

# Despliegue de aplicaciones web mediante plataformas Cloud



Cuando desarrollamos una aplicación web no tenemos por qué encargarnos de todos los componentes necesarios para que funcione. Podemos desarrollar y administrar nosotros mismos el servidor, el sistema operativo, el entorno de ejecución, la base de datos y la aplicación, o podemos delegar parte de estas tareas en proveedores cloud.

Desde el punto de vista de un desarrollador, la principal diferencia entre los distintos modelos de servicios cloud es qué parte de la infraestructura y del software gestionamos nosotros y qué parte proporciona el proveedor. En esta unidad estudiaremos las principales opciones y conceptos necesarios para comprender el ecosistema actual de despliegue en la nube, y practicaremos con un ejemplo en alguna de ellas.

## 1. Modelos de servicios cloud

Los **servicios cloud** pueden clasificarse atendiendo al nivel de responsabilidad que asume el proveedor frente al desarrollador. La clasificación tradicional distingue tres modelos principales:

- **IaaS (Infrastructure as a Service)**
- **PaaS (Platform as a Service)**
- **SaaS (Software as a Service)**

Estos modelos representan diferentes niveles de abstracción y de responsabilidad entre el proveedor y el desarrollador. A continuación explicamos las características principales de cada uno.

### 1.1 IaaS - Infrastructure as a Service

**IaaS o Infraestructura como Servicio** consiste en contratar recursos informáticos como máquinas virtuales, almacenamiento, redes o capacidad de procesamiento. El proveedor se encarga de la infraestructura física, pero nosotros seguimos teniendo que administrar gran parte del entorno en el que se ejecutará nuestra aplicación.

Por ejemplo, podemos contratar una máquina virtual e instalar en ella Linux, Node.js, Express y nuestra aplicación. También tendremos que encargarnos de tareas como las actualizaciones, la configuración de la red, la seguridad y el mantenimiento del sistema. Proporciona, por tanto, mucho control, pero también mucha responsabilidad. Algunos ejemplos son **Amazon EC2**, **Azure Virtual Machines** o **Google Compute Engine**.

### 1.2 PaaS - Platform as a Service

**PaaS o Plataforma como Servicio** proporciona un entorno preparado para ejecutar nuestras aplicaciones. El proveedor se encarga de gran parte de la infraestructura y nosotros nos centramos principalmente en

desarrollar, configurar y desplegar nuestro software. Algunos ejemplos son **Render**, **Railway**, **Google App Engine** o **Azure App Service**.

Podemos conectar un repositorio de GitHub con una plataforma como Render y configurar los comandos necesarios para construir y ejecutar nuestro backend Node.js + Express. La plataforma se ocupa de gran parte de las tareas relacionadas con el servidor y nosotros nos centramos en nuestra aplicación. PaaS ofrece, por tanto, menos control que IaaS, pero simplifica considerablemente el despliegue y el mantenimiento.

### 1.3 SaaS — Software as a Service

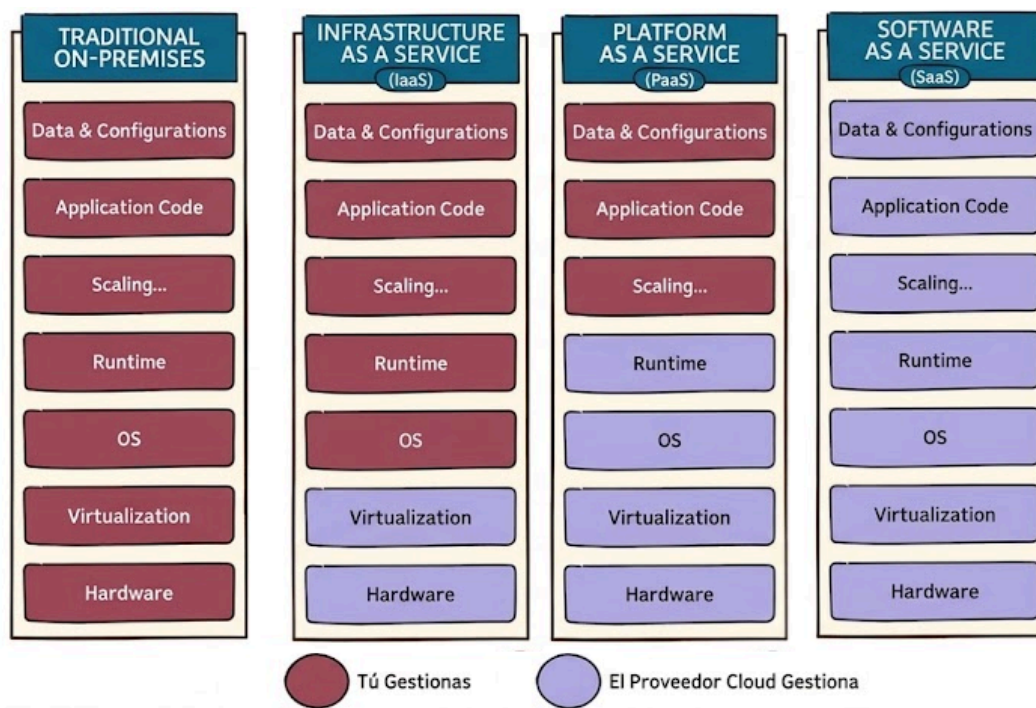
**SaaS** o **Software como Servicio** es el nivel más alto de abstracción en la nube. Consiste en utilizar una aplicación completamente desarrollada, alojada y mantenida por un tercero, a la que solemos acceder a través de un navegador web o mediante una API.

