansible-vs-Puppet-Declarative-DevOps-tools-square-off.

OpenSSH Team. OpenSSH: Manual Pages, 2025. URL https://www.openssh.com/manual.html.

Painter Lee. The Benefits of SSH Key Authentication, 2023. URL https://jadaptive.com/ssh-key-management/the-benefits-of-ssh-key-authentication/.

PCMag. Definition of interpreter, 2025. URL https://www.pcmag.com/encyclopedia/term/interpreter.

Portnox. How Does SSH Passwordless Login Work?, 2025. URL https://www.portnox.com/cybersecurity-101/ssh-passwordless-login/.

K. Prabhu. How to Write an Ansible Playbook, 2024. URL https://digitalvarys.com/how-to-write-an-ansible-playbook/.

Progress. Chef Platform Overview, 2025a. URL https://docs.chef.io/platform_overview/.

Progress. Chef Infra Overview, 2025b. URL https://docs.chef.io/chef_overview/.

Progress. Chef Infra Server Overview, 2025c. URL https://docs.chef.io/server/#external-cookbooks.

Progress. Security, 2025d. URL https://docs.chef.io/server/server_security/.

Progress. RBAC: Organizations and Groups, 2025e. URL https://docs.chef.io/server/server_orgs/.

Progress. Chef Infra Client Overview, 2025f. URL https://docs.chef.io/chef_client_overview/.

Progress. About the Chef Infra Language, 2025g. URL https://docs.chef.io/infra_language/.

Progress. About Recipes, 2025h. URL https://docs.chef.io/recipes/.

Progress. Custom resource guide, 2025i. URL https://docs.chef.io/custom_resources/.

Progress. About Resources, 2025j. URL https://docs.chef.io/resource/.

Progress. About Cookbooks, 2025k. URL https://docs.chef.io/cookbooks/.

Progress. Knife Cloud Plugins, 2025l. URL https://docs.chef.io/workstation/plugin_knife/.

Progress. An Overview of Chef InSpec, 2025m. URL https://docs.chef.io/inspec/.

Progress. Chef Supermarket, 2025n. URL https://docs.chef.io/supermarket/.

Progress. Install Chef Workstation, 2025o. URL https://docs.chef.io/workstation/install_workstation/.

Progress. Chef Infra Server Prerequisites, 2025p. URL https://docs.chef.io/server/install_server_pre/.

Progress. Supported platforms, 2025q. URL https://docs.chef.io/platforms.

Progress. System Requirements, 2025r. URL https://docs.chef.io/chef_system_requirements/.

Progress. Progress Chef Enterprise Edition, 2025s. URL https://docs.chef.io/enterprise_chef/.

Progress. About the Chef Community, 2025t. URL https://docs.chef.io/community/.

Puppet. PE Console RBAC APIs, 2022. URL https://github.com/puppetlabs/pltraining-rbac.

Puppet. What is Puppet?, 2025a. URL https://help.puppet.com/core/current/Content/PuppetCore/what_is_puppet.htm.

Puppet. Overview of Puppet's architecture, 2025b. URL https://help.puppet.com/core/current/Content/PuppetCore/architecture.htm.

Puppet. The Puppet platform, 2025c. URL https://help.puppet.com/core/current/Content/PuppetCore/platform_components.htm.

Puppet. Facts and built-in variables, 2025d. URL https://help.puppet.com/core/current/Content/PuppetCore/lang_facts_and_builtin_vars.htm.

Puppet. The Puppet language, 2025e. URL https://help.puppet.com/core/current/Content/PuppetCore/puppet_language.htm.

Puppet. Puppet language overview, 2025f. URL https://help.puppet.com/core/current/Content/PuppetCore/intro_puppet_language_and_code.htm.

Puppet. Resource examples, 2025g. URL https://help.puppet.com/core/current/Content/PuppetCore/lang_examples_resource.htm.

Puppet. Key concepts behind Puppet, 2025h. URL https://help.puppet.com/core/current/Content/PuppetCore/key_concepts_puppet.htm.

Puppet. Conditional statements and expressions, 2025i. URL https://help.puppet.com/core/current/Content/PuppetCore/lang_conditional.htm.

Puppet. Relationships and ordering, 2025j. URL https://help.puppet.com/core/current/Content/PuppetCore/lang_relationships.htm.

