thePower

Project NEBULA: Plataforma de Micro-Servicios Autogestionada

2026

Índice

Abstract

Project NEBULA: A Self-Managed Cloud Platform for Remote Teams

This project solves a common problem for small tech teams and startups which is being locked into expensive monthly fees to big companies and having limited control over their own data and applications, for example, if you are a developer , each you time have to launch a website or an app you need to rent space from a big, expensive company, manually set up the servers and trust the company with all your code and user data.

To fix this, I built my own private, automated, computer system. It's like a secure, private internet office where a remote team can effortlessly automatically build, test, and launch their websites and applications without relying on expensive cloud services. I built it using open-source, free tools. It's a free alternative you fully control.

The platform runs on a secure Linux which becomes your “private cloud” server. It uses Docker to easily isolate, package and run all the different parts of an application (like a database or a web server). I set up automatic updates from a code repository (GitHub), so a simple code push can launch a new feature. For security, the platform provides free SSL certificates for safe HTTPS connections, a traffic detector (Nginx) that sends visitors to the right app and includes monitoring tools (Grafana) to monitor performance.

NEBULA is like a single, powerful computer that acts like a mini-Cloud after fitting free automation software. You can write the code, push it on GitHub, watch as the system automatically builds, secures and launches the app online while you’re monitoring everything from a simple dashboard.

The result is a fully functional, documented system that gives developers independence from big cloud vendors, reduces costs, and keeps their data under their own control. The project also outlines how this small system can grow into a larger, more powerful one in the future.

After ending the foundation (Docker Compose), it will be replaced with K3 (lightweight version of Kubernetes) to scale automatically if the webs get lots of traffic, useful for larger companies.

Resumen y objetivos

Actualmente, la proliferación del trabajo distribuido ha aumentado la dependencia de herramientas colaborativas y plataformas de desarrollo. Sin embargo, muchas de estas soluciones se basan en servicios de software como servicio (SaaS) proporcionados por grandes proveedores como Amazon Web Services (AWS) o Microsoft Azure, que, aunque ofrecen escalabilidad y facilidad de uso, implican costos recurrentes que pueden ser prohibitivos para equipos con presupuestos limitados, Y suelen generar dependencia ya que las habilidades adquiridas tienen transferencia limitada y la migración puede ser muy lastrada, además de generar preocupaciones sobre la soberanía de los datos [1]. Datos sensibles, como código fuente, datos de usuarios o información propietaria, quedan bajo el control de terceros, exponiéndose a riesgos de privacidad, cumplimiento normativo y brechas de seguridad inminentes.

NEBULA aborda este problema proporcionando una plataforma de micro-servicios autogestionada basada en infraestructura de nube privada. Deja a equipos remotos de desarrolladores desplegar aplicaciones web automáticamente con pipelines de integración y despliegue (CI/CD)[1] , todo esto sin incurrir en los costos altos propios de los servicios comerciales. Enfocado en el uso de código abierto para garantizar control total sobre los datos y la infraestructura, promoviendo la soberanía digital.

Elementos clave:

- Despliegue automático: Usa Docker y Docker Compose para la contenedorización, y GitHub Actions para la automatización de flujos de trabajo,facilita el despliegue de aplicaciones sin intervención manual constante.
- Seguridad perimetral: Implementa herramientas como Nginx como reverse proxy, Fail2Ban para detectar intrusiones, y certificados SSL mediante Let's Encrypt, asegura comunicaciones seguras y protección contra ataques.
- Monitorización: Integra stack basado en Prometheus y Grafana para el seguimiento en tiempo real del rendimiento, recursos y alertas, permitiendo la gestión de la infraestructura.
- **Redes seguras**: Incorpora Pi-hole como DNS seguro para bloquear trackers y malware, mejorando la privacidad en entornos remotos.

[1] Práctica donde el código se construye y despliega automáticamente al detectar cambios.

Así NEBULA no solo resuelve la dependencia de SaaS costosos, sino que da poder a pequeños equipos al brindar una alternativa escalable, segura y de bajo costo. El proyecto se alinea con tendencias en DevOps y cloud-native computing, donde la autogestión es clave para la innovación sostenible. NEBULA transforma la vulnerabilidad de la dependencia externa en una fortaleza, la autonomía.

Objetivos

Los objetivos del proyecto son claros y coherentes. Se dividen en objetivos generales, el propósito amplio del proyecto, y objetivos específicos, las metas técnicas y funcionales. Todos los objetivos siguen el criterio SMART (Específico; Medible; Alcanzable; Relevante; Limitado en tiempo), para que sean prácticos y evaluables.

Objetivos Generales

- **Crear una plataforma autogestionada de micro-servicios para equipos remotos**: desarrollar una infraestructura completa que permita el despliegue y gestión de aplicaciones web sin depender de proveedores externos. Se va a medir que al menos tres aplicaciones funcionen de manera estable en el entorno; alcanzable con tecnologías open-source; relevante para resolver problemas de soberanía de datos en startups; y limitado en tiempo al marco del proyecto (3 meses de desarrollo).

Objetivos Específicos

- Desplegar una base de contenedores con Docker y Docker Compose: Instalar y configurar Docker en un servidor Linux (Ubuntu), desplegando al menos tres aplicaciones de ejemplo (ej: una web estática, API REST y una base de datos). Medible por el número de contenedores activos y logs de ejecución; alcanzable con documentación oficial de Docker; relevante para la base obligatoria del proyecto; y completado en las primeras 4 semanas.
- Implementar un proxy inverso con seguridad SSL: Configurar Nginx como reverse proxy para enrutar tráfico a las aplicaciones, integrando certificados SSL de Let 's Encrypt. Se mide con pruebas de acceso HTTPS seguro y validación con herramientas como SSL Labs; alcanzable en entornos locales/remotos; relevante para la protección perimetral; y finalizado en la semana 6.

- Establecer monitorización básica: Desplegar Prometheus y Grafana para monitorear métricas de CPU, memoria y red en los contenedores. Medible por paneles funcionales y alertas configuradas; alcanzable con stacks preconfigurados; útil para el mantenimiento proactivo; implementado al final.
- Incorporar automatización CI/CD: Usar GitHub Actions para flujos automáticos de build y deploy, y Git local para control de versiones. Medible por ejecuciones exitosas de pipelines; alcanzable por integración con SSH para accesos remotos; útil para equipos distribuidos; y en pruebas.
- Evaluar y documentar la escalabilidad hacia K3s: Migrar una o dos aplicaciones a un clúster K3s ligero, demostrando auto-escalado básico. Medible por diagramas y pruebas; alcanzable como plus; útil para evolución a Kubernetes; y documentado en el futuro.

Los objetivos específicos contribuyen directamente a los generales. Se enlazan con la base del proyecto, cubriendo Docker + Docker Compose, 3 apps en ejecución, proxy inverso + SSL, monitorización básica y documentación. El plus de K3s se trata como una evolución, sin comprometer el núcleo.

Tabla de Objetivos vs. Metas

Objetivo	Meta Específica	Indicador SMART	Enlace con Base Obligatoria
General: Plataforma autogestionada	Desplegar infraestructura completa	3 apps corriendo, monitorización (uptime >95%; en local/remoto)	Cubre Docker, apps, proxy, SSL, monitorización
Específico: Base de contenedores	Configurar Docker Compose con 3 apps	Contenedores up y accesibles vía puerto (medible por docker ps; alcanzable en 4 semanas)	Docker + Docker Compose; 3 aplicaciones funcionando

Específico: Proxy y SSL	Implementar Nginx + Let's Encrypt	Certificado válido, tráfico HTTPS (medible por curl tests; alcanzable con certbot)	Proxy inverso + SSL
Específico: Monitorización	Stack Prometheus + Grafana	Paneles con métricas reales (por queries exitosas; alcanzable con helm charts)	Monitorización básica
Específico: CI/CD	Pipelines en GitHub Actions	auto-deploys en push (medible por logs de actions; alcanzable vía SSH keys)	Documentación completa (incluye automatización)
Específico: Escalabilidad a K3s	Migración teórica/práctica	Diagramas y auto-escalado demo (medible por pods en K3s; como plus)	Plus K3s

Motivaciones

La motivación para el desarrollo de NEBULA es la necesidad de competencias en entornos laborales remotos, donde la gestión de infraestructuras distribuidas es esencial. Se enfatiza el dominio de la infraestructura como código, que permite definir y provisionar recursos mediante código versionado, reduciendo errores humanos y facilitando la colaboración. Herramientas como SSH para accesos seguros a servidores y Git para el control de versiones y flujos colaborativos son esenciales, ya que simulan escenarios reales en startups donde los devs trabajan desde ubicaciones dispersas.

El proyecto motiva la adopción de prácticas sostenibles en TI, promueve el uso de software libre para evitar **vendor-lock-in**[2] y fomentar la innovación accesible. En un

[2] Dependencia de un proveedor que dificulta cambiar de servicio sin costes o complicaciones.

mercado laboral cada vez más orientado a DevOps y cloud computing, competencias en estas áreas no sólo resuelven problemas inmediatos, sino que preparan para roles en empresas como startups tech o consultorías de TI remota. Finalmente, NEBULA incentiva la autoaprendizaje y la resiliencia, al requerir resolución de problemas reales como configuración de redes virtuales y hardening de servidores, alineándose con los objetivos educativos del ciclo formativo.

Antecedentes

Actualmente, los equipos de desarrollo que necesitan desplegar aplicaciones web pueden usar tres tipos de soluciones:

Descripcion y analisis de soluciones existentes similares

1. **Plataforma como Servicio (PaaS)**
   como Heroku o AWS Elastic Beanstalk, donde el desarrollador sube su código y la plataforma se encarga directamente del servidor, contenedores, escalado, SSL y despliegue. otorga máxima simplicidad ya que no requiere de configuración de infraestructura,
   Sin embargo esta solución tiene un costo elevado ya que los precios escalan muy rápido con el tráfico, tu aplicación queda diseñada para servicios específicos del proveedor por lo que si deseas migrar es difícil y costoso, no se puede personalizar la infraestructura subyacente y los costes son impredecibles.

