brutal.

Introducción a Kubernetes

Autor: Víctor Díaz Marco
Publicación: 11 de abril de 2019
Actualización: 12 de abril de 2023

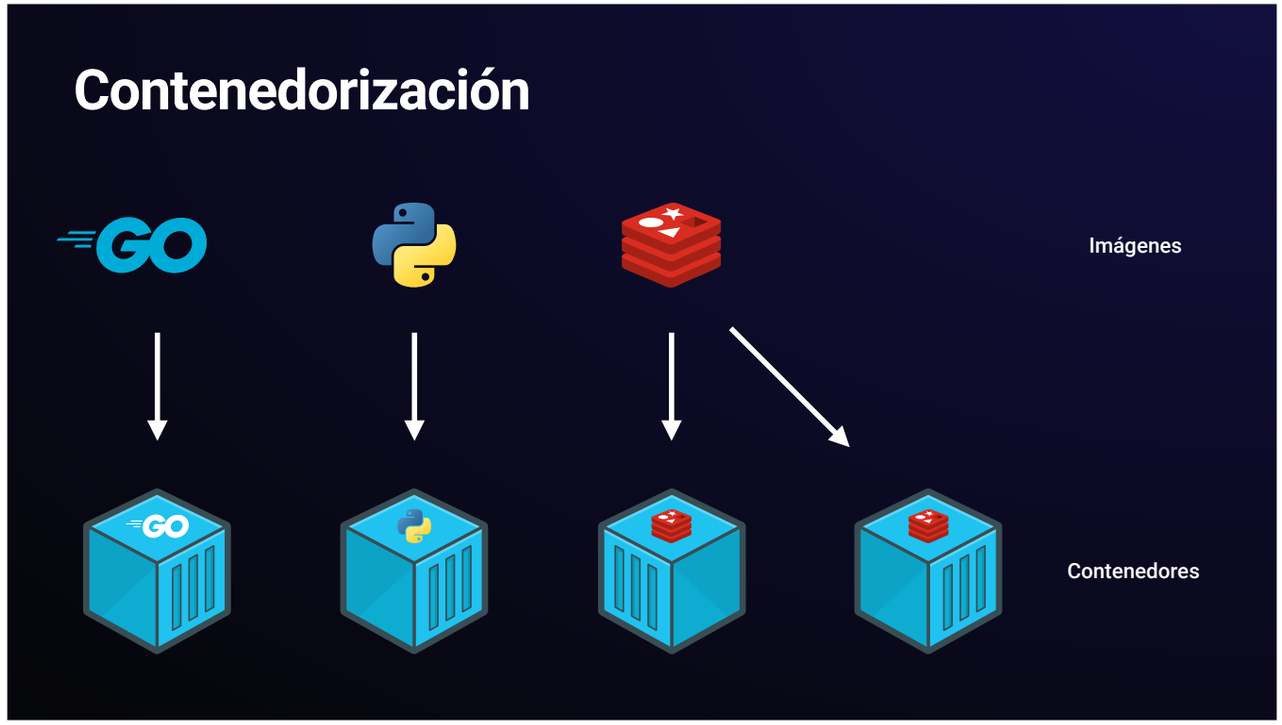
brutal.systems

Contenedorización

- Técnica de distribución de software.
- Permite empaquetar, distribuir y ejecutar software en paquetes que incluyen todas sus dependencias.
- Estandarizada por la Open Container Initiative, proyecto de la Fundación Linux.
- Utiliza las características de aislamiento de procesos del *kernel* de Linux.

Imagen de la izquierda: despliegue clásico (directo sobre el sistema operativo).
Imagen de la derecha: despliegue contenedorizado.

Un solo proceso por contenedor, ya que se están aislando procesos individuales. Usar un supervisor es una señal que indica la necesidad de **separar responsabilidades** en diferentes contenedores. Implica no tener control desde fuera y no poder aprovechar ciertas características (por ejemplo, controlar si termina un proceso y con qué estado).



Estos paquetes se llaman **imágenes**.

- Las **imágenes** son **fotografías** de un software en un momento concreto, junto con sus dependencias (como bibliotecas o configuración estática).
- Los **contenedores** son **instancias efímeras** de esas imágenes que se encuentran en ejecución.
- Los contenedores pueden tener **volúmenes** de almacenamiento de bloque y exponer servicios a la **red**.

Kubernetes

- Software de orquestación de contenedores.
- Del griego κυβερνήτης (/ky.ber.nɛ̌ː.tɛːs/): capitán o timonel.
- Desarrollado por Google y donado a la Cloud Native Computing Foundation, proyecto de la Fundación Linux.
- Permite desplegar, escalar, coordinar y gestionar contenedores de software.
- Ofrece resiliencia, escalabilidad, tenencia múltiple y reconciliación.
- Tiene un bajo acoplamiento.
- Escrito en Go.

/kubernetis/

Segundo proyecto de código abierto con más autores, después de Linux.

Utiliza una especificación llamada **Container Runtime Interface** (CRI), lo que permite utilizar contenedores creados por cualquier software compatible con dicha especificación, como Docker, Containerd o CRI-O.

- **Resiliencia:** reinicia contenedores si estos finalizan o no responden a las pruebas de vida.
- **Escalabilidad** horizontal (crea réplicas conforme la carga aumenta) y vertical (dota automáticamente a las instancias de más recursos cuando los necesitan).
- **Tenencia múltiple** o multitenencia: aprovecha los recursos compartiendo hardware sin interferencias.
- **Reconciliación:** se encarga de que el estado actual siempre sea el deseado.

Ventajas:
- Automatización prácticamente total de la infraestructura.
- Simplicidad a la hora de administrar la infraestructura y hacer cambios.
- Encaja con muchos casos de uso. Cada vez más.
- Importante aprovechamiento de recursos.
- Al utilizar contenedores en todas las etapas de desarrollo, todos los entornos (desarrollo, preproducción y producción) son exactamente iguales.

Facilita la adopción de una arquitectura de software orientada a **microservicios**: **más mantenibilidad**, **menor acoplamiento** y mayor facilidad para **escalar**.

Como curiosidad, es **utilizado por**:

- GitHub.
- Reddit.
- Slack.

Kubernetes como servicio

Se **integra** con el resto de servicios del proveedor: **computación**, **almacenamiento de bloque** o **balanceadores de carga**, entre otros.

- **Google Cloud.** Primero en aparecer. Muy bien integrado. Actualizaciones automáticas y autoescalado del *cluster*.
- **AWS.** Es el único que cobra por los *masters*: 140 $/mes.
- **Azure.**
- **DigitalOcean.** Finales de 2018. El más económico (sin estar OVH en la comparativa).
- **OVH.** Principios de 2019.