Puppet. Templates, 2025k. URL https://help.puppet.com/core/current/Content/PuppetCore/lang_template.htm.

Puppet. Puppet Forge, 2025m. URL https://forge.puppet.com/.

Puppet. About Hiera, 2025n. URL https://help.puppet.com/core/current/Content/PuppetCore/hiera_intro.htm.

Puppet. Puppet Bolt, 2025o. URL https://www.puppet.com/docs/bolt/latest/bolt.html.

Puppet. Puppet reports, 2025q. URL https://help.puppet.com/core/current/Content/PuppetCore/reporting.htm.

Puppet. Installing Puppet Server, 2025r. URL https://help.puppet.com/core/current/Content/PuppetCore/server/install_from_packages.htm.

Puppet. System requirements, 2025s. URL https://help.puppet.com/core/current/Content/PuppetCore/system_requirements.htm.

Puppet. Hardware requirements, 2025t. URL https://help.puppet.com/core/current/Content/PuppetCore/hardware_requirements.htm.

Puppet. Firewall configuration, 2025u. URL https://help.puppet.com/core/current/Content/PuppetCore/firewall_configuration.htm.

Puppet. Supported agent platforms, 2025v. URL https://help.puppet.com/core/current/Content/PuppetCore/supported_operating_systems.htm.

Puppet. Timekeeping and name resolution, 2025w. URL https://help.puppet.com/core/current/Content/PuppetCore/timekeeping_and_name_resolution.htm.

PyNet Labs. Ansible Architecture Overview: Key Components & Workflow, 2025. URL https://www.pynetlabs.com/ansible-architecture/.

D. Rand. What is configuration management?, 2021. URL https://www.tanium.com/blog/what-is-configuration-management/.

M. Rastenis. *A study of bugs found in the Ansible configuration management system.* PhD thesis, Delft University of Technology, 6 2022. URL https://repository.tudelft.nl/file/File_29ed7643-5d10-479e-8a90-54a114dbd7f9?preview=1.

realtebo. Ansible requires python-apt but it's already installed, 2021. URL https://stackoverflow.com/a/51623235.

Red Hat. What is YAML?, 2023. URL https://www.redhat.com/en/topics/automation/what-is-yaml.

Red Hat. What is configuration management, 2023a. URL https://www.redhat.com/en/topics/automation/what-is-configuration-management.

Red Hat. Ansible Collaborative - How Ansible Works, 2025. URL https://www.redhat.com/en/ansible-collaborative/how-ansible-works.

C. Reiher, A. Elstad, and S. Veach. Puppet Manifests, 2023. URL https://docs.oracle.com/en/operating-systems/solaris/oracle-solaris/11.4/use-puppet/puppet-manifests.html.

S. Semushin. Install python3-dbus package instead of python-dbus when python3 is used · issue 3544 · openshift/openshift-ansible, 2017. URL https://github.com/openshift/openshift-ansible/issues/3544.

K. Sen. Declarative vs. Imperative Models for Configuration Management: Which Is Really Better?, 2025. URL https://www.upguard.com/blog/declarative-vs-imperative-models-for-configuration-management.

SSH. SSH command usage, options, and configuration in Linux/Unix, 2025. URL https://www.ssh.com/academy/ssh/command.

B. Steinglass. Opscode Closes $11 Million Series B Round, 2010. URL https://www.chef.io/blog/opscode-closes-11-million-series-b-round.

The Go Authors. Frequently Asked Questions (FAQ) - The Go Programming Language, 2025a. URL https://go.dev/doc/faq.

The Go Authors. Installing Go from source - The Go Programming Language, 2025b. URL https://go.dev/doc/install/source.

The Go Authors. template package - text/template - Go Packages, 2025c. URL https://pkg.go.dev/text/template.

The Go Authors. ssh package - golang.org/x/crypto/ssh - Go Packages, 2025d. URL https://pkg.go.dev/golang.org/x/crypto/ssh.

A. Velimirovic. Ansible vs. Chef: Differences Explained, 2024. URL https://phoenixnap.com/blog/ansible-vs-chef.

N. Vummadi. Aborting, target uses selinux but Python bindings (libselinux-Python) aren't installed - Stack Overflow, 2020. URL https://stackoverflow.com/q/63173955.

YAML Language Development Team. YAML Ain't Markup Language (YAML™) revision 1.2.2, 2021. URL https://yaml.org/spec/1.2.2/.