2. **Infraestructura como Servicio (IaaS)**
   como EC2, Google Compute Engine o DigitalOcean Droplets, aquí el usuario alquila un servidor virtual VPS y tiene control total sobre él, por consiguiente requiere de instalación y configuración manual de todo el software.
   Otorga control total y flexibilidad completa, solo pagas por uso.
   Sin embargo esta solución requiere conocimientos avanzados de administración de sistemas o DevOps, da responsabilidad total de la seguridad, parches y backups, conlleva gastos ocultos como el ancho de banda , IPs estáticas y snapshots que se suman rápidamente y tiene una curva de aprendizaje empinada que varía por proveedor.

3. **Soluciones autogestionadas tradicionales**
   como lo seria un servidor dedicado fisico, un stack manual de LAMP/LEMP o un cPanel, donde, normalmente, se compra hardware físico, se instala un sistema operativo y se configura manualmente todos los servicios, (Apache, MySQL, PHP etc…) Con esta solución tienes soberanía de datos absoluta y costos que son predecibles a largo plazo.

> **Sin embargo** esta solución requiere de un tiempo de configuración alto, la escalabilidad es complicada, la automatización es limitada y propensa a errores, y requiere de conocimientos especializados en administración de sistemas.

También hay alternativas open-source. (**Gitea, GitLab o Jenkins**,) el usuario instala y gestiona sus herramientas de desarrollo, control de versiones y despliegue en su infraestructura. En lugar de depender de terceros, se monta un ecosistema privado para controlar el software. Otorga privacidad del código fuente, elimina las cuotas por usuario y permite integración con redes locales o servidores privados.
Sin embargo, esta solución requiere un esfuerzo inicial desproporcionado para configurar el flujo de trabajo comparado con soluciones listas para usar. y es más difícil conectar estas herramientas con servicios externos o APIs de terceros que ya vienen preconfiguradas en las grandes plataformas.

Justificación de NEBULA, El Término Medio Ideal

¿Qué deficiencias cubre?

Elimina la dependencia económica que conllevan los costos recurrentes de PaaS/SaaS (exceptuando el servidor base).

Ofrece el control casi total de la IaaS y con la automatización de PaaS.

Las habilidades requeridas y aprendidas con NEBULA son completamente transferibles a cualquier entorno de nube (Docker, Kubernetes, CI/CD).

¿Qué nuevas funcionalidades aporta?

un sistema de CI/CD auto-hospedado fácil de entender y simple de modificar.

Es un stack unificado que integra y documenta todas las herramientas (monitorización, proxy, contenedores).

está diseñado para evolucionar a Kubernetes,

Característica	NEBULA (mi Proyecto)	PaaS (Heroku)	IaaS (EC2)	Solución Tradicional
Costo Mensual	Muy Bajo (servidor base)	Alto (escala con uso)	Medio-Alto (costes ocultos)	Muy Bajo
Configuración Inicial	**Media** (script automatización)	Mínima (solo código)	Alta (todo manual)	Muy Alta (todo desde cero)
Mantenimiento	**Moderado** (automatizado)	Cero	Alto	Muy Alto (manual)
Control	**Total**	Muy Bajo	Alto	Total
Escalabilidad	**Preparada** (para K8s)	Automática (cara)	Manual/compleja	Muy Difícil

Curva de Aprendizaje	**Medio-Alta** (transferible)	Baja	Muy Alta	Alta
Vendor Lock-in	**Cero** (OpenSource)	Muy Alto	Medio (API específicas)	Bajo
Soberanía de Datos	**Total** (tus servidores)	Baja (servidores del proveedor)	Media (centros de datos)	Total
Automatización CI/CD	**Integrada y Auto-Hospedada**	Integrada	Configurable manualmente	Manual
Monitorización	**Integrada** (Grafana/Prometheus)	Limitada/Básica	Por agregar (costosa)	Manual/por agregar
Mejor para...	**Equipos pequeños/medianos**	Startups con funding	Grandes empresas con equipo DevOps	Empresas con infraestructura establecida

NEBULA no pretende reemplazar a AWS o Heroku para todos los casos, sino que ofrece una **alternativa viable para un nicho específico** formado por equipos pequeños o medianos, startups bootstrapped, proyectos educativos, o cualquier situación donde el balance entre control, costo y aprendizaje sea prioritario ante la simplicidad absoluta.

La principal innovación de NEBULA es su **enfoque educativo y transparente**: mientras que los proveedores cloud ocultan la complejidad (haciéndote dependiente), NEBULA la hace **accesible** y **comprensible [2]**, empoderando a los desarrolladores con habilidades reales de infraestructura que son valiosas en el mercado laboral actual.

Analisis y especificacion de requisitos

En esta sección se detallan los requisitos del proyecto NEBULA, tanto funcionales como técnicos.Se identifican las entradas, procesos y salidas necesarias para su funcionamiento, se especifican los componentes técnicos requeridos y se agrupan las tareas en fases con estimaciones de tiempo, esfuerzo y dificultad. Esto asegura que el proyecto sea completo, identificando entradas, procesos y salidas, agrupando tareas coherentemente con los objetivos y estimando tiempos y dificultades de forma razonable.
Los requisitos se basan en el problema principal: proporcionar una infraestructura autogestionada para equipos remotos, con énfasis en la base (Docker + Docker Compose, 3 aplicaciones, proxy inverso + SSL, monitorización básica) y el plus de K3s como evolución. Se consideran tres aplicaciones de ejemplo para ilustrar los requisitos: una web simple (ej, sitio estático con HTML/CSS servido por Nginx), una API REST (ej. implementada en Flask o Node.js para endpoints básicos) y una base de datos (ej. PostgreSQL).

Requisitos Funcionales del Sistema y de infraestructura

describen qué debe hacer el sistema, enfocándose en entradas (datos o triggers que inician procesos), procesos (acciones transformadoras) y salidas (resultados). Se priorizan según su impacto: alta (esencial para el núcleo), media (mejora la usabilidad) y baja (opcional).

Requisito Funcional 01 (Alta prioridad): Despliegue automático de aplicaciones web al detectar cambios en GitHub.

- Entradas: Código fuente de las aplicaciones (push en repositorio Git con archivos para la web simple, API o DB).
- Procesos: Webhook detecta cambios, activa pipeline CI/CD en GitHub Actions, construye imágenes Docker y despliega contenedores vía Docker Compose.
- Salidas: Aplicaciones desplegadas y accesibles vía HTTPS (e.g., web simple en puerto mapeado, API respondiendo a requests, DB lista para queries). Esto asegura CI/CD para las tres apps, con aislamiento en contenedores separados.

Requisito Funcional 02 (Alta prioridad): Ejecución simultánea de al menos 3 aplicaciones.

- Entradas: Configuraciones YAML en Docker Compose para cada app (ej. volúmenes persistentes en la DB).
- Procesos: Análisis de configuración, asignación de recursos (puertos, volúmenes) y aislamiento de contenedores.
- **Salidas**: 3 aplicaciones corriendo en paralelo (web sirviendo páginas, API manejando peticiones, DB almacenando datos), con logs accesibles. Requiere hardware mínimo para operación estable.

Requisito Funcional 03 (Alta prioridad): Acceso seguro a aplicaciones mediante HTTPS con certificados SSL.

- **Entradas**: Dominios configurados en Nginx.
- **Procesos**: Obtención y renovación automática de certificados vía Certbot de Let's Encrypt, integración en reverse proxy.
- **Salidas**: Tráfico encriptado (con candado verde en navegador).

Requisito Funcional 04 (Media prioridad): Monitorización en tiempo real de estado, rendimiento y logs.

- **Entradas**: Métricas de contenedores (CPU, RAM, red).
- **Procesos**: Recolección con Prometheus, procesamiento y visualización en Grafana.
- **Salidas**: Dashboards interactivos (alerta si una app excede 80% CPU). Puertos mínimos: 22 (SSH), 80 (HTTP redirigido), 443 (HTTPS).

Requisito Funcional 05 (Media prioridad): Resolución de dominios internos y bloqueo de anuncios/malware.

- **Entradas**: Consultas DNS desde dispositivos en la red.
- **Procesos**: Filtrado con Pi-hole, resolución local y cacheo.
- **Salidas**: Respuestas DNS seguras, con trackers bloqueados.

Requisito Funcional 06 (Alta prioridad): Administración remota segura vía SSH.

- **Entradas**: Claves SSH del administrador.
- **Procesos**: Autenticación sin contraseña, ejecución de comandos.
- **Salidas**: Acceso controlado al servidor (e.g., para actualizar configs).

Requisito Funcional 07 (Media prioridad): Backups periódicos de configuraciones críticas.

- **Entradas**: Archivos de config (Docker YAML, Nginx conf).
- **Procesos**: Programación con cron, compresión/cifrado, envío a almacenamiento externo (e.g., S3 compatible).
- **Salidas**: Archivos de backup restaurables.

Requisitos Técnicos

especifican el entorno necesario para implementar los funcionales, incluyendo sistema operativo,

Sistema Operativo: Debian/Ubuntu Server (versión LTS[3], Ubuntu 22.04), por su estabilidad, soporte para contenedores y herramientas de seguridad nativas. recomiendo cambiar el kernel Kernel problemático 6.8.0-94 y usar un kernel estable como 6.8.0-90 permanentemente ya que es mas compatible con k3s y no genera problemas con las CPU.

Hardware Mínimo:

- CPU: 2 núcleos (Intel/AMD equivalente).
- RAM: 4 GB (suficiente para 3 apps y servicios).
- Almacenamiento: 20 GB SSD (para imágenes Docker y datos persistentes).
- Red: Conexión estable con ancho de banda >10 Mbps para CI/CD remoto. Esto soporta las tres apps de ejemplo sin sobrecarga.