En este modelo, el proveedor se encarga de todas las capas: infraestructura física, sistema operativo, base de datos, código de la aplicación y escalabilidad. Nosotros únicamente consumimos el servicio y gestionamos nuestra propia cuenta o datos.

Tradicionalmente está orientado a usuarios finales (como Google Workspace, Jira o GitHub) pero, desde la perspectiva de un desarrollador, este concepto se extiende a la integración de servicios especializados vía API o SDK. En lugar de construir subsistemas complejos desde cero, consumimos soluciones ya operativas para:

- Procesamiento de pagos (ej. Stripe, PayPal).
- Envíos de correo transaccional (ej. SendGrid, Resend).
- Autenticación y gestión de usuarios (ej. Auth0, Clerk).
- Analítica y monitorización (ej. PostHog, Sentry).

La principal ventaja es que podemos incorporar funcionalidades rápidamente sin tener que desarrollar y mantener toda la infraestructura necesaria. A cambio, nuestra aplicación pasa a depender del proveedor, de sus precios, límites, disponibilidad y API.



## 1.4. Otros modelos

Con la evolución de la computación en la nube han aparecido otros modelos que describen formas específicas de consumir servicios cloud, como:

- **FaaS (Function as a Service)**: es un modelo asociado a las arquitecturas serverless. En lugar de desplegar y mantener un servidor que ejecute continuamente nuestra aplicación, desarrollamos funciones individuales que se ejecutan cuando reciben una petición o cuando ocurre determinado evento. Resulta especialmente interesante para determinadas APIs, tareas automáticas y aplicaciones cuyo tráfico puede variar mucho. Algunos ejemplos son **AWS Lambda**, **Azure Functions**, **Vercel Functions** o **Netlify Functions**.
- **BaaS (Backend as a Service)**: proporciona funcionalidades de backend ya desarrolladas que podemos integrar directamente en nuestra aplicación, como autenticación de usuarios, bases de datos, almacenamiento de archivos, notificaciones o APIs. Algunos ejemplos son **Firebase**, **Supabase** o **Appwrite**.
- **DBaaS (Database as a Service)**: el proveedor ofrece una base de datos lista para utilizar. Nosotros nos encargamos de diseñar y utilizar la base de datos, mientras que el proveedor administra gran parte de la infraestructura necesaria para ejecutarla. Un ejemplo conocido en este curso puede ser **MongoDB Atlas**.

Estos conceptos no deben entenderse necesariamente como categorías independientes al mismo nivel que IaaS, PaaS y SaaS, sino como formas especializadas de ofrecer o consumir determinados servicios cloud.

También cabe mencionar que estos modelos no son necesariamente excluyentes. Una aplicación web moderna puede combinar diferentes servicios cloud. Por ejemplo, podemos desarrollar un frontend Angular y un backend Node.js + Express, y desplegarlos en una plataforma como Render, y utilizar MongoDB Atlas como servicio de base de datos. De esta forma, podemos construir una aplicación combinando software desarrollado por nosotros con infraestructura y servicios proporcionados por terceros.