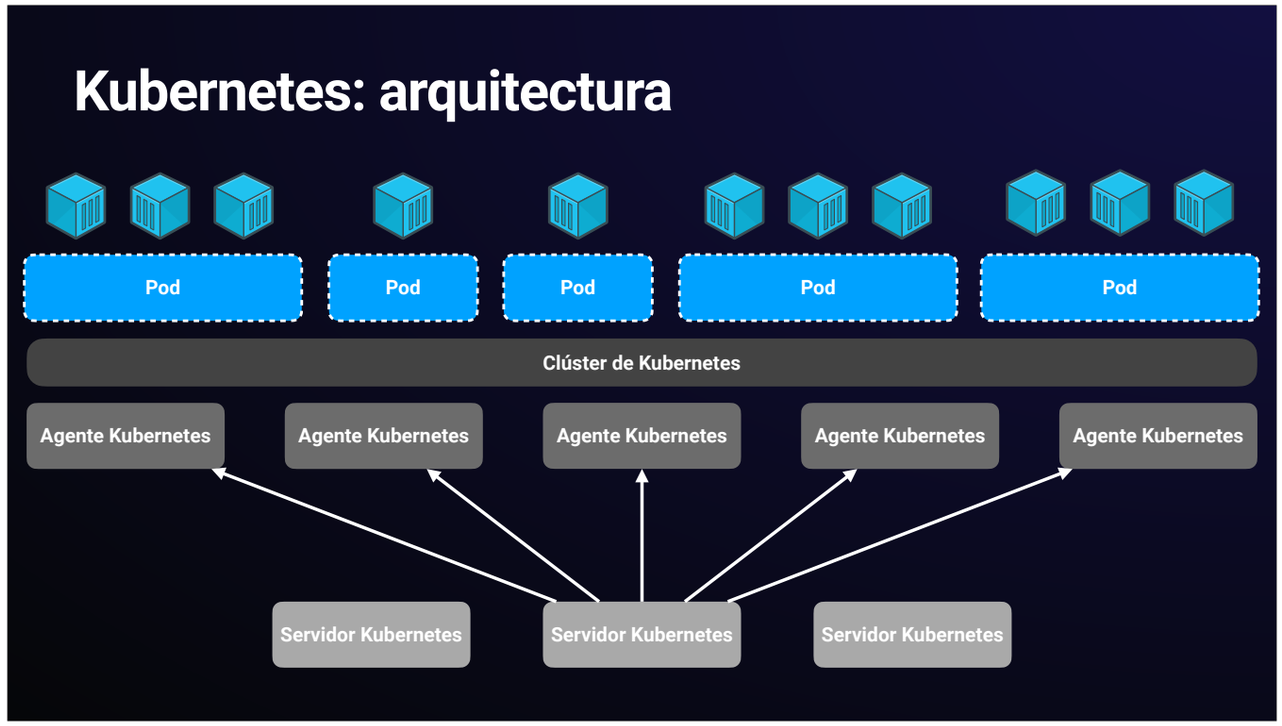
Kubernetes con instalación propia

Obviamente, también es posible desplegarlo *on-premises* (con instalación propia), con la ayuda de diversas herramientas.

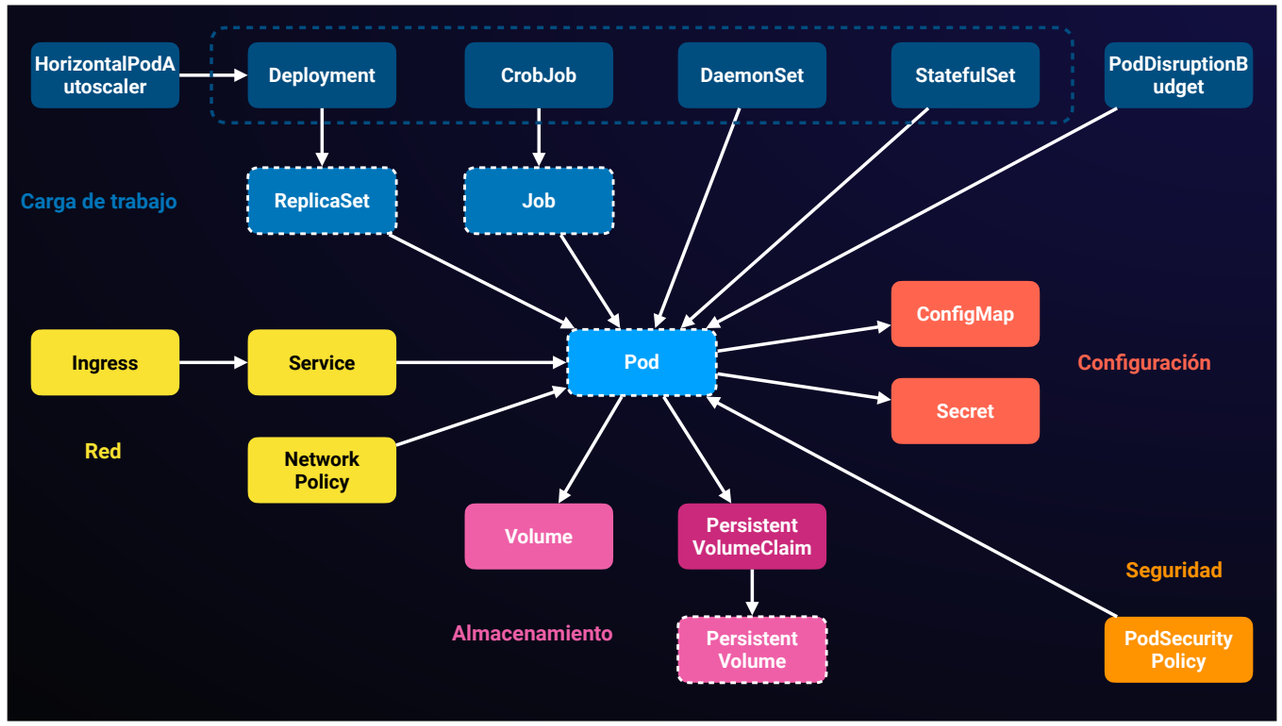
- **Kubeadm.** Herramienta oficial que permite desplegar un *cluster* mínimo viable y seguro. Está pensado para ser usado por herramientas más completas, ya que, por ejemplo, no despliega *plugin* de red.
- **Kubespray.** Utiliza Kubeadm y Ansible para desplegar un *cluster* listo para ser usado en producción con alta disponibilidad y de forma modular. Se integra con diversos proveedores en la nube y también permite desplegar directamente en *bare metal*.
- **Rancher.** Despliega un *cluster* completo (maestros, nodos y red) con todas las herramientas necesarias (monitorización y *logs*), incluyendo interfaces gráficas para administrarlo sencillamente sin necesidad de tener conocimientos extensos sobre Kubernetes. No utiliza la distribución oficial de Kubernetes, sino una propia certificada, al contrario que las herramientas anteriores.

Otros orquestadores

- **Mesos:** muy completo y maduro pero menos famoso.
- **Nomad:** más sencillo que Kubernetes.
- **Swarm:** más sencillo que Kubernetes y la apuesta de Docker hasta que pasó a apostar por Kubernetes.



- Un *cluster* es un grupo de máquinas que se comporta como una sola.
- Kubernetes ofrece una **capa de abstracción** sobre todas las máquinas, exponiendo una API con la que se interactúa.
- Las máquinas tienen dos roles: **servidores** (orquestadores) y **agentes** (trabajadores). Los primeros controlan a los segundos.
- Si cae un maestro, otro lo puede sustituir.
- Si cae un nodo, los maestros se encargan de volver al estado deseado (conciliación) reubicando recursos.
- Todos los recursos dentro del *cluster* son efímeros. La persistencia está fuera.
- Un *pod* está físicamente en un nodo, pero a nivel lógico no hay distinción (a pesar de que se puede saber: afinidad y antiafinidad).



Elementos entre línea de **puntos**: **no** se trabaja con ellos **directamente** aunque es necesario conocerlos.
Bloque entre línea de **puntos**: controladores de más alto nivel. Son las diferentes formas de trabajar con los *pods*.

- **Carga de trabajo:** Pod, Deployment, DaemonSet, StatefulSet, Job y CronJob.
- **Configuración:** ConfigMap y Secret.
- **Almacenamiento:** Volume, PersistentVolumeClaim y PersistentVolume.
- **Red:** Service, Ingress.