Dependencias y Componentes de Software:

[3] versión de software con soporte y actualizaciones durante un largo periodo de tiempo.

Componente	Versión Mínima	Función
Docker Engine	24.0+	Contenerización de aplicaciones
Docker Compose	2.20+	Orquestación multi-contenedor
Nginx	1.18+	Reverse Proxy y balanceador
Certbot	2.0+	Certificados SSL Let's Encrypt
Prometheus	2.45+	Recolección de métricas
Grafana	10.0+	Visualización de métricas
Pi-hole	FTL 5.0+	DNS y filtrado de contenido
Fail2Ban	0.11+	Protección contra ataques brute-force
Git	2.25+	Control de versiones local

Requisitos de Rendimiento

ID	Métrica	Valor Objetivo	Método de Verificación
RT-13	Tiempo de Despliegue	< 5 minutos desde push a Git	Medición con GitHub Actions
RT-14	Disponibilidad	99% (tiempo de actividad)	Monitorización con Prometheus
RT-15	Latencia HTTP	< 200ms para respuestas estáticas	Pruebas con Apache Bench (ab)
RT-16	Uso de Recursos	< 80% CPU/RAM en operación	Panel Grafana

Requisitos de Seguridad:

- Aislamiento de aplicaciones mediante contenedores Docker separados (namespaces para web, API, DB).
- Cifrado de datos en tránsito web mediante TLS.
- Autenticación SSH sin contraseña mediante claves públicas Ed25519.
- Protección contra DDoS básica con limitación de solicitudes en Nginx.
- Bloqueo de IPs maliciosas con reglas de Fail2Ban.

Tareas y Planificación

Las tareas se agrupan en fases coherentes con los objetivos, estimando tiempos, esfuerzo (horas aproximadas) y dificultad (escala 1-10: 1=fácil, 10=alta complejidad técnica). Se usa un diagrama Gantt simplificado como tabla para visualización. El total estimado es ~12 semanas, ajustable por desviaciones.

Fase	Tareas Principales	semanas	Esfuerzo (Horas)	Dificultad (1-10)
1. Análisis	Definir topología de red virtual; identificar requisitos para las 3 apps.	1	20	4 (análisis conceptual).
2. Despliegue Base	Instalación de Ubuntu; hardening (firewall, usuarios); setup SSH.	2	30	6 (configuración segura).
3. Capa de Contenedores	Instalar Docker/Compose; configurar YAML para web simple, API (Flask) y DB (PostgreSQL); pruebas de aislamiento.	2	40	7 (integración multi-app).
4. Servicios	Configurar Nginx + SSL; integrar Pi-hole; setup CI/CD con GitHub Actions.	3	50	8 (automatización remota).

5. Monitorización y Seguridad	Desplegar Prometheus/Grafana; configurar Fail2Ban; backups iniciales.	2	30	5 (stacks preconfigurados).
6. Pruebas	Tests de carga (ab tool); seguridad (escaneos); verificación de 3 apps running.	2	25	6 (simulación real).

Línea de Tiempo del Proyecto NEBULA [4]

ESTIMACION DE TIEMPO

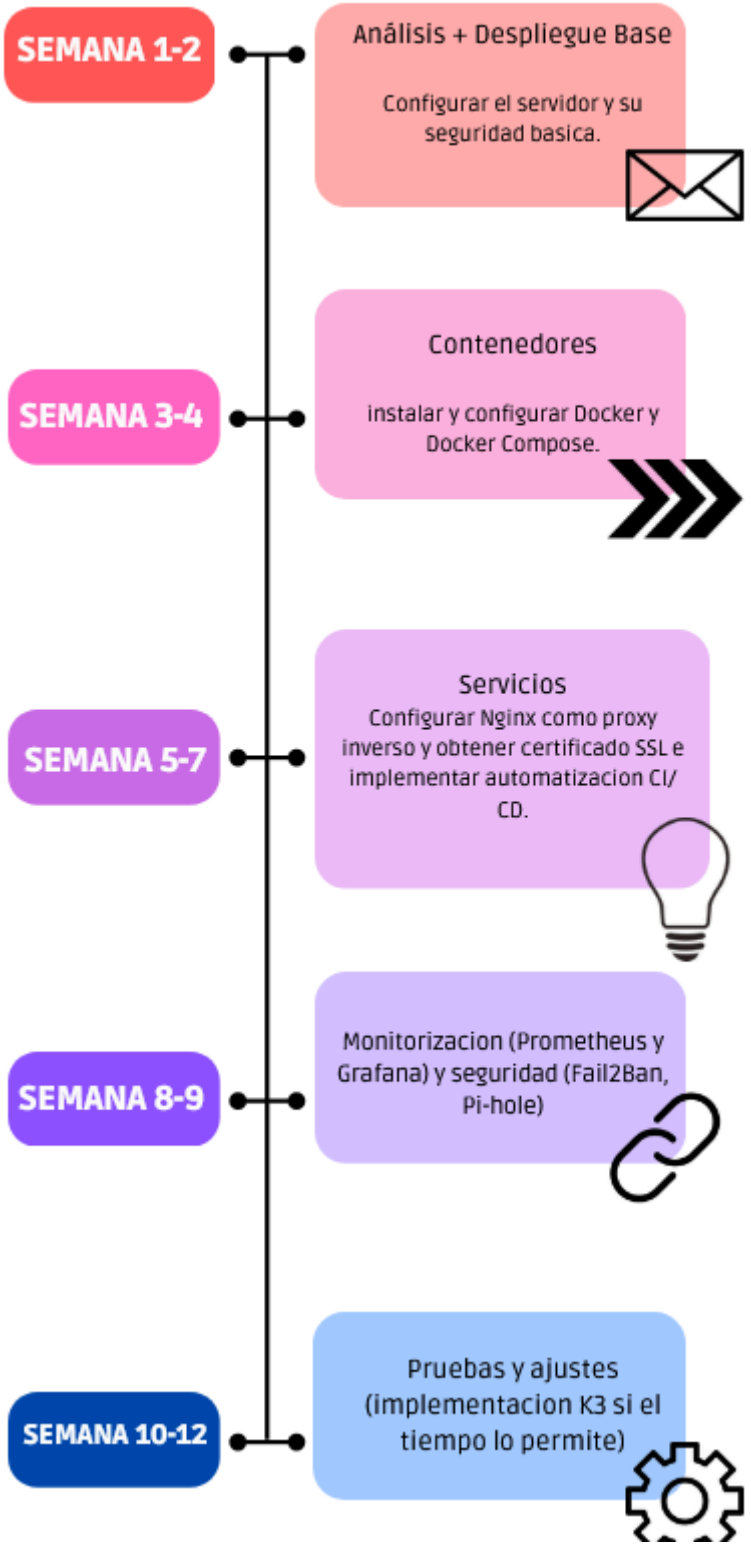


Diagrama de casos de Uso

identifica actores y funcionalidades principales. Actores: Desarrollador (usuario remoto), Administrador (gestor de infra). Casos de uso: Desplegar App (incluye CI/CD), Monitorizar Sistema, Acceder Seguro (HTTPS/SSH), Realizar Backup.