## 2. Ejemplo: Render y MongoDB Atlas

---

Para mostrar un ejemplo de todo esto emplearemos **Render**. En los siguientes apartados probaremos a desplegar una API REST con *Express* y *Mongoose*, alojando la base de datos en **MongoDB Atlas**.

### 2.1. Características principales de Render

Como hemos comentado anteriormente, Render entra dentro de la categoría de *PaaS (Platform as a Service)*, y nos proporciona un entorno completamente configurado para ejecutar nuestras aplicaciones web. Es especialmente interesante para nuestro caso porque permite desplegar actualmente servicios web Node.js/Express directamente desde un repositorio GitHub o similar, y dispone de una modalidad gratuita para experimentar ([web oficial](#))

En Render podemos desplegar tanto servidores *backend* persistentes (desarrollados en Node.js, Python, Go o cualquier tecnología empaquetada con Docker), como sitios web *frontend* en HTML/CSS/JS servidos de forma rápida y gratuita a través de una red CDN. Además, permite alojar y gestionar bases de datos PostgreSQL y Redis, y también conectar con otras bases de datos remotas.

Como punto fuerte, permite mantener procesos en ejecución continua (como servidores Express o WebSockets) y ofrece redes privadas para conectar de forma segura nuestro backend con la base de datos sin exponerla a Internet. En su modalidad gratuita, los servicios web se suspenden tras unos 15 minutos sin tráfico, por lo que la primera petición tras la inactividad puede tardar unos segundos en responder (arranque en frío).

Si lo comparamos con un VPS (*Servidor Privado Virtual*), Render ofrece una serie de ventajas:

- **Menor complejidad operativa:** no tenemos que instalar ni mantener el sistema operativo, las dependencias del servidor ni las actualizaciones de seguridad.
- **Certificados SSL automáticos:** la plataforma genera y renueva los certificados HTTPS de forma transparente sin que tengamos que configurar nada.
- **Gestión sencilla de variables de entorno:** podemos definir claves secretas, puertos o cadenas de conexión a la base de datos directamente desde el panel de control web.
- **Escalabilidad inmediata:** si nuestra aplicación necesita más recursos (más memoria RAM o más instancias para repartir la carga), podemos aumentarlos con un simple clic.

En cambio, en un VPS tenemos un control total del sistema y el entorno, pero debemos encargarnos de instalar y administrar todo: sistema operativo, servidores, conexiones remotas, etc.

Además, una de las características más potentes de las plataformas *PaaS* modernas es su **integración directa con repositorios** de GitHub o GitLab. Esto nos permite implementar un flujo de trabajo automatizado y continuo:

- **Detección de cambios:** conectamos nuestro repositorio a la plataforma. Cada vez que hacemos un `git push` a la rama principal (*main*), el *PaaS* detecta automáticamente la nueva versión.

- **Fase de construcción:** la plataforma descarga nuestro código, instala las dependencias necesarias (por ejemplo, ejecutando `npm install`) y prepara los archivos para producción.
- **Fase de arranque:** la plataforma inicia nuestro servidor ejecutando el comando configurado (por ejemplo, `npm run start` o `node server.js`).
- **Despliegue sin caídas:** el nuevo contenedor o versión de la aplicación se arranca en segundo plano. Hasta que no comprueba que responde correctamente, la plataforma no sustituye la versión anterior, evitando que los usuarios experimenten caídas del servicio durante la actualización.

## 2.2. Preparando el proyecto

Tomaremos como base el proyecto *ContactosREST\_v2* que se desarrolló en [este documento](#). En él construimos una API REST para gestionar un modelo sencillo de contactos.

### 2.2.1. Cambios respecto al proyecto Node local

[Aquí](#) puedes descargar el código del proyecto que usaremos, desde un repositorio GitHub. Si le echas un vistazo, hay algunos pequeños cambios sobre el original de aquel documento:

- Un cambio importante es que se utiliza un **fichero de entorno** para almacenar valores de configuración, tales como la URL de conexión a la base de datos, o el puerto por el que lanzar la aplicación. En la raíz del proyecto verás un fichero `.env.example` con datos de prueba. Deberás copiarlo a otro llamado `.env` y cambiar en él los datos por los de tu propia conexión. Para gestionar las variables de entorno instalamos la librería `dotenv` desde `package.json`, junto con `express` y `mongoose`.

```
PORT=3000
MONGODB_URI=mongodb+srv://usuario:password@url_cluster/nombreBD?parametros
```