Objetos

- Kubernetes define bloques básicos (**objetos**) que representan **recursos**.
- Los objetos se definen mediante **especificaciones** en formato **YAML**.
- Una especificación define el **estado deseado** junto con algunos **metadatos**.

```yaml
kind: string
apiVersion: string

metadata:
  name: string
  namespace: string
  labels: {}
  annotations: {}

spec: {}
```

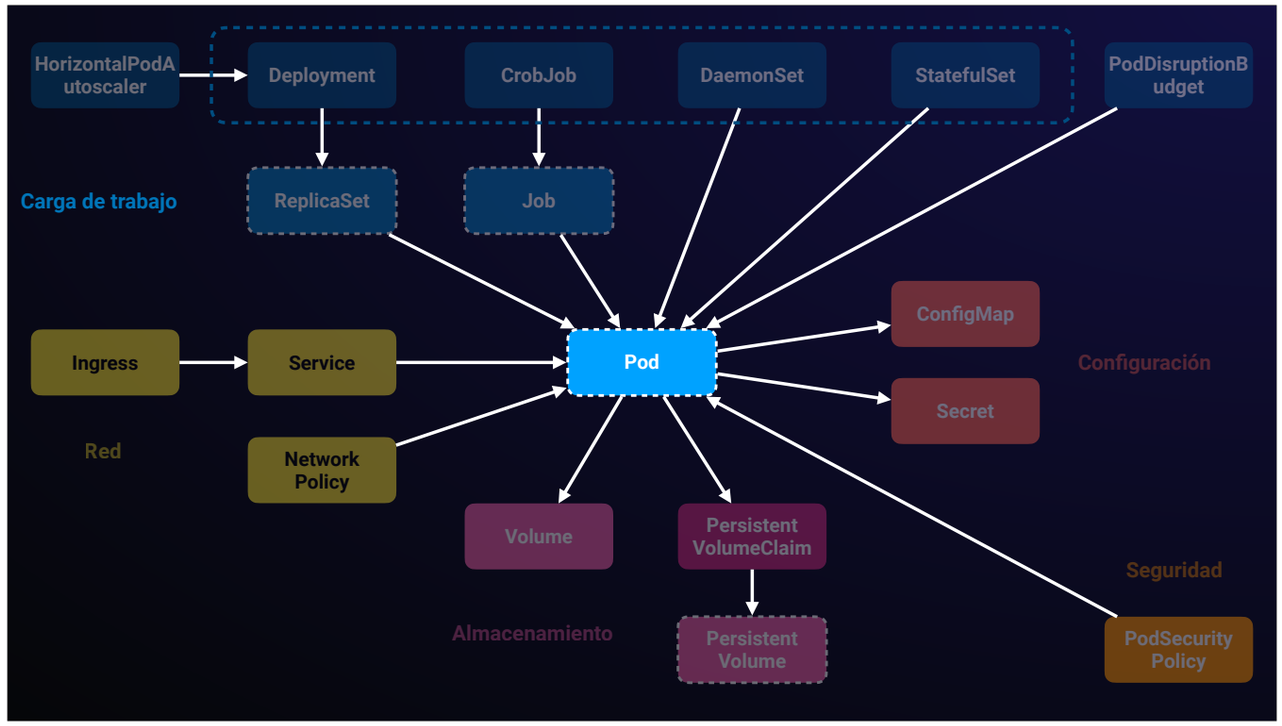
- Configuración **declarativa** del *cluster*, **versus imperativa** (ejecutar comandos a través de la interfaz de línea de comandos).
- Permite **definir** todos los aspectos de **infraestructura** a través de **especificaciones** (manifiestos) que se almacenan en el **repositorio** y **forman parte del proyecto de software**. Por tanto, la infraestructura también pasa a estar versionada.
- Fácilmente replicable.
- **Metadatos:** nombre del objeto (único para ese tipo de objeto en el espacio de nombres), nombre del espacio de nombres, etiquetas y anotaciones.
- **Especificación:** diferente para cada tipo de objeto.

Objetos: metadatos

- Además del **nombre** y del **espacio de nombres**, existen las etiquetas y anotaciones.
- Las **etiquetas** son **identificativas** y relacionan objetos entre sí.
- Las **anotaciones no** son **identificativas**.

```yaml
metadata:
  name: superapi-proxy
  namespace: backend
  labels:
    app.kubernetes.io/instance: superapi
    app.kubernetes.io/component: proxy
  annotations:
    kubernetes.io/change-cause: 'Release 1.1.0'
    prometheus.io/scrape: ‘true'
```

- Las **etiquetas relacionan objetos**. Por ejemplo, todos los objetos de una misma aplicación tendrán el mismo valor para la etiqueta `app.kubernetes.io/instance`. Hay etiquetas estandarizadas y recomendadas por Kubernetes, como esta en concreto.
- Tanto **etiquetas** como **anotaciones** pueden ser utilizadas para **seleccionar objetos**.
- Las **etiquetas** suelen ser **utilizadas por Kubernetes** y son **más eficientes**, mientras que las **anotaciones** suelen ser **utilizadas** por **herramientas externas**.

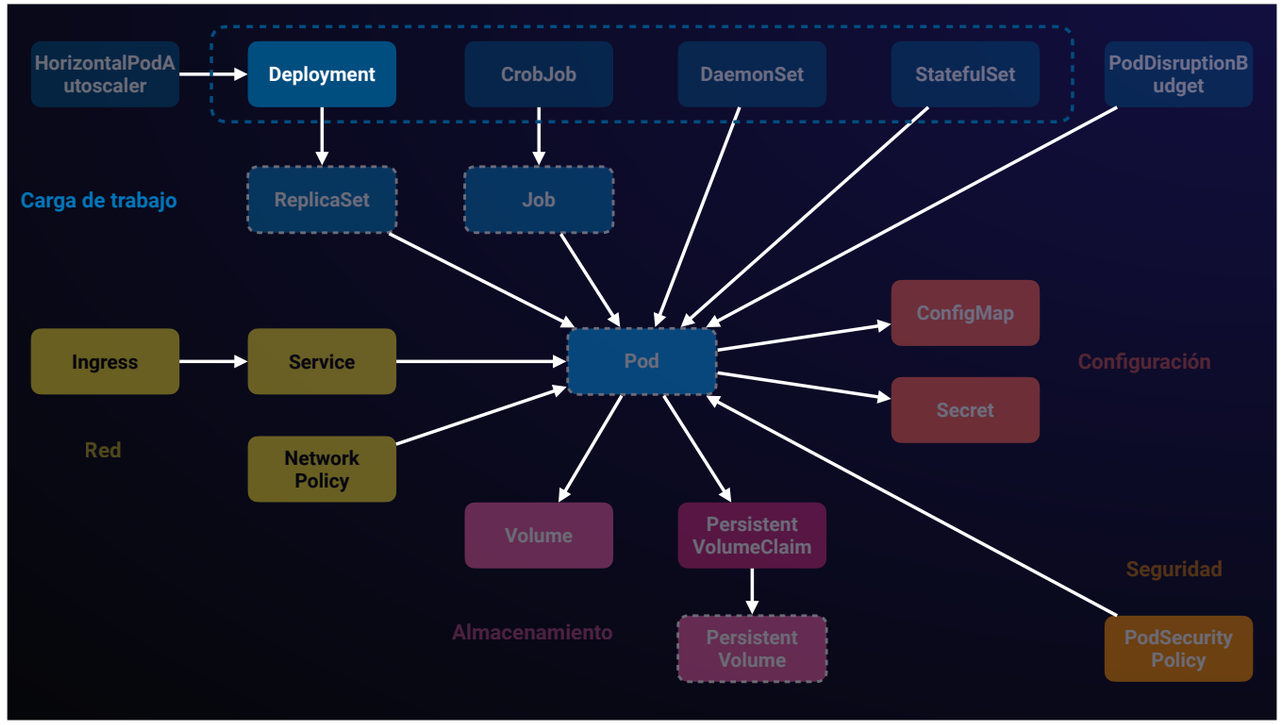

HorizontalPodAutoscaler
Deployment
CrobJob
DaemonSet
StatefulSet
PodDisruptionBudget
Carga de trabajo
ReplicaSet
Job
ConfigMap
Ingress
Service
Pod
Configuración
Secret
Red
Network Policy
Volume
Persistent VolumeClaim
Seguridad
Almacenamiento
Persistent Volume
PodSecurity Policy

Objetos: Pod

- Es el objeto **elemental** de Kubernetes y la **unidad mínima** de despliegue.
- **Encapsula** uno o varios **contenedores**.
- Tiene una **IP propia** privada dentro del *cluster*.
- No se trabaja directamente con ellos, sino con los **controladores**.

```yaml
kind: Pod
apiVersion: v1

metadata:
  name: hello
  namespace: backend
  labels:
    app.kubernetes.io/instance: hello

spec:
  containers:
    - name: default
      image: busybox
      command:
        - sh
        - '-c'
        - 'echo "Hello Kubernetes!" && sleep 3600'
```

- Un ***pod*** en **Kubernetes** es como un **servidor** en un **centro de datos,** pero **virtual**. Un **contenedor** sería como un **proceso** ejecutándose de forma aislada dentro de dicho servidor.
- Los contenedores de un *pod* **comparten** interfaz virtual de **red** y pueden compartir **volúmenes** de almacenamiento.
- Los controladores son una abstracción que permite trabajar con conjuntos de *pods* (ver más adelante).