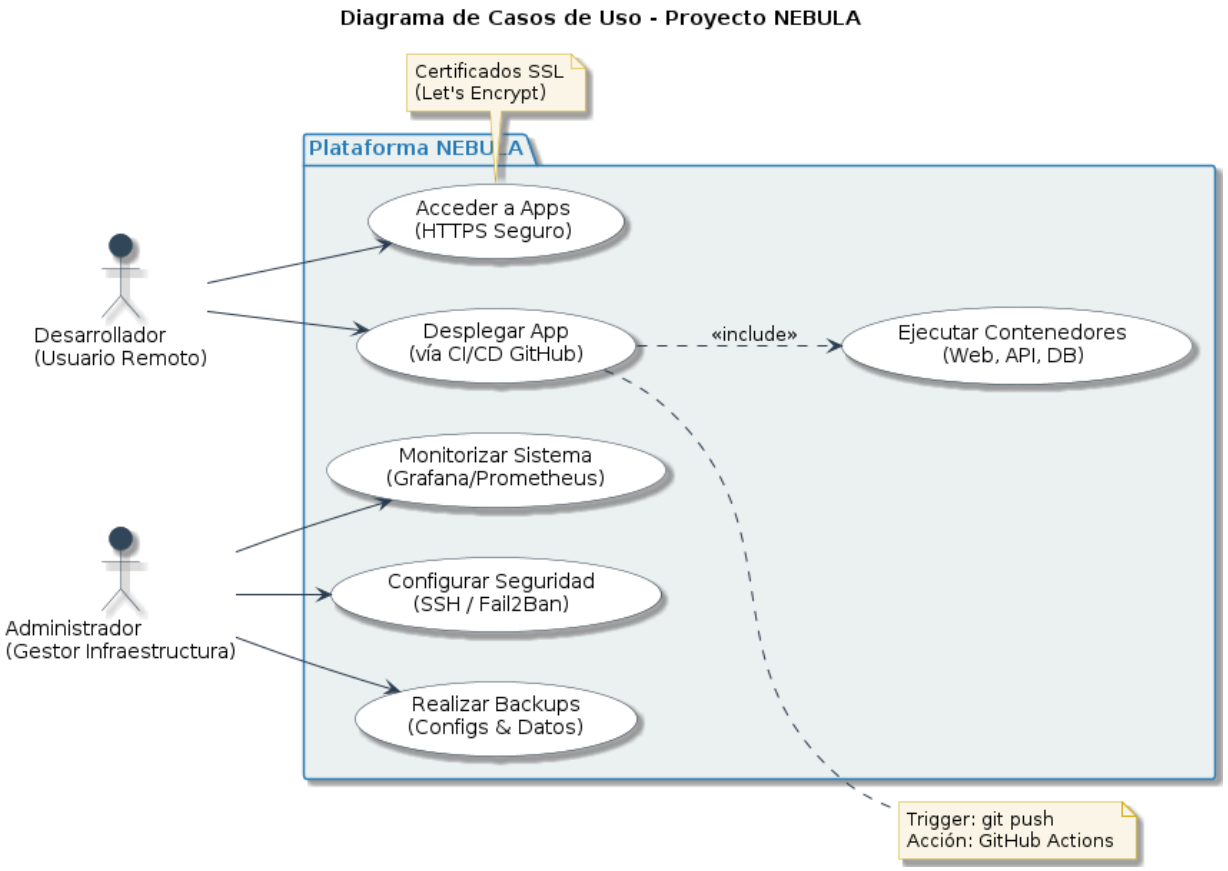
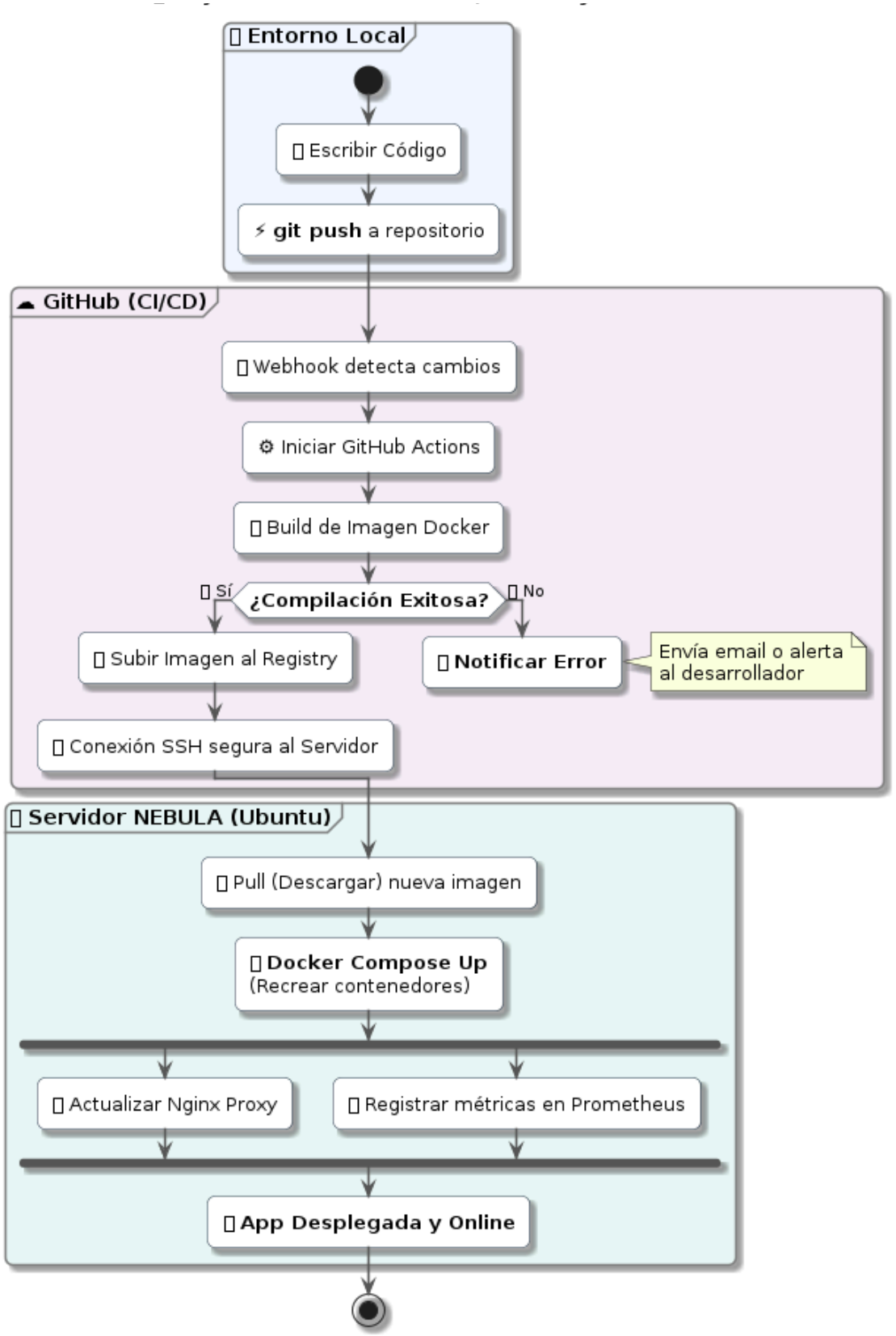


Diagrama de Flujos de Actividad [3]



el flujo para desplegar las tres apps, con ramificaciones para errores.

Propuesta de solución

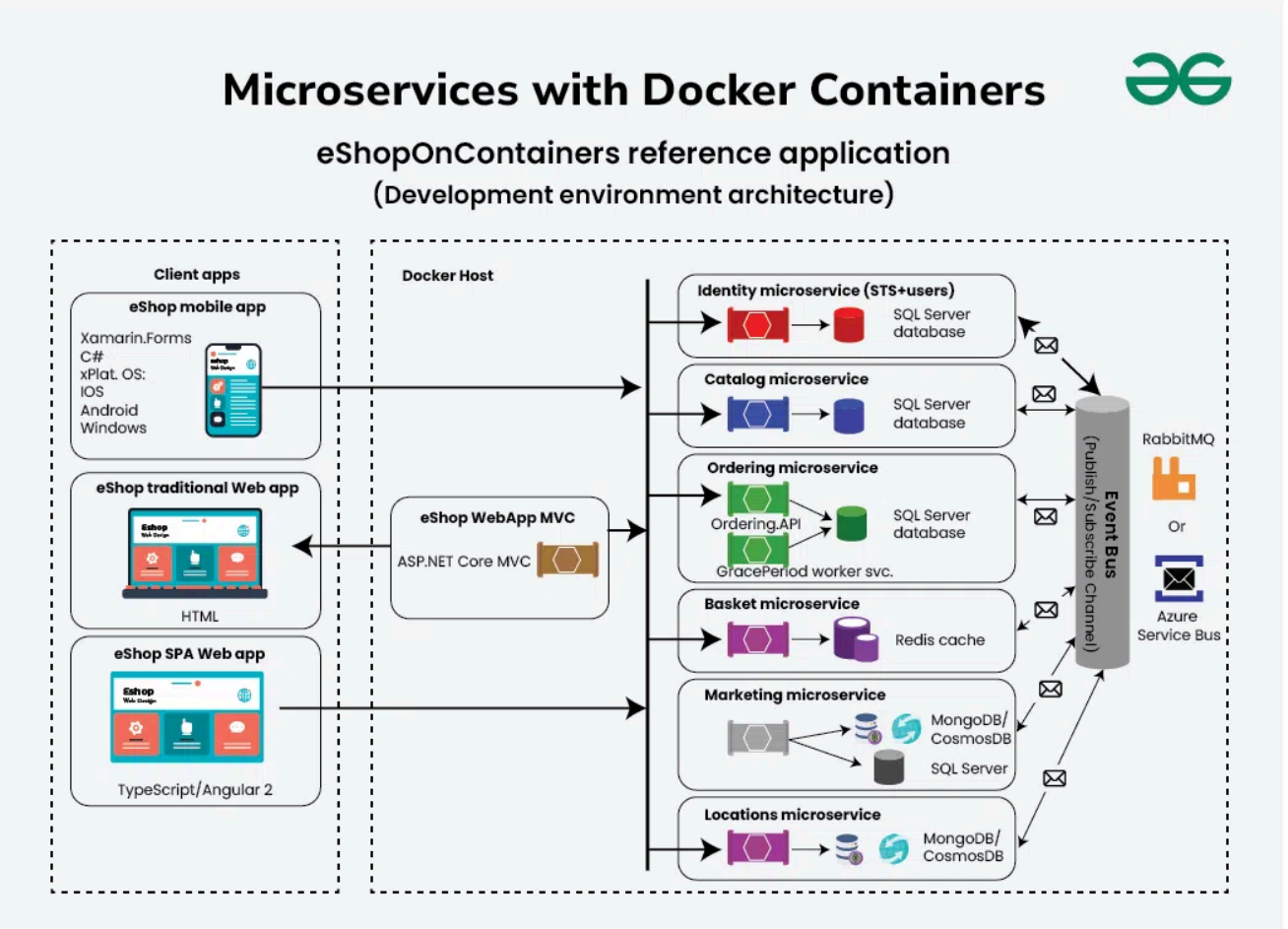
En esta sección se presenta la propuesta de solución técnica para el proyecto NEBULA, que consiste en una plataforma de micro-servicios autogestionada para equipos remotos. El diseño se representa mediante diagramas y esquemas gráficos, justificando la elección de tecnologías y metodologías empleadas. La solución es viable, funcional y escalable, con recursos proporcionales al alcance del problema (CE4.2), y se ajusta al nivel de complejidad adecuado para el ciclo formativo (CE4.4). Se enfatiza la base obligatoria en Docker Compose, con un "plus" teórico en K3s para demostrar evolución futura. Todos los elementos se alinean con el Resultado de Aprendizaje RA4, representando la solución gráficamente (CE4.1) y justificando su adecuación tecnológica (CE4.3).