- Otro cambio a destacar es la inclusión de un **fichero .gitignore** para indicar qué no queremos que se suba al repositorio GitHub que crearemos. En nuestro caso, descartaremos tanto la carpeta `node_modules` como el fichero `.env`, que puede tener datos sensibles para pruebas en local (sólo se sube `.env.example`, con datos de prueba inofensivos).

```
node_modules/
.env
```

- También encontrarás un **fichero db\_seed.js** en la raíz del proyecto, que contiene código para insertar en la base de datos una serie de contactos, restaurantes y mascotas de prueba, para que no esté vacía. Este fichero lo ejecutaremos luego para poblar la base de datos al inicio.
- Adicionalmente, se han añadido unos **scripts** en la sección `scripts` de `package.json` para lanzar la aplicación con el comando `npm` (aunque también se puede lanzar con `node index.js`). Desde Render se podrá elegir el comando a ejecutar para ponerla en marcha.

```
"scripts": {  
  "start": "node index.js",  
  "dev": "node --watch index.js",  
  "seed": "node db_seed.js"  
},
```

### 2.2.2. Acceso a MongoDB Atlas

Como segundo paso importante deberemos tener una cuenta creada en [MongoDB Atlas](#), y haber creado un cluster siguiendo los pasos indicados en [este documento](#).

En el fichero `.env` de nuestro proyecto debemos tener almacenada la URL de conexión a la base de datos, con el formato ya conocido de `mongodb+srv://usuario:password@url_cluster/nombreBD?parametros`. Dentro de los parámetros, pueden venir bien un par de ellos:

```
mongodb+srv://usuario:password@url_cluster/nombreBD?retryWrites=true&w=majority
```

- El parámetro `retryWrites=true` permite que, si una operación de escritura (insert, update, delete) falla por un error temporal de red o una reorganización del clúster (por ejemplo, un cambio de nodo principal), el driver de Node.js la reintente automáticamente una vez antes de lanzar un error a tu aplicación. Evita que la API falle ante cortes momentáneos de conexión.
- El parámetro `w=majority` establece el nivel de confirmación de escritura. En MongoDB Atlas los datos se replican en varios servidores (un conjunto de réplicas, usualmente 3 nodos). Con este parámetro indicamos a MongoDB que no devuelva la confirmación de éxito a la API hasta que el dato haya sido escrito físicamente en la mayoría de los nodos del clúster (al menos 2 de los 3). Garantiza que los datos no se pierdan si uno de los servidores sufre una caída justo después de recibir la petición.

## 2.3. Configuración de Render y repositorio GitHub

Ahora que ya tenemos configurado el proyecto Node de forma adecuada, y nuestro *cluster* en MongoDB Atlas, vamos a configurar un repositorio GitHub donde alojaremos nuestro proyecto, y a sincronizarlo con Render para que despliegue la aplicación.

### 2.3.1. Creación del repositorio en GitHub

Vamos a crear nuestro repositorio público en GitHub donde almacenar este proyecto. Supondremos, para este paso, que tienes cuenta creada en [GitHub](#) y que has instalado la herramienta [Git](#) en tu sistema. Si no es así, completa estos dos pasos antes de continuar.