HorizontalPodAutoscaler
Deployment
CrobJob
DaemonSet
StatefulSet
PodDisruptionBudget
Carga de trabajo
ReplicaSet
Job
ConfigMap
Ingress
Service
Pod
Configuración
Secret
Red
Network Policy
Volume
Persistent VolumeClaim
Seguridad
Almacenamiento
Persistent Volume
PodSecurity Policy

Objetos: Deployment

- Se encarga de mantener en ejecución un **conjunto** de *pods* idénticos (réplicas).
- Se asume que los *pods* deben estar en ejecución de forma permanente.
- Al hacer **cambios**, el controlador los aplica siguiendo una **estrategia**.
- Permite **deshacer cambios**.
- Incluye la especificación del *pod* dentro de él.

- El **controlador más sencillo** es el Deployment.
- Que estén en **ejecución permanente** significa que **únicamente** se espera que **terminen** su ejecución debido a un **error**.
- Hay **diferentes estrategias** de despliegue, que se verán a continuación.
- Mantiene un **histórico de cambios** y permite **deshacerlos** (*rollback*).
- La especificación del *pod* va dentro del *deployment*, por lo que **no se trabaja directamente con** objetos de tipo **Pod**. El **controlador** asociado al objeto **Deployment** es quien **se encarga** de todo.
- Desplegar aplicaciones de esta forma permite tener **SLAs** (Service Level Agreements) **muy buenos**, ya que hay una **alta tolerancia a fallos** (réplicas, reinicios automáticos).

Objetos: Deployment

```yaml
kind: Deployment
apiVersion: apps/v1

metadata:
  name: website
  namespace: frontend
  labels:
    app.kubernetes.io/instance: website

spec:
  replicas: 2
  revisionHistoryLimit: 4

  strategy:
    rollingUpdate:
      maxUnavailable: 1

  selector:
    matchLabels:
      app.kubernetes.io/instance: website
```

Pod

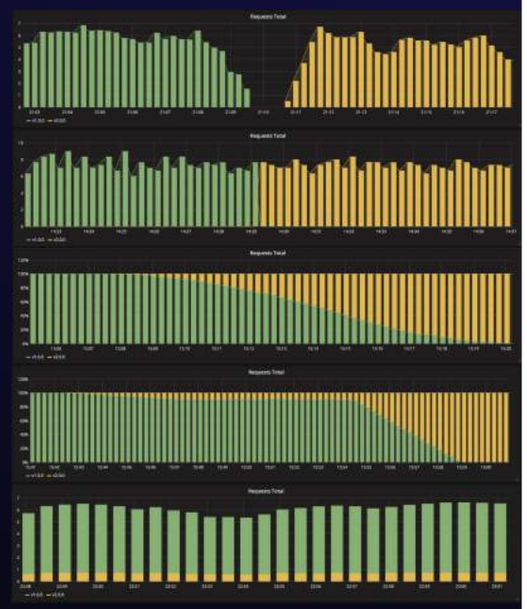
```yaml
  template:
    metadata:
      labels:
        app.kubernetes.io/instance: website

    spec:
      containers:
        - name: nginx
          image: nginx:1.18.0-alpine
          imagePullPolicy: Always
          ports:
            - name: http
              containerPort: 3000
          lifecycle:
            preStop:
              exec:
                command:
                  - nginx
                  - '-s'
                  - quit
```

```yaml
          resources:
            requests:
              cpu: 10m
              memory: 20Mi
            limits:
              cpu: 20m
              memory: 30Mi
          livenessProbe:
            tcpSocket:
              port: http
            initialDelaySeconds: 5
            timeoutSeconds: 2
```

- La **estructura básica** es la que se ha visto antes: *kind*, *apiVersion*, *metadata* y *spec*.
- El objeto ***spec* dentro de *template*** es la **especificación de un Pod**. Es decir, un objeto puede contener otros objetos en su definición.
- Las **réplicas** se pueden fijar **manualmente** o definir un **HorizontalPodAutoscaler** para automatizar el escalado.
- El **límite de recursos** es fundamental por cuestiones de **integridad del *cluster*** y **escalabilidad**. Notación de CPU y memoria.
- Nombre del contenedor, imagen y puertos.
- **Prueba de vida** (resiliencia). TCP, HTTP, ejecutar comando dentro del contenedor, etc.
- *Hooks*.
- Buena práctica: **fijar versiones**. Por razones de estabilidad y reproducibilidad.

Estrategias de actualización

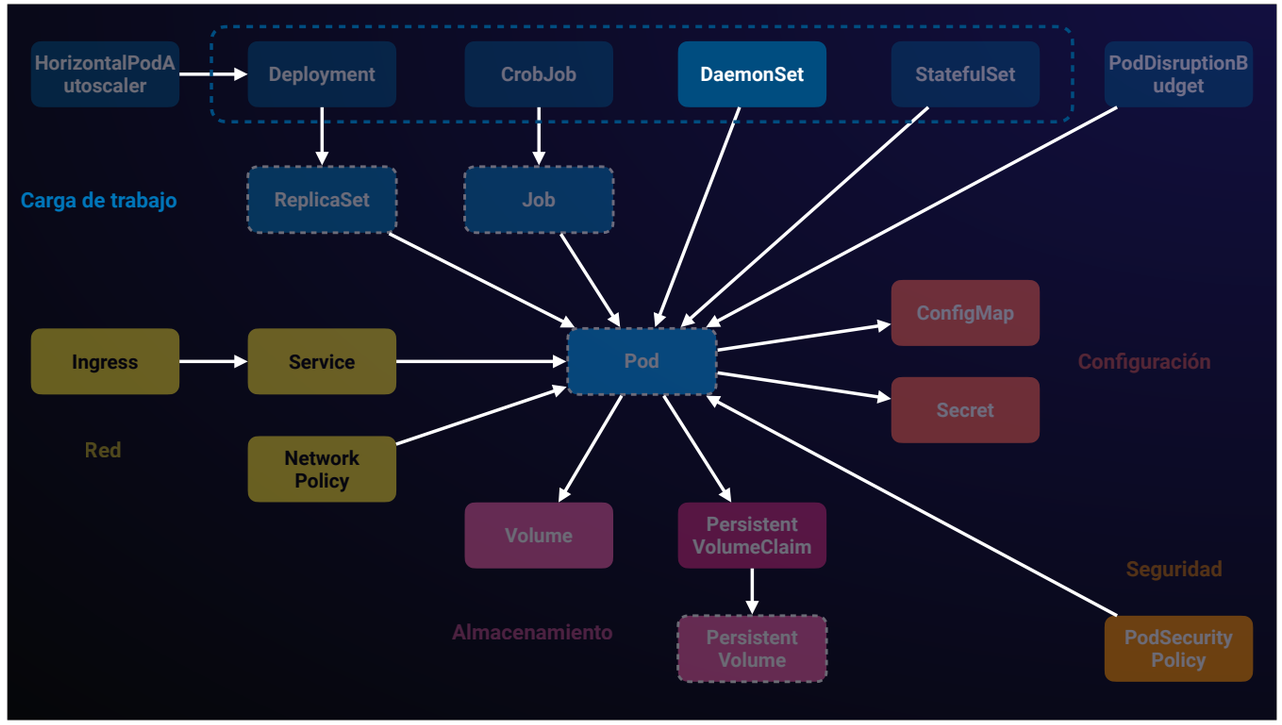
- ***Recreate*.** Se eliminan los *pods* viejos y a continuación se crean los nuevos.
- ***Blue/green*.** La operación inversa que con *recreate*.
- ***Rolling*.** Se van creando los *pods* nuevos mientras se van eliminando los viejos, todo de forma progresiva.
- ***Canary*.** Se crean *pods* nuevos para un subconjunto específico de usuarios y se va ampliando el conjunto hasta el total, de forma cada vez más acentuada.
- ***A/B testing*.** Se crean *pods* nuevos únicamente para un subconjunto de usuarios en base a algún criterio.