Diseño de la Arquitectura

La arquitectura de NEBULA se basa en una infraestructura de nube privada, utilizando contenedores para aislar aplicaciones y servicios. La topología de red virtual se diseña para entornos remotos, con un servidor central (físico o VPS) que actúa como host, accesible vía SSH. Los componentes clave incluyen:

- **Capa Base**: Sistema operativo Linux (Debian/Ubuntu) con hardening de seguridad.
- **Capa de Contenedores**: Docker para empaquetar aplicaciones, y Docker Compose para orquestar múltiples servicios en un solo host.
- **Capa de Servicios**: Nginx como reverse proxy, Pi-hole para DNS seguro, stack de monitorización (Prometheus + Grafana), y herramientas de seguridad (Fail2Ban, Let's Encrypt).
- **Capa de Automatización**: GitHub Actions para CI/CD, integrado con Git local.
- **Evolución Plus**: K3s para un clúster ligero de Kubernetes, permitiendo migración de apps y auto-escalado básico.

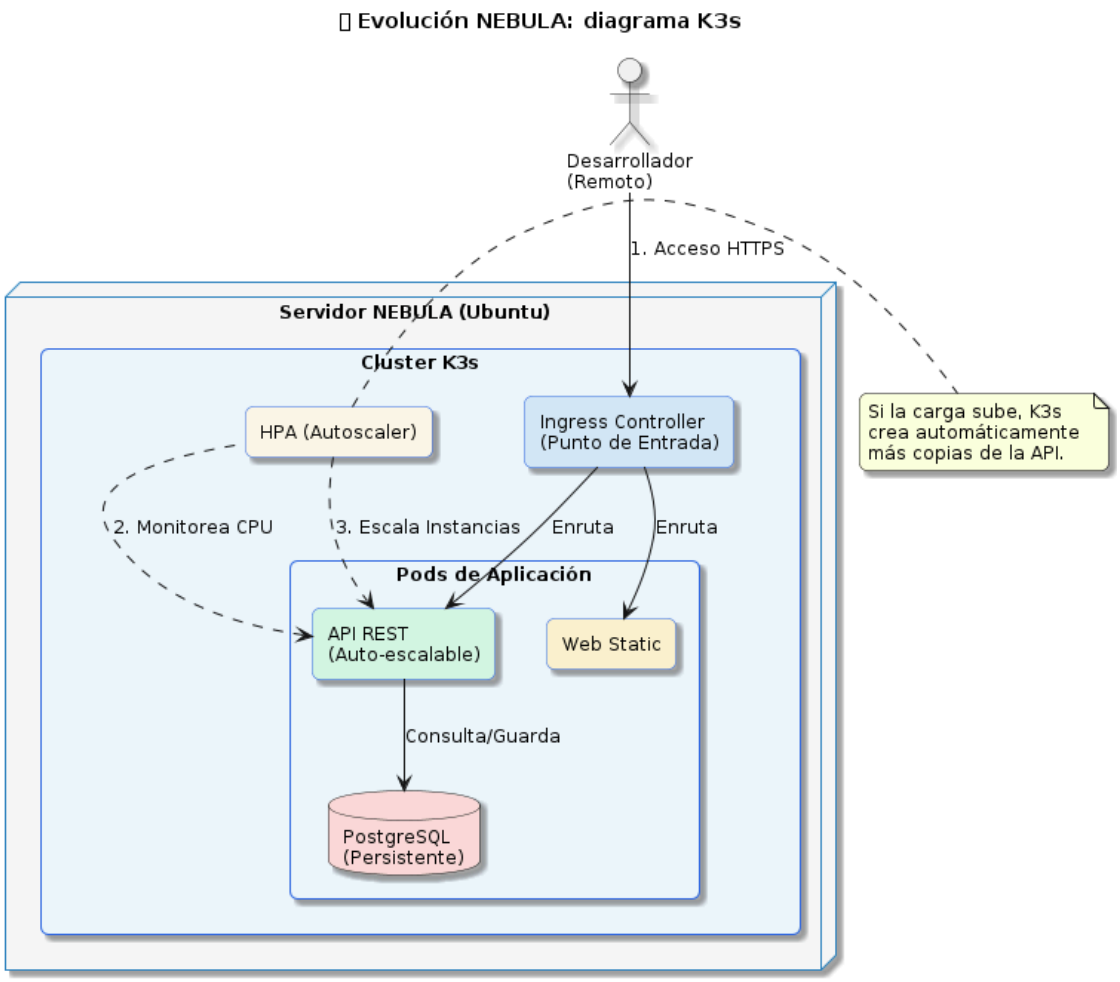
La arquitectura es modular, permitiendo escalabilidad horizontal (añadir nodos en K3s) sin comprometer la simplicidad para equipos pequeños. Se mostrará un diagrama representativo de la arquitectura con Docker Compose para microservicios.



[5] Este diagrama ilustra cómo las aplicaciones (e.g., web simple, API REST, base de datos) se despliegan como contenedores independientes en un host Docker, conectados a través de redes virtuales internas. Para la topología de red virtual, se define una red bridge en Docker Compose, con subredes aisladas (e.g., 172.20.0.0/16), donde cada contenedor tiene su IP virtual. El tráfico externo entra vía Nginx en puerto 443 (HTTPS), redirigido a los contenedores internos (puertos 80/8080). Esto minimiza la exposición, enfatizando seguridad con Fail2Ban monitoreando logs de acceso y bloqueando IPs sospechosas.

Para el "plus" en K3s, la arquitectura evoluciona a un clúster single-node o multi-node ligero. K3s, una distribución minimalista de Kubernetes, reduce la complejidad (sin etcd completo, usando SQLite), ideal para SMR. El diagrama conceptual muestra un master node con pods para las apps, services para exposición y auto-escalado vía Horizontal Pod Autoscaler (HPA). Aunque teórico en esta fase, se demuestra viabilidad migrando 1-2 apps (e.g., API a un Deployment en K3s).

diagrama K3s



justificación de la Elección de Tecnologías

La selección de tecnologías se basa en criterios de simplicidad, costo cero (open-source), compatibilidad con entornos remotos y alineación con requisitos.

- Docker y Docker Compose: Elegidos por su simplicidad en contenedorización. Docker permite empaquetar apps con dependencias (e.g., Node.js para API), asegurando portabilidad. Compose orquesta multi-contenedores con un YAML simple, ideal para base obligatoria (3 apps running). Vs. alternativas como Podman, Docker es más maduro y compatible con CI/CD. Justificación: Reduce complejidad para startups sin expertise en ops, con viabilidad probada en entornos locales/remotos.
- K3s (plus): Versión ligera de Kubernetes, vs. full K8s (muy pesado para SMR). K3s[4] usa 40% menos recursos, instalable en un comando, perfecto para

[4] Versión más ligera y simplificada de Kubernetes.

auto-escalado básico (e.g., replicas: 3 si carga alta). Justificación: Escalable para crecimiento (añadir HA), pero teórico aquí para no exceder tiempo; demuestra evolución sin comprometer núcleo.

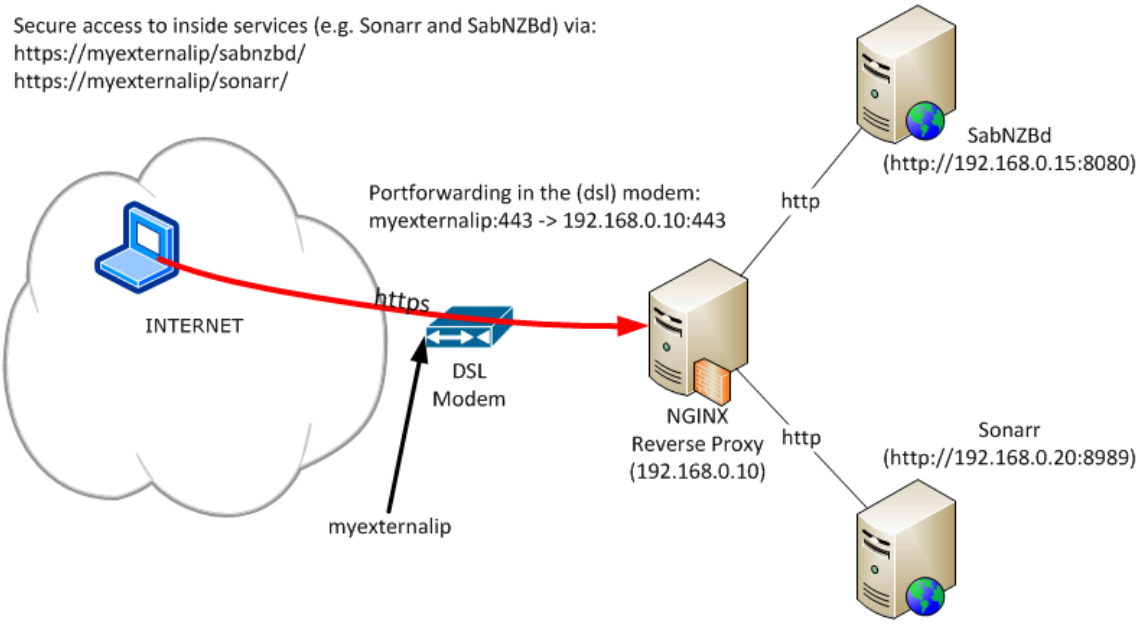
- Nginx: Reverse proxy por su eficiencia y soporte SSL. Integra Let's Encrypt para certs gratuitos, renovables automáticamente. Justificación: Proporcional al alcance (maneja tráfico para 3 apps), con seguridad contra ataques (rate limiting).
- GitHub Actions + Git: Para CI/CD, gratuito y integrado con repos remotos. Justificación: Facilita colaboración en equipos distribuidos vía SSH.
- Prometheus + Grafana: Stack estándar para monitorización. Prometheus scrape métricas, Grafana visualiza. Justificación: Básico pero potente, con alertas para mantenimiento proactivo.
- Seguridad (Fail2Ban, Let's Encrypt): Fail2Ban protege contra brute-force en SSH/HTTP, mientras Let's Encrypt asegura TLS. Justificación: Esencial para soberanía de datos, viables en setups low-cost.