Lo primero que haremos será inicializar un repositorio local en nuestra carpeta de proyecto Node. Podemos hacerlo desde el terminal integrado en VS Code, dentro del proyecto, escribiendo estos comandos:

```
# Inicializamos repositorio
git init

# Añadimos todo el contenido inicialmente (el fichero .gitignore excluye lo que corr
git add .
git commit -m "Commit inicial: proyecto Express y Mongoose configurado"

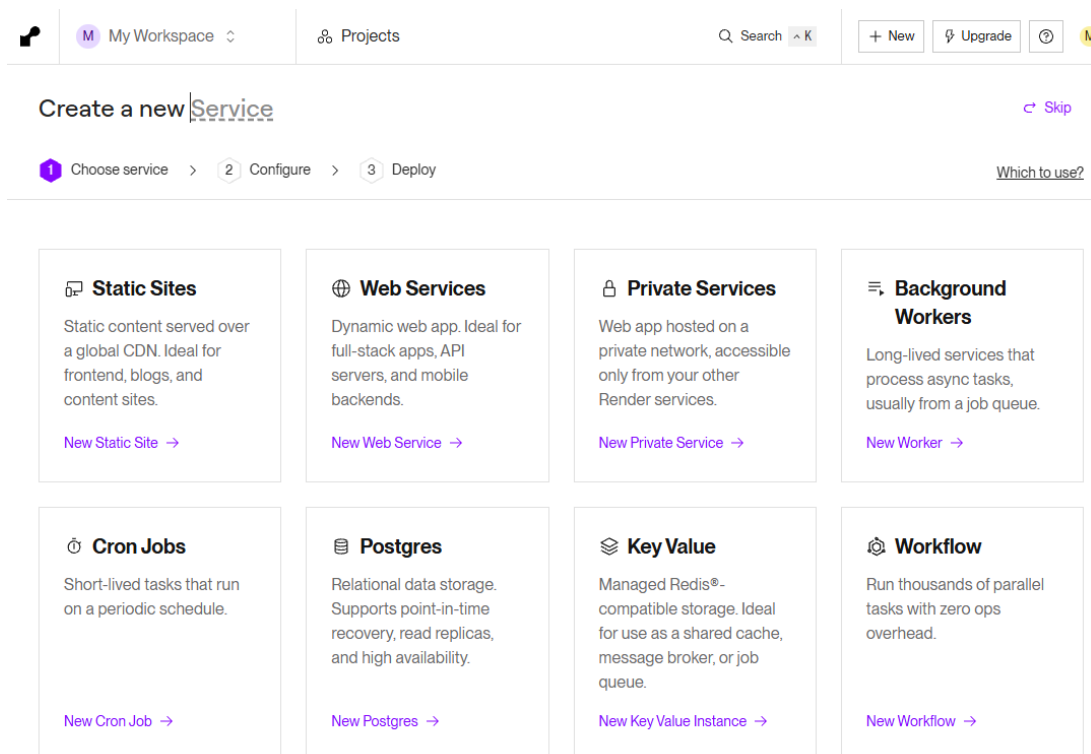
# Renombramos la rama principal a "main" (opcional, aunque recomendable)
git branch -M main
```

Ahora debemos acceder a [GitHub](#), crear un nuevo repositorio vacío (podemos llamarlo como queramos. Por ejemplo, `ContactosREST`) y quedarnos con su dirección (por ejemplo, `https://github.com/tu_usuario/ContactosREST.git`). Ahora vamos a conectar nuestro repositorio local con ese repositorio GitHub:

```
git remote add origin https://github.com/tu-usuario/tu-repositorio.git
git push -u origin main
```

### 2.3.2. Creación de proyecto en Render y conexión con GitHub

Como último paso, vamos a crear un nuevo proyecto en nuestra cuenta de [Render](#) (si no la tienes, créala). Desde nuestro panel (*Dashboard*) elegimos crear un nuevo *Web Service*.



Después conectamos con el repositorio en cuestión

Git Provider

Public Git Repository

Existing Image

PR Previews and Auto-Deploy are available only for repositories configured with Blueprints



Connect →

El siguiente paso es configurar el servicio. En el formulario nos piden que pongamos un nombre, el lenguaje o framework de desarrollo (Node), la rama principal del repositorio, la región donde desplegarlo...

**Name**  
A unique name for your web service.

**Language**  
Choose the [runtime environment](#) for this service.

**Branch**  
The Git branch to build and deploy.

**Region**  
Your services in the same [region](#) can communicate over a [private network](#).

**Root Directory** Optional  
If set, Render runs commands from this directory instead of the repository root. Additionally, code changes outside of this directory do not trigger an auto-deploy. Most commonly used with a [monorepo](#).

También indicamos el comando para construir el proyecto (en nuestro caso, `npm install`) y para ponerlo en marcha (podemos elegir `node index.js` o bien `npm run start` de nuestros *scripts*). Finalmente, podemos especificar la tarifa (gratuita en nuestro caso) y las variables de entorno que se van a utilizar. Podemos añadirlas a mano o bien copiarlas/pegarlas de nuestro fichero `.env` (recuerda que este fichero NO se sube a GitHub).