Fuente: Deployment Strategies on Kubernetes

La **estrategia** define **cómo** los nuevos *pods* **reemplazan** a los viejos:

- ***Recreate.*** Implica caída y tiene un alto impacto en el usuario, pero puede ser útil cuando ambas versiones no pueden convivir.
- ***Rolling.*** Sin caída: es totalmente transparente para el usuario. Eficiente con los recursos, pero tarda más. No se controla qué tráfico es dirigido a qué instancias.
- ***Blue/green.*** Orden inverso que recreate. El precio de eliminar el impacto al usuario es doblar el uso de recursos temporalmente. Útil cuando ambas versiones no pueden convivir.
- ***Canary.*** Es lento y más complejo, ya que requiere fijar el tráfico por usuario, cosa que no ocurre en el resto y que añade complejidad al balanceo de carga (no se puede hacer de forma nativa). Útil para cambios drásticos que requieren monitorización.
- ***A/B testing.*** También requiere fijar el tráfico. Útil cuando es necesario tener versiones en paralelo.


HorizontalPodAutoscaler
Deployment
CrobJob
DaemonSet
StatefulSet
PodDisruptionBudget
Carga de trabajo
ReplicaSet
Job
ConfigMap
Ingress
Service
Pod
Configuración
Secret
Red
Network Policy
Volume
Persistent VolumeClaim
Seguridad
Almacenamiento
Persistent Volume
PodSecurity Policy

Objetos: DaemonSet

- **Similar** a un objeto de tipo *deployment*.
- Tiene **tantas réplicas como nodos**.
- **Cada réplica** se ejecuta **en un nodo diferente**.
- Útil para **casos de uso específicos**, como:
  - Recolección de métricas de todos los nodos.
  - Ofrecer servicios dependientes del nodo.

- **Al igual que** un objeto de tipo **Deployment**, se encarga de **mantener** en ejecución un **conjunto de *pods*** de **larga vida**.
- Se **diferencia** en las **réplicas**, que en este caso es una **por nodo**.
- **Casos** de uso **muy específicos**, íntimamente relacionados con el nodo en sí.

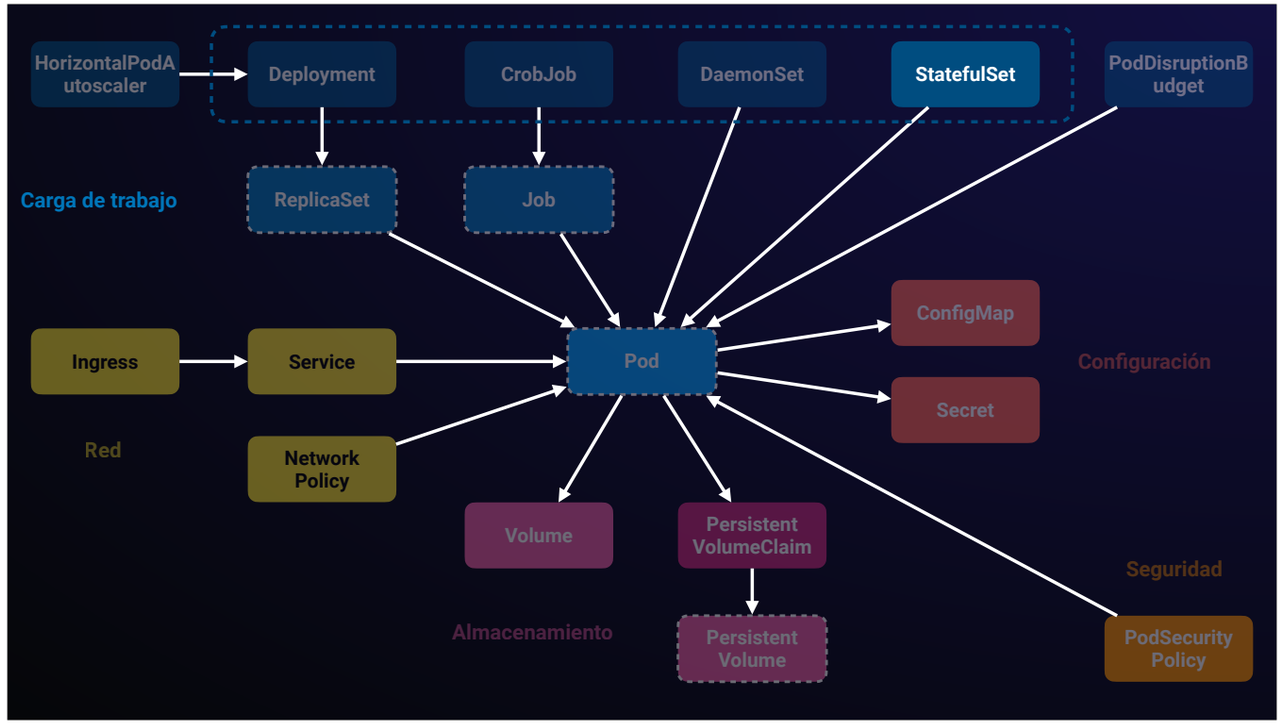
Objetos: DaemonSet

```yaml
kind: DaemonSet
apiVersion: apps/v1

metadata:
  name: node-exporter
  namespace: monitoring
  labels:
    app.kubernetes.io/instance: node-exporter

spec:
  selector:
    matchLabels:
      app.kubernetes.io/instance: node-exporter

  template:
    metadata:
      labels:
        app.kubernetes.io/instance: node-exporter
```

Pod

```yaml
spec:
  priorityClassName: system-node-critical

  containers:
    - name: default
      image: prom/node-exporter:v1.1.2
      args:
        - --path.procfs=/host/proc
        - --path.sysfs=/host/sys
      ports:
        - name: metrics
          containerPort: 9100
      resources:
        limits:
          cpu: 10m
          memory: 50Mi
        requests:
          cpu: 10m
          memory: 50Mi
```