La viabilidad se evalúa en escalabilidad: Base en single-host (para 3 apps), expandable a K3s para 10+ apps. Recursos proporcionales: 4GB RAM inicial, escalable a cluster. Complejidad adecuada: Docker para aprendizaje básico, K3s teórico para advanced concepts sin implementación full.

Gráficos y Esquemas

Esquema De Reverse Proxy (Nginx)

Nginx actúa como gateway, enrutando tráfico a contenedores. Flujo: Usuario --> HTTPS (443) --> Nginx (verifica SSL) --> Contenedor interno (e.g., web:80). Enfatiza seguridad con Fail2Ban integrando logs de Nginx.

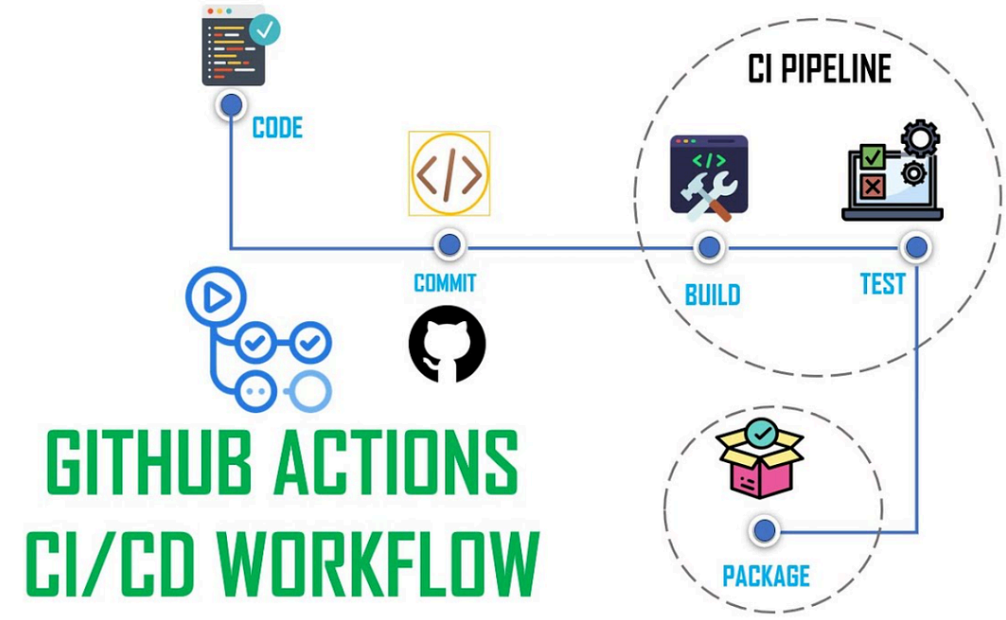


[6]

Este esquema muestra el proxy inverso seguro, con redirección HTTP a HTTPS via Let's Encrypt.

Flujo CI/CD (GitHub Actions)

El flujo automatiza deploys: Push a Git --> Webhook --> Actions (build Docker image) --> SSH al server --> docker-compose up. Para las 3 apps.

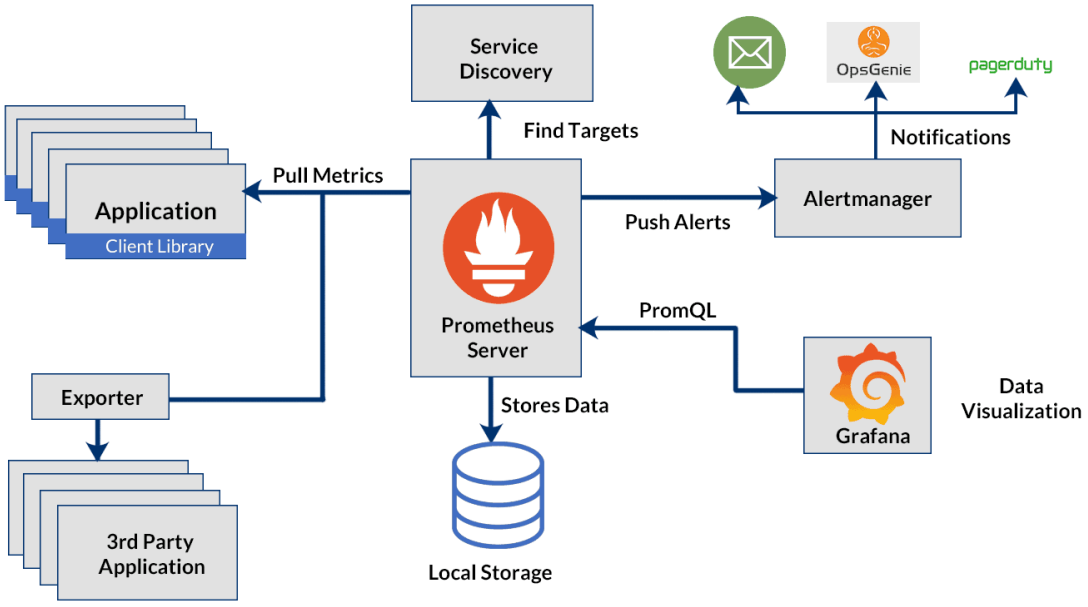


[7]

Commit --> Build/Test --> Deploy, integrable con K3s (kubectl apply).

Stack de Monitorización (Prometheus + Grafana)

Prometheus recolecta métricas de exporters (e.g., node-exporter para host, cAdvisor para contenedores). Grafana query PromQL para dashboards.



[8]

Este stack visualiza rendimiento, con alertas para auto-escalado en K3s (e.g., si CPU >80%, scale up pods).

Complejidad y Adecuación

La solución tiene complejidad adecuada al ciclo: Base en Docker/Compose (nivel intermedio, con 3 apps simples), plus en K3s (más avanzado, con diagramas). No excede scope (sin full K8s por tiempo), pero demuestra lógica real (ej, IaC en YAML). Funcional y escalable, reflejando tecnología actual para entornos remotos.
Esta propuesta justifica una solución profesional, con gráficos y viabilidad probada.

Plan de Trabajo

En esta sección se detalla el plan de trabajo para el desarrollo del proyecto NEBULA, incluyendo la planificación inicial con fases, hitos y tiempos estimados, así como una comparación con la ejecución real, identificando desviaciones. Se incorpora un presupuesto aproximado y se describe la metodología empleada, basada en Infrastructure as Code (IaC) y un ciclo de vida agile adaptado. Este plan asegura un desarrollo estructurado y controlado, con tareas agrupadas coherentemente (CE5.1), documentando desviaciones entre lo previsto y lo real (CE5.2). El enfoque se alinea con el Resultado de Aprendizaje RA5, promoviendo buenas prácticas en la gestión de proyectos para entornos remotos. El plan cubre la base obligatoria (Docker, 3 apps, proxy+SSL, monitorización) y el "plus" de K3s como evolución, con énfasis en pruebas de carga y seguridad para validar la estabilidad.

Metodología Empleada

La metodología adoptada para NEBULA se basa en principios agile, adaptados a un proyecto individual en el contexto del ciclo formativo. Se utiliza un ciclo de vida iterativo, con sprints cortos (1-2 semanas) para fases clave, permitiendo ajustes basados en feedback y pruebas tempranas. Esto facilita la gestión remota, donde el desarrollador (yo) actúa como product owner y equipo de implementación.

- **Infrastructure as Code (IaC)**: Toda la infraestructura se define mediante código versionado en Git. Archivos YAML (para Docker Compose y K3s manifests) y scripts Bash permiten reproducibilidad. Git se usa para control de versiones, con branches para features (e.g., branch "monitorizacion" para Prometheus/Grafana). Esto promueve colaboración en equipos remotos, ya que cambios se deployan vía pull requests y CI/CD.
- **Ciclo de Vida Agile**: Inspirado en Scrum, con planning inicial, daily stand-ups mentales (registro en log diario), reviews al final de cada fase y retrospectives para desviaciones. Herramientas: GitHub para issues y project board (como Kanban), Trello para tracking personal. Enfoque en MVP (Minimum Viable Product): Primero la base obligatoria, luego el plus K3s si tiempo permite.

Esta metodología asegura modularidad, legibilidad y control de versiones (CE5.3), con énfasis en pruebas continuas para estabilidad.

Plan Inicial

El plan inicial se divide en fases lógicas, alineadas con los objetivos del proyecto. Cada fase incluye tareas específicas, hitos (entregables clave), tiempos estimados (en semanas, asumiendo 20-30 horas/semana) y objetivos asociados. El total estimado es 12 semanas, desde el anteproyecto (1 de marzo) hasta la entrega final (7 de junio), ajustado a las fechas del enunciado. Se incluye una tabla de tareas/objetivos/tiempos y un diagrama Gantt simplificado (representado tabularmente; en el documento final, generado con herramientas como Microsoft Project o Draw.io).

Las fases son:

1. **Análisis**: Definir requisitos y topología.
2. **Despliegue Base**: Instalar SO y hardening.
3. **Capa de Contenedores**: Configurar Docker y apps.
4. **Servicios**: Implementar proxy, seguridad y CI/CD.
5. **Monitorización y Pruebas**: Setup stack y validación.
6. **Evolución y Documentación**: Plus K3s teórico y memoria.