**Compute**  
For more power and to get the most out of Render, we recommend using one of our paid compute plans.  
All paid compute plans support:

- Zero Downtime
- SSH Access
- Scaling
- One-off jobs
- Support for persistent disks

Selected:  
\$0 / month 0.1 CPU 512 MB RAM

Free instances spin down after periods of inactivity. They do not:

|                                              |         |
|----------------------------------------------|---------|
| <input checked="" type="radio"/> \$0 / month | 0.1 CPU |
| <input type="radio"/> \$7 / month            | 0.5 CPU |
| <input type="radio"/> \$25 / month           | 1 CPU   |
| <input type="radio"/> \$85 / month           | 2 CPU   |

Show all 15 compute plans

**Environment Variables**  
Set environment-specific config and secrets (such as API keys), then read those values from your code. [Learn more.](#)

| KEY         | VALUE                              |
|-------------|------------------------------------|
| MONGODB_URI | <input type="text" value="value"/> |

En la sección de opciones avanzadas podemos especificar algún detalle más. Por ejemplo (para opciones de pago) podemos indicar un *pre-deploy Command*, que vendría bien para ejecutar `npm run seed` y poblar la base de datos al inicio. Como está deshabilitado para la opción gratuita, deberemos ejecutar este comando desde nuestro ordenador local para rellenar la base de datos de MongoDB Atlas.

Una vez rellenas las opciones de configuración, podemos desplegar el servicio y, si todo es correcto, tras unos segundos para hacer el *build* e instalar las librerías necesarias, lo tendremos disponible en una dirección por defecto dentro del dominio `xxxxx.onrender.com`, donde `xxxxx` será el nombre que le hayamos dado al proyecto en Render.

## 2.4. Alternativas a Render

Actualmente existen muchas plataformas *PaaS* y, aunque todas nos permiten conectar un repositorio de GitHub para automatizar el despliegue, cada una está pensada para resolver necesidades distintas, desde frontends estáticos optimizados en CDN hasta backends persistentes y bases de datos gestionadas. A continuación nombramos algunas con sus particulares características.

- **Vercel:** es la plataforma de referencia para desplegar proyectos frontend modernos y aplicaciones creadas con frameworks de renderizado híbrido. En ella podemos desplegar tanto aplicaciones *frontend* hechas con frameworks modernos (React, Vue, Astro...) como arquitecturas *serverless*. Sin embargo, no está diseñada para alojar servidores backend tradicionales de ejecución continua.
- **Netlify:** plataforma que popularizó el concepto Jamstack (JavaScript, APIs y Markup). Al igual que Vercel, está muy enfocada en la capa frontend. Permite desplegar sitios web estáticos, aplicaciones SPA (*Single Page Applications*) y funciones *serverless*.
- **Railway:** diseñada para desplegar arquitecturas full-stack y microservicios mediante una interfaz visual muy intuitiva. Permite desplegar backends en cualquier lenguaje, contenedores Docker y múltiples bases de datos con un solo clic (PostgreSQL, MySQL, Redis y MongoDB). Sin embargo, su modelo de uso está más enfocado al consumo por recursos que a un plan gratuito tradicional permanente.

Aquí vemos un análisis comparativo de las plataformas comentadas:

Resumen técnico de características y selección de herramientas

| Característica                    | Render                | Vercel                | Netlify               | Railway                         |
|-----------------------------------|-----------------------|-----------------------|-----------------------|---------------------------------|
| Tipo de proyecto ideal            | Backends y Full-Stack | Frontends y Next.js   | Frontends y Jamstack  | Microservicios y Full-Stack     |
| Servidores persistentes (Express) | Sí (Nativo)           | NO (Serverless)       | NO (Serverless)       | Sí (Nativo)                     |
| Bases de datos integradas         | PostgreSQL y Redis    | Integración externa   | Integración externa   | PostgreSQL, MySQL, Mongo, Redis |
| Soporte para Docker               | Sí (Nativo)           | NO                    | NO                    | Sí (Nixpacks / Docker)          |
| WebSockets / Conexión continua    | Sí (Soportado)        | LIMITADO (Serverless) | LIMITADO (Serverless) | Sí (Soportado)                  |
| Despliegues automáticos con Git   | Sí (Build & Deploy)   | Sí (Deploy Previews)  | Sí (Deploy Previews)  | Sí (Entornos efímeros)          |
| Plan gratuito para aprender       | Sí (con suspensión)   | Sí (muy generoso)     | Sí (muy generoso)     | Créditos de prueba              |