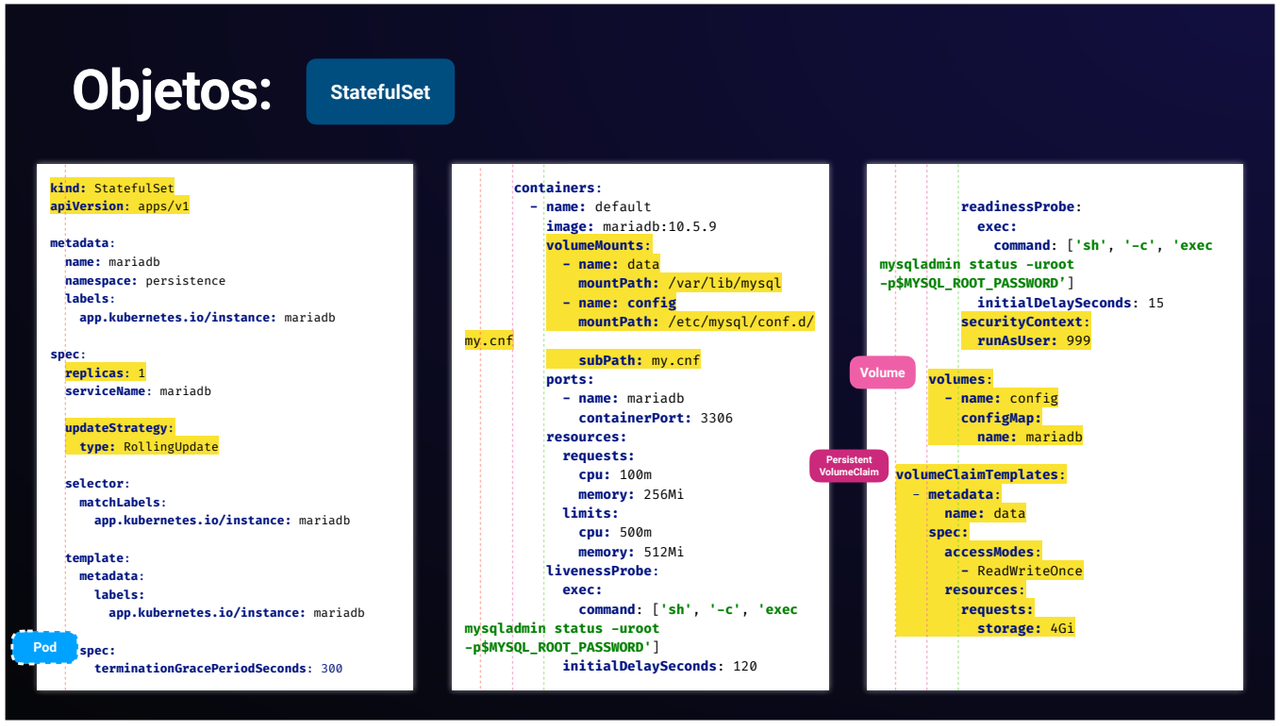
- **Misma estructura** que todos los objetos. Nada a destacar en ese sentido.
- Se aprovecha para explicar ***priorityClassName*** y ***args***, aunque se pueden usar en cualquier especificación de *pod*.


HorizontalPodAutoscaler
Deployment
CrobJob
DaemonSet
StatefulSet
PodDisruptionBudget
Carga de trabajo
ReplicaSet
Job
ConfigMap
Ingress
Service
Pod
Configuración
Secret
Red
Network Policy
Volume
Persistent VolumeClaim
Seguridad
Almacenamiento
Persistent Volume
PodSecurity Policy

Objetos: StatefulSet

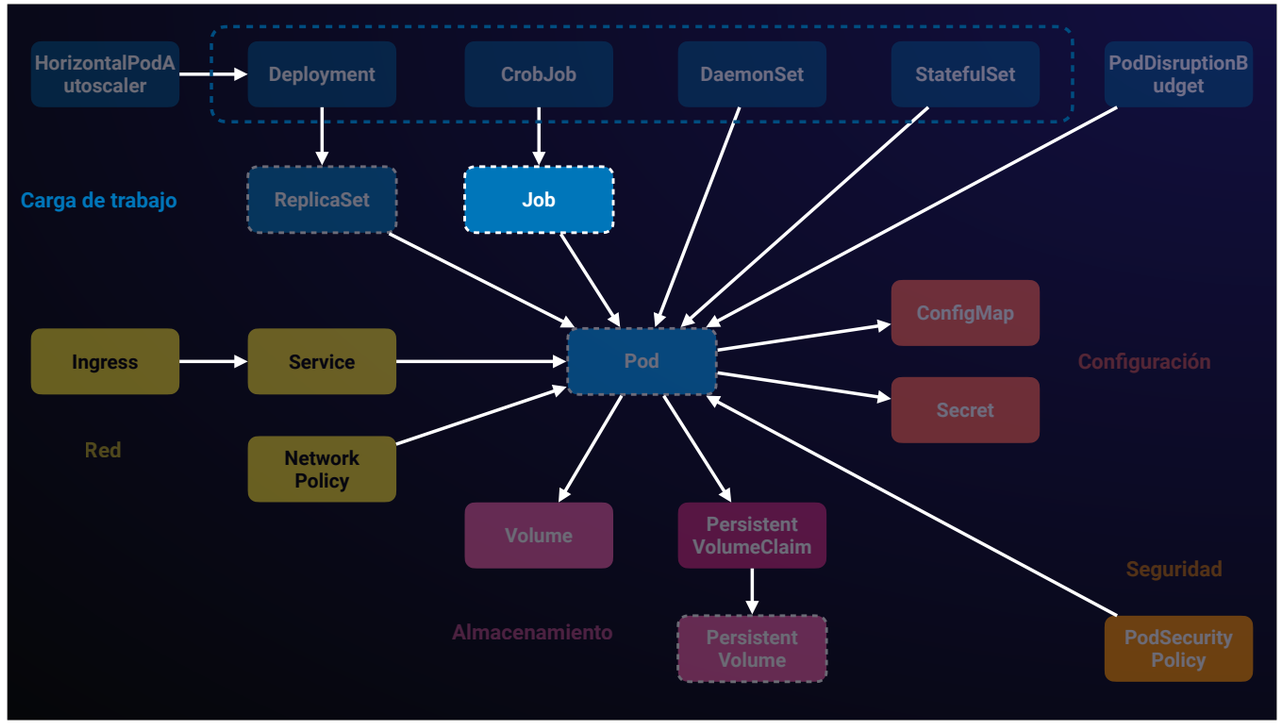
- **Similar** a un objeto de tipo *deployment* pero **con estado**.
- **Cada réplica** tiene una **identidad** propia y es **única**.
- Las **réplicas** se **crean**, **destruyen** y **escalan** de forma **ordenada**.
- Al tener estado, cada réplica tiene su propia **persistencia**.

- La **identidad** de los *pods* es **estable**, **no como con** los objetos **Deployment** y **DaemonSet**.
- Al igual que los **Deployment**, tiene estrategias de actualización.
- Tienen **persistencia** a través de **Volume**, **PersistentVolumeClaim** y **PersistentVolume**.



Inciso para explicar los volúmenes.

- **Volume.** Efímero y es posible compartirlo entre varios contenedores del mismo *pod*.
- **PersistentVolumeClaim.** Solicita a Kubernetes que se le provisione un volumen persistente (almacenamiento de bloque).
- **PersistentVolume.** Se crea a causa de un **PersistentVolumeClaim**. Hay diferentes **StorageClass**, para permitir elegir entre SSD NVME, SSD SATA o disco magnético, por ejemplo.


HorizontalPodAutoscaler
Deployment
CrobJob
DaemonSet
StatefulSet
PodDisruptionBudget
Carga de trabajo
ReplicaSet
Job
ConfigMap
Ingress
Service
Pod
Configuración
Secret
Red
Network Policy
Volume
Persistent VolumeClaim
Seguridad
Almacenamiento
Persistent Volume
PodSecurity Policy