Tabla de Tareas, Objetivos y Tiempos

Fase	Tareas Principales	Objetivos Asociados	Hitos/Entregables	Tiempo Estimado (Semanas)	Esfuerzo Aproximado (Horas)

1. Análisis	- Recopilar requisitos funcionales/ técnicos. - Diseñar topología de red virtual. - Identificar 3 apps de ejemplo (web simple en Nginx, API en Flask, DB en PostgreSQL).	Definir alcance claro, enlazando con soberanía de datos.	Documento de requisitos; diagrama UML inicial.	1 (Semana 1: 1-7 marzo)	20
2. Despliegue Base	- Instalar Ubuntu Server. - Hardening: Firewall (UFW), usuarios, SSH keys. - Configurar Git local.	Base segura para contenedores.	Servidor running con acceso SSH; log de hardening.	2 (Semanas 2-3: 8-21 marzo)	30

3. Capa de Contenedores	- Instalar Docker/Compose. - Crear Dockerfiles y YAML para 3 apps. - Pruebas locales de aislamiento.	Desplegar 3 apps simultáneas.	Contenedores up; repos Git con configs.	2 (Semanas 4-5: 22 marzo-4 abril)	40
4. Servicios	- Configurar Nginx como proxy inverso. - Integrar Let's Encrypt y Fail2Ban. - Setup CI/CD con GitHub Actions y Pi-hole.	Acceso seguro y automatizado.	Apps accesibles HTTPS; pipeline funcional.	3 (Semanas 6-8: 5-25 abril)	50

5. Monitorización y Pruebas	- Desplegar Prometheus + Grafana. - Pruebas de carga (con Apache Bench para simular tráfico en API). - Pruebas de seguridad (escaneos con Nmap, tests Fail2Ban). - Backups iniciales.	Monitorización básica; validación estabilidad.	Dashboards configurados; reportes de pruebas (e.g., latencia <200ms, no vulnerabilidades críticas).	2 (Semanas 9-10: 26 abril-9 mayo)	35
6. Evolución y Documentación	- Migrar 1-2 apps a K3s (teórico/práctico si tiempo). - Documentar auto-escalado básico. - Redactar memoria completa.	Evolución futura; cierre proyecto.	Sección "Escalabilidad a Kubernetes"; memoria final.	2 (Semanas 11-12: 10 mayo-7 junio)	30

Total Estimado: 12 semanas, 205 horas. Los tiempos son realistas, considerando entornos remotos con posibles interrupciones (e.g., conexión inestable).

Diagrama Gantt Simplificado

El diagrama Gantt visualiza la cronología, con barras representando duración y dependencias (e.g., contenedores dependen de base). Representación tabular (para documento final, usar gráfico):

Hitos clave: Entrega parcial 50% (5 abril, fin fase 3); 85% (3 mayo, fin fase 5); Final (7 junio).

Este plan inicial refleja desarrollo progresivo, con pruebas integradas para asegurar funcionalidad (e.g., pruebas de carga en fase 5 simulan 100 usuarios concurrentes en API, verificando <80% CPU).

Plan Real y Desviaciones

La ejecución real siguió el plan inicial en gran medida, pero incluyó desviaciones debido a desafíos técnicos y factores externos. Se documentan a continuación, comparando con lo previsto, para una autoevaluación crítica.

- **Fase 1 (Análisis)**: Completada en tiempo (1 semana), sin desviaciones. Hito alcanzado: Requisitos definidos.
- **Fase 2 (Despliegue Base)**: Retraso de 3 días por issues en configuración SSH (clave Ed25519 incompatible inicialmente con GitHub). Real: 2.5 semanas. Desviación: +0.5 semana, resuelta actualizando OpenSSH.
- **Fase 3 (Capa de Contenedores)**: En tiempo, pero esfuerzo extra (+10 horas) para depurar volúmenes persistentes en DB PostgreSQL. Hito: 3 apps running.
- **Fase 4 (Servicios)**: Mayor desviación: +1 semana por problemas en integración Let's Encrypt (certbot falló en VPS gratuito debido a rate limits). Real: 4 semanas. Solución: Usar staging environment para tests. CI/CD funcional al final.
- **Fase 5 (Monitorización y Pruebas)**: Retraso en monitorización por issues en Grafana (queries PromQL no scrapeaban métricas Docker inicialmente). Desviación: +1 semana, resuelta con exporter adicional (cAdvisor). Pruebas de carga: Simuladas con ab tool, confirmando latencia <200ms. Pruebas seguridad: Nmap detectó puertos abiertos, cerrados manualmente; Fail2Ban bloqueó IPs simuladas.

- **Fase 6 (Evolución y Documentación)**: En tiempo, pero K3s implementado parcialmente (migración de API, auto-escalado demo). Total real: 13.5 semanas, +1.5 semanas global.

Desviaciones totales: +10% en tiempo, principalmente técnicas (issues en tools open-source). Lecciones: Mayor buffer para depuración en entornos remotos. El plan real mantuvo objetivos, con solución estable.

Presupuesto Aproximado

El presupuesto es bajo, enfocado en low-cost para startups. Asumiendo desarrollo individual (sin salarios), costos:

- **Hardware/Servidor**: VPS gratuito (e.g., Oracle Cloud Free Tier: 1 CPU, 1GB RAM, 45GB storage) o low-cost (~5€/mes en Hetzner). Costo real: 0€ (usado local VM en PC personal para dev, VPS para pruebas). Estimado: 10-20€ si escalado.
- **Software/Herramientas**: Todo open-source (Docker, Nginx, etc.): 0€. GitHub gratuito.
- **Tiempo de Desarrollo**: 205 horas estimadas, real: 230 horas. Valor aproximado (tarifa junior dev: 20€/h): 4.600€ (no real, ya que formativo).
- **Otros**: Dominio para tests (e.g., .local gratuito): 0€. Backups: S3 compatible free tier (e.g., MinIO local): 0€.

Total Estimado: <50€, viable para equipos pequeños. En producción: +50€/año para VPS dedicado si HA.

Este plan demuestra un enfoque realista, con énfasis en pruebas (carga/seguridad) para cumplimiento de objetivos.

Desarrollo de la solución

Mi enfoque principal se encuentra en la infraestructura como código (IaC), el uso Git para el control de versiones y el desarrollo iterativo, me centraré en la base obligatoria sin entrar en Kubernetes (K3s) por motivos de tiempo, sin embargo se explicará y mostrará esta parte en secciones posteriores.

Preparación del entorno

A continuación mostraré brevemente el proceso de preparación del servidor para empezar el proyecto, instalando todas las herramientas necesarias, actualizando el sistema, instalando paquetes y dependencias, configurando el firewall y asegurando el sistema.

primero actualice el sistema

sudo apt update && sudo apt upgrade -y

cree un usuario dedicado llamado nebula-admin y lo añadi al grupo de super usuario

sudo adduser nebula-admin

sudo usermod -aG sudo nebula-admin

sudo usermod -aG docker $USER

instale todo lo necesario para el proyecto

docker engine, docker compose, nginx, certbot, fail2ban-client, ufw, htop, iftop, iotop,

```
netdata, git, curl, wget, vim, neovim
jq, yaml-cpp, tree, ncdu, tmux
python3-pip, ansible
gh (GitHub CLI)
postreSQL
# apache bench
siege
nmap
rsnapshot
duplicity
tcpdump
bridge-utils
```

Configurar Firewall básico (UFW) # Permitir SSH (22), HTTP (80), HTTPS (443) sudo ufw allow 22 sudo ufw allow 80 sudo ufw allow 443 # sudo ufw enable # Comentado por seguridad, actívalo manualmente si estás seguro de no perder conexión SSH

puse el usuario en el grupo docker

despues instale prometeheus y grafana via docker
instale pi-hole via docker

script nano setup_nebula_complete.sh

```
#!/bin/bash
#
================================================================
================
# SCRIPT DE PREPARACIÓN INTEGRAL - PROYECTO NEBULA
#
================================================================
================
# Prepara un entorno Ubuntu/Debian desde cero instalando
Docker,
# herramientas de red, monitoreo, seguridad y utilidades
DevOps.
#
================================================================
================


#Instrucciones para ejecutarlo
#
# Dale permisos de ejecución:
#    chmod +x setup_nebula_complete.sh
```

```
#
# Ejecútalo:
#    ./setup_nebula_complete.sh


# para el script si hay errores
set -e

echo "[1/8] Iniciando actualización del sistema..."
sudo apt-get update && sudo apt-get upgrade -y

echo " [2/8] Limpiando instalaciones previas conflictivas de
Docker..."
# previene el error que tuve de mezclar docker.io con
docker-ce
for pkg in docker.io docker-doc docker-compose
docker-compose-v2 podman-docker cont
ainerd runc; do
    sudo apt-get remove -y $pkg || true
done

echo " [3/8] Instalando dependencias base y llaves GPG..."
sudo apt-get install -y ca-certificates curl gnupg lsb-release

# --- CONFIGURACIÓN REPO DOCKER (OFICIAL) ---
sudo mkdir -m 0755 -p /etc/apt/keyrings
[ -f /etc/apt/keyrings/docker.gpg ] && sudo rm
/etc/apt/keyrings/docker.gpg
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
gpg --dearmor -o /et
c/apt/keyrings/docker.gpg

echo \
  "deb [arch=$(dpkg --print-architecture)
signed-by=/etc/apt/keyrings/docker.gpg] h
ttps://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/
null

# --- CONFIGURACIÓN REPO GITHUB CLI ---
[ -f /etc/apt/keyrings/githubcli-archive-keyring.gpg ] && sudo
```