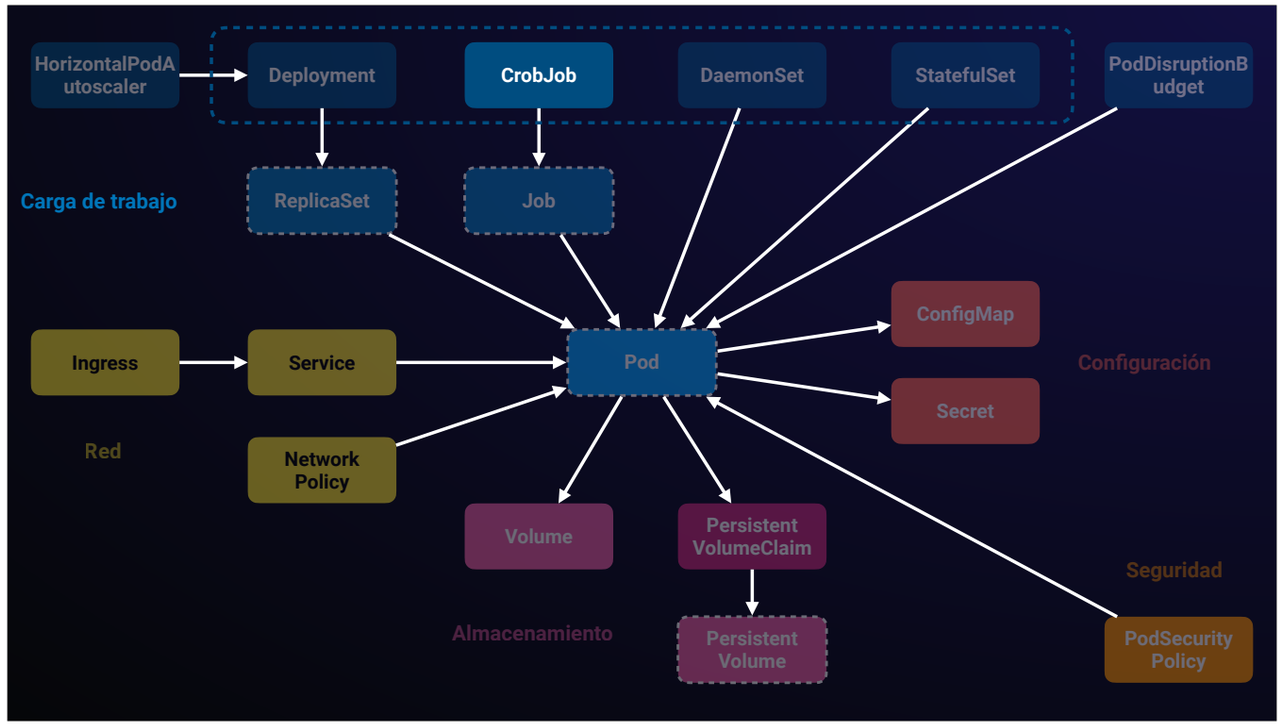
Objetos: Job

- Un objeto de tipo *job* **crea** uno o diversos *pods*.
- Se asume que **la ejecución debe terminar**.
- Se asegura que **terminan** su ejecución **correctamente**.

Pod

```yaml
kind: Job
apiVersion: batch/v1

metadata:
  name: pi

spec:
  backoffLimit: 4

  template:
    spec:
      restartPolicy: Never

      containers:
        - name: default
          image: perl
          command:
            - perl
            - -Mbignum=bpi
            - -wle
```

- Como un **Deployment** pero para cargas de trabajo de duración finita.
- Deben terminar y hacerlo correctamente (código de salida 0).


HorizontalPodAutoscaler
Deployment
CrobJob
DaemonSet
StatefulSet
PodDisruptionBudget
Carga de trabajo
ReplicaSet
Job
ConfigMap
Ingress
Service
Pod
Configuración
Secret
Red
Network Policy
Volume
Persistent VolumeClaim
Seguridad
Almacenamiento
Persistent Volume
PodSecurity Policy

Objetos: CronJob

- Un objeto de tipo *cron job* **crea** un ***job*** de forma **repetida** y **planificada** en el tiempo.
- Las repeticiones tienen la **misma sintaxis** que las tareas **Cron** de **UNIX**.
- Tienen bastantes **limitaciones**.
- **Ineficaces** para **repeticiones** muy **frecuentes** por la sobrecarga que supone el arranque.

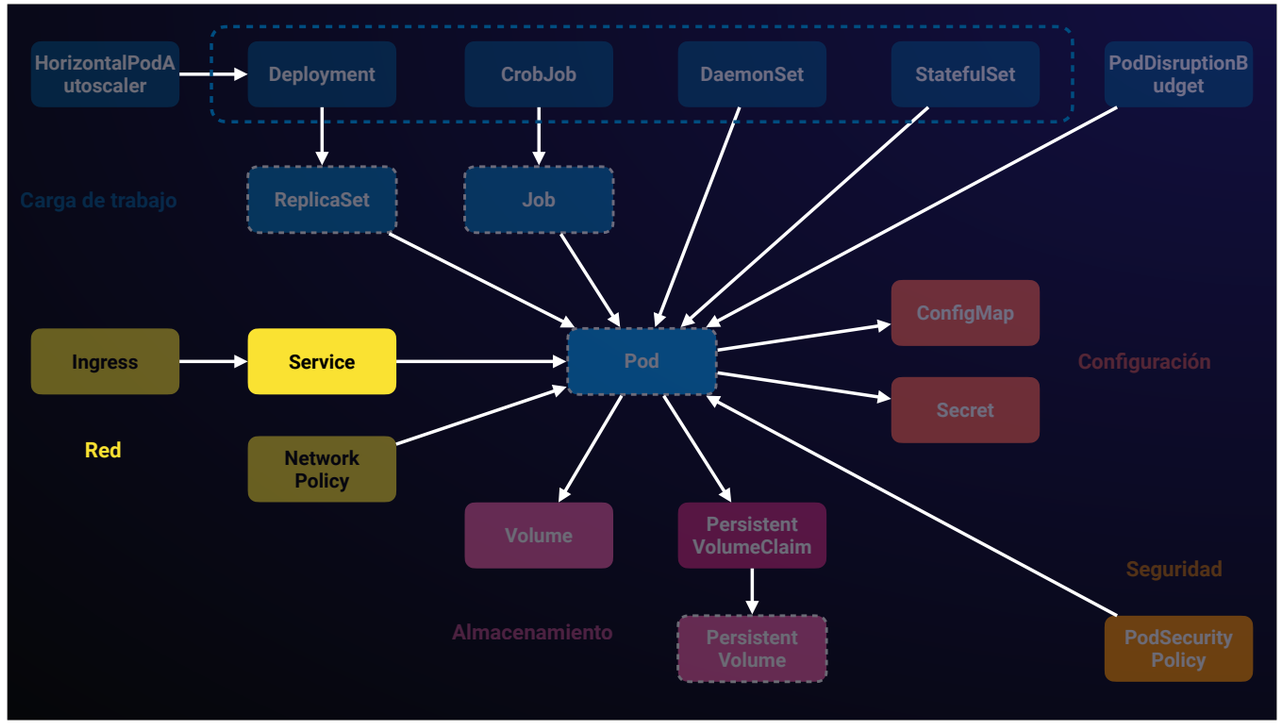
Job

Pod

```yaml
kind: CronJob
apiVersion: batch/v1beta1

metadata:
  name: hello

spec:
  schedule: '*/1 * * * *'
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: OnFailure

          containers:
            - name: default
              image: busybox
              args:
                - /bin/sh
                - -c
                - echo 'Hello from Kubernetes'
```

- **Limitaciones:** en ciertas circunstancias un CronJob puede lanzar diversos Job o ninguno. Por tanto, tienen que ser idempotentes. Es decir, que por muchas veces que un mismo Job se ejecute, el resultado final debería ser el mismo.
- **Ineficacia para repeticiones frecuentes:** crear un Job implica crear un Pod, lo cual implica arrancar un contenedor, que es costoso temporalmente hablando. Por ejemplo, en un CronJob repetido cada minuto puede no dar tiempo a que los Job arranquen y/o finalicen, por lo que se solaparían. A pesar de que hay mecanismos para evitar solapamiento, esto cambia sustancialmente la naturaleza del objeto.

  Hay otras soluciones para este caso, como tener un Pod con Cron funcionando y que vaya procesando las tareas.


HorizontalPodAutoscaler
Deployment
CrobJob
DaemonSet
StatefulSet
PodDisruptionBudget
Carga de trabajo
ReplicaSet
Job
ConfigMap
Ingress
Service
Pod
Configuración
Secret
Red
Network Policy
Volume
Persistent VolumeClaim
Seguridad
Almacenamiento
Persistent Volume
PodSecurity Policy

Objetos: Service

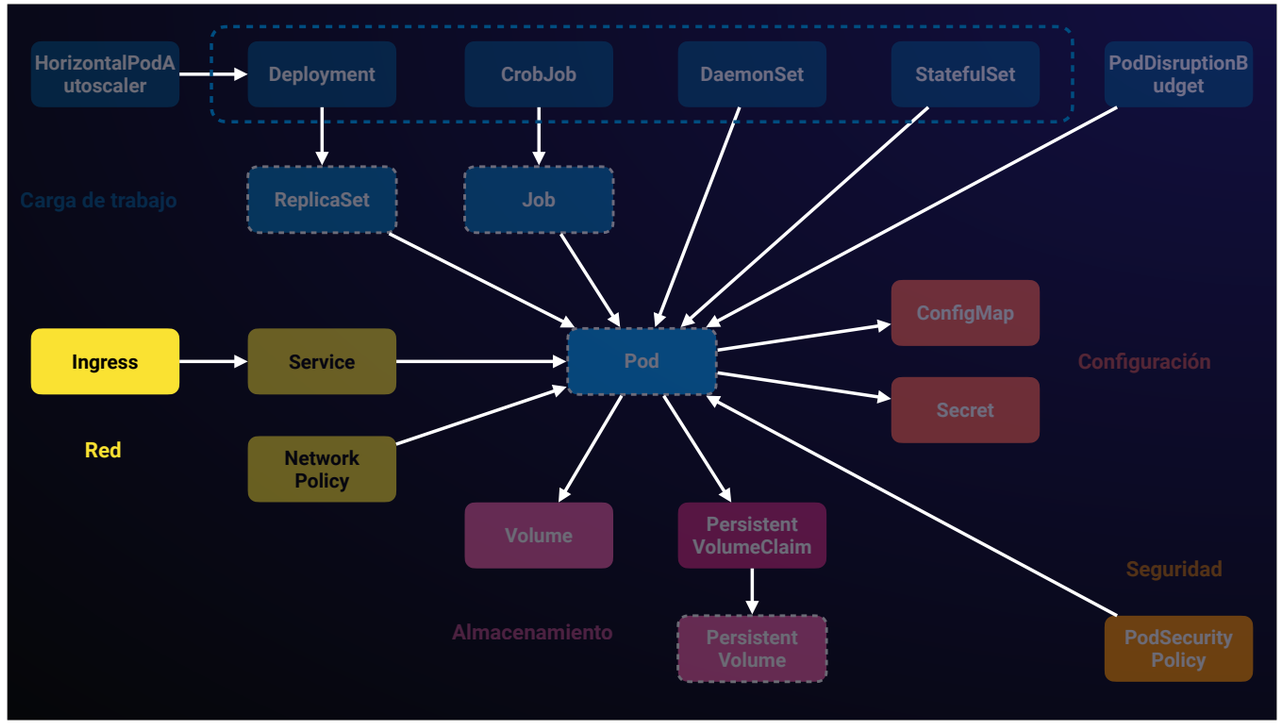
- Los ***pods*** son **efímeros**, por lo que sus **direcciones de red cambian**.
- Los **servicios exponen** conjuntos de ***pods*** bajo un **único nombre lógico**.
- Pueden actuar como **balanceadores de carga** muy primitivos.
- Son **imprescindibles** para **exponer *pods*** dentro y fuera del *cluster* a través de la red.
- Los hay de diferentes **tipos**: **ClusterIP**, **NodePort**, **LoadBalancer** y **ExternalName**.

- Un **servicio sabe** qué **pods** está exponiendo porque los selecciona a través de las **etiquetas**.
- Son **accesibles por DNS** desde dentro del *cluster*. El puerto que exponen no tiene por qué ser el mismo que el de los *pods*. Es decir, **tienen dirección de red (IP) propia**.
- Actúan como **balanceadores** de carga muy **básicos**, **excepto** cuando son de tipo ***headless*** (útil para *stateful sets*).
- Tipos:
  - **ClusterIP.** El servicio se expone en una IP interna del *cluster*. Valor por defecto.
  - **NodePort.** Igual que ClusterIP, pero además también lo expone en un puerto fijo de cada nodo (o puertos).
  - **LoadBalancer.** Igual que NodePort, pero además utiliza un proveedor externo para asignar un balanceador de carga físico y exponer el servicio a través de él.
  - **ExternalName.** Redirige a un servicio externo del *cluster*. En este caso no hay IP interna. Útil para apuntar a servicios de persistencia que están fuera.

Objetos: Service

```yaml
kind: Service
apiVersion: v1

metadata:
  name: website
  namespace: frontend
  labels:
    app.kubernetes.io/instance: website

spec:
  selector:
    app.kubernetes.io/instance: website

  ports:
    - name: http
      port: 80
      targetPort: http
```

```yaml
kind: Service
apiVersion: v1

metadata:
  name: mariadb
  namespace: persistence
  labels:
    app.kubernetes.io/instance: mariadb
  annotations:
    prometheus.io/scrape: 'true'

spec:
  selector:
    app.kubernetes.io/instance: mariadb

  clusterIP: None

  ports:
    - name: mariadb
      port: 3306
      targetPort: mariadb
```


HorizontalPodAutoscaler
Deployment
CrobJob
DaemonSet
StatefulSet
PodDisruptionBudget
Carga de trabajo
ReplicaSet
Job
ConfigMap
Ingress
Service
Pod
Configuración
Secret
Red
Network Policy
Volume
Persistent VolumeClaim
Seguridad
Almacenamiento
Persistent Volume
PodSecurity Policy

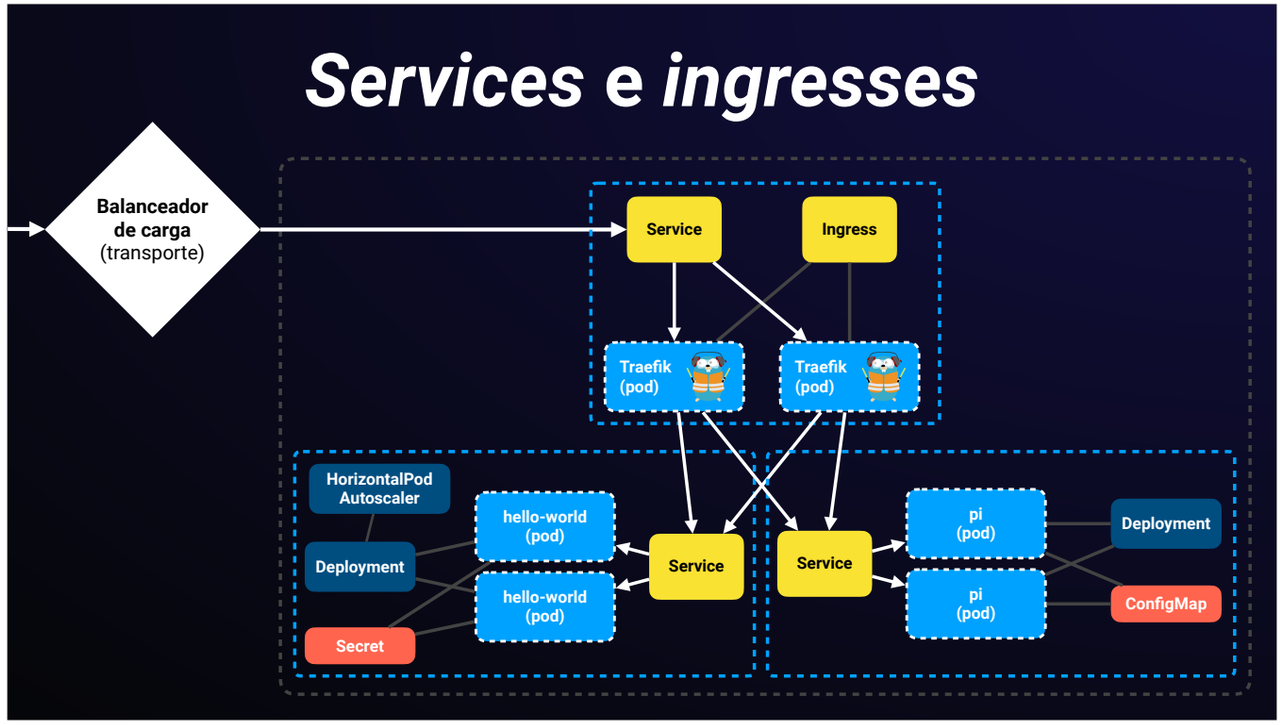
Objetos: Ingress