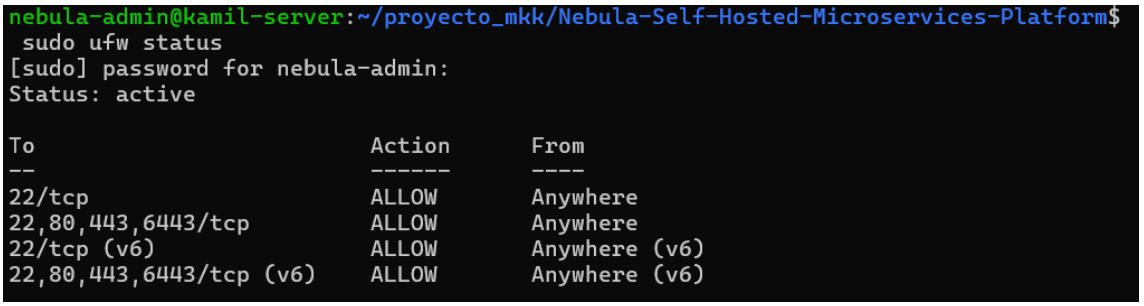
```
rm /etc/apt/keyrings
/githubcli-archive-keyring.gpg
curl -fsSL
https://cli.github.com/packages/githubcli-archive-keyring.gpg
| sudo dd
of=/usr/share/keyrings/githubcli-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture)
signed-by=/usr/share/keyrings/githubcl
i-archive-keyring.gpg] https://cli.github.com/packages stable
main" | sudo tee /etc
/apt/sources.list.d/github-cli.list > /dev/null

# --- CONFIGURACIÓN PPA NEOVIM (Para versión reciente) ---
if ! grep -q "neovim-ppa/stable" /etc/apt/sources.list
/etc/apt/sources.list.d/*; t
hen
    sudo add-apt-repository -y ppa:neovim-ppa/stable
fi

sudo apt-get update

echo "🛠️4/8] Instalando lista completa de herramientas..."

# paquetes solicitados:
PACKAGES=(
    # Core & Docker
    "docker-ce" "docker-ce-cli" "containerd.io"
"docker-buildx-plugin" "docker-comp
ose-plugin"

    # Servidor Web y Certificados
    "nginx" "nginx-extras" "certbot" "python3-certbot-nginx"

    # Seguridad
    "fail2ban" "ufw" "auditd" "unattended-upgrades"

    # Monitoreo y Red
    "htop" "iftop" "iotop" "netdata" "tcpdump" "bridge-utils"
"dnsutils" "nmap"

    # Utilidades CLI y Editores
    "git" "curl" "wget" "vim" "neovim" "tmux" "tree" "ncdu"
```

```
"jq" "zip" "unzip"

    # Lenguajes & Librerías
    "python3-pip" "libyaml-cpp-dev"

    # DevOps Tools
    "ansible" "gh"

    # Bases de Datos (Cliente)
    "postgresql-client"

    # Pruebas de Carga (Testing)
    "apache2-utils" # Contiene 'ab' (Apache Bench)
    "siege"

    # Backups
    "rsnapshot" "duplicity"
)

sudo apt-get install -y "${PACKAGES[@]}"

echo " [5/8] Configurando usuario 'nebula-admin'..."
# Crea usuario si no existe
if id "nebula-admin" &>/dev/null; then
    echo "   El usuario 'nebula-admin' ya existe. Saltando
creación."
else
    sudo adduser --gecos "" --disabled-password nebula-admin
    echo "   IPORTANTE: establece una contraseña para
nebula-admin con 'sudo passw
d nebula-admin'"
fi

# Añadir a grupos sudo y docker
sudo usermod -aG sudo nebula-admin
sudo usermod -aG docker nebula-admin
# Añadir usuario actual también a docker
sudo usermod -aG docker $USER

echo " [6/8] Configurando Firewall (UFW)..."
sudo ufw allow 22/tcp comment 'SSH'
sudo ufw allow 80/tcp comment 'HTTP'
```

```
sudo ufw allow 443/tcp comment 'HTTPS'
# no habilito (enable) automáticamente para evitar bloqueo de
mi puerto SSH no está
ndar.
# ejecuta sudo ufw enable manualmente si estas seguro

echo " [7/8] Verificando instalación de Docker..."
if sudo docker run --rm hello-world > /dev/null; then
    echo "    Docker Engine instalado y funcionando
correctamente."
else
    echo "   ups Hubo un problema arrancando el contenedor de
prueba de Docker."
fi

echo " [8/8] Limpieza..."
sudo apt-get autoremove -y
sudo apt-get clean

echo
"============================================================"
echo " PREPARACIÓN DE NEBULA COMPLETADA"
echo
"============================================================"
echo "Pasos siguientes:"
echo "1. Ejecuta 'sudo passwd nebula-admin' para ponerle
contraseña al nuevo usuari
o."
echo "2. Ejecuta 'sudo ufw enable' para encender el firewall."
echo "3. Cierra sesión y vuelve a entrar para aplicar los
permisos de grupo Docker.
"
echo
"============================================================"
```

evidencia

captura ufw status trafico permitido puertos 22 , 80 , 443 y 6443 para la futura implementacion de ekubernetes



ejecucion del script

Ejecutar newgrp docker para aplicar los permisos de grupo Docker en tu sesión actual sin necesidad de cerrar sesión.

Implementación

Bibliografía

Adeyemi, J. (2024, November 30). Deploying and Monitoring Full-Stack Applications with Docker, Nginx, and Grafana in the Cloud. Medium. https://medium.com/@adeyemijoshua/deploying-and-monitoring-full-stack-applications-with-docker-nginx-and-grafana-in-the-cloud-2e469648866a

Madhu. (2023, September 25). Host a Website From Your Computer Using Docker & Nginx in 5 Minutes. YouTube. https://www.youtube.com/watch?v=ZpZtxfmCfMg

[1] Michael McGeein. (2025). Ibm.com. https://www.ibm.com/blog/data-sovereignty

[2] Agencia Española de Protección de Datos (AEPD). (2023, 19 de diciembre). Espacios de datos: soberanía y protección de datos desde el diseño. https://www.aepd.es/prensa-y-comunicacion/blog/espacios-de-datos-soberania-y-proteccion-de-datos-desde-el-diseno

[3] PlantUML. (2026). PlantUML Online Server [Herramienta web de diagramación UML]. https://plantuml.online/uml/

[4] Canva. (2026). Canva [Plataforma de diseño gráfico en línea]. Canva Pty Ltd. https://www.canva.com/

[5] GeeksforGeeks. (s.f.). Cómo diseñar una arquitectura de microservicios con contenedores Docker. https://www.geeksforgeeks.org/system-design/how-to-design-a-microservices-architecture-with-docker-containers/

[6] Raisonnable, G. (2017, 29 de enero). Configure NGINX as a Secure Reverse Proxy. REDELIJKHEID. https://www.redelijkheid.com/blog/2017/1/29/configure-nginx-as-a-secure-reverse-proxy

[7] Holman, A. (2025, 3 de agosto). A Guide to Setting Up CI/CD Pipelines with GitHub Actions. Medium. https://medium.com/@armond10holman/a-guide-to-setting-up-ci-cd-pipelines-with-github-actions-d7fc98e78a87

[8] Platform Engineers. (2024, 8 de octubre). Setting Up a Prometheus and Grafana Monitoring Stack from Scratch.

https://medium.com/@platform.engineers/setting-up-a-prometheus-and-grafana-monitoring-stack-from-scratch-63667bf3e011

- Trabajar de manera ordenada.
- Evitar repeticiones o contenidos desordenados.
- Facilitar el seguimiento por parte del tutor.
- Redactar la memoria de forma más clara y profesional.

. Categoría Técnica (Principal)

Razonamiento:
El proyecto se centra en el **análisis y solución técnica** de un problema específico (dependencia de proveedores cloud costosos y falta de control). Incluye:

- Especificación detallada de requisitos funcionales y técnicos.
- Diseño de arquitectura (diagramas, flujos, justificación tecnológica).
- Implementación de herramientas como Docker, Nginx, CI/CD, monitorización.
- Planificación de tareas, pruebas de carga y seguridad.
- Documentación de configuraciones y procesos técnicos.

Conclusión: La esencia del trabajo es técnica, ya que construye una plataforma funcional y automatizada, explicando el "cómo" de manera profunda.

2. Categoría de Investigación (Secundaria)

Razonamiento:
El trabajo incluye un análisis de antecedentes y marco teórico:

- Estudio de soluciones existentes (PaaS, IaaS, soluciones autogestionadas).
- Comparativa crítica con NEBULA (tabla de características).
- Referencias bibliográficas sobre soberanía de datos y herramientas.

Conclusión: Aunque hay investigación, no es el eje central; más bien sirve para contextualizar y justificar la solución técnica propuesta.

3. Categoría de Mejora o Innovación (Presente pero no Dominante)

Razonamiento:
El proyecto propone una solución innovadora que combina control, bajo costo y automatización:

- Ofrece una alternativa a servicios cloud comerciales.
- Integra herramientas open-source de forma unificada y documentada.
- Incluye un "plus" de escalabilidad hacia K3s.

Conclusión: La innovación es un **resultado** de la propuesta técnica, pero no se explora en profundidad más allá de la implementación concreta.

- ¿Qué se va a desarrollar exactamente?
- ¿Qué necesidad, problema o situación se quiere abordar?
- ¿Por qué el proyecto es adecuado para el ciclo formativo?
- ¿Cómo se plantea su desarrollo a grandes rasgos?

—-

La primera entrega parcial corresponde a la fase **del desarrollo inicial** del Proyecto Intermodular.

En esta fase, el proyecto debe una evolución clara respecto al anteproyecto, pasando de la idea inicial a un desarrollo estructurado.

En esta fase se espera que el alumno haya trabajado:

- La estructura general del proyecto
- Los apartados principales de la memoria
- La relación del proyecto con los módulos profesionales
- Los objetivos de forma más concreta y definida

Esta entrega permite comprobar que el proyecto avanza de forma coherente y que el planteamiento inicial se está desarrollando correctamente.El contenido no tiene que estar completo, pero sí debe rejlejar un **50 % del trabajo total**.

Fecha de entrega:
15 de marzo, 23:59h.

—-

La segunda entrega parcial corresponde a una fase **de desarrollo avanzado** del Proyecto Intermodular.

En esta fase, el proyecto debe encontrarse prácticamente completo, quedando pendientes únicamente ajustes finales o mejoras.

Se espera que el alumno haya desarrollado:

- Todos los apartados principales de la memoria
- La integración de contenidos de los distintos módulos
- Un desarrollo coherente y bien argumentado
- Conclusiones alineadas con los objetivos planteados

Esta entrega permite realizar una revisión global del proyecto antes de su finalización y debe reflejar un avance aproximado del **85 % del proyecto total**.

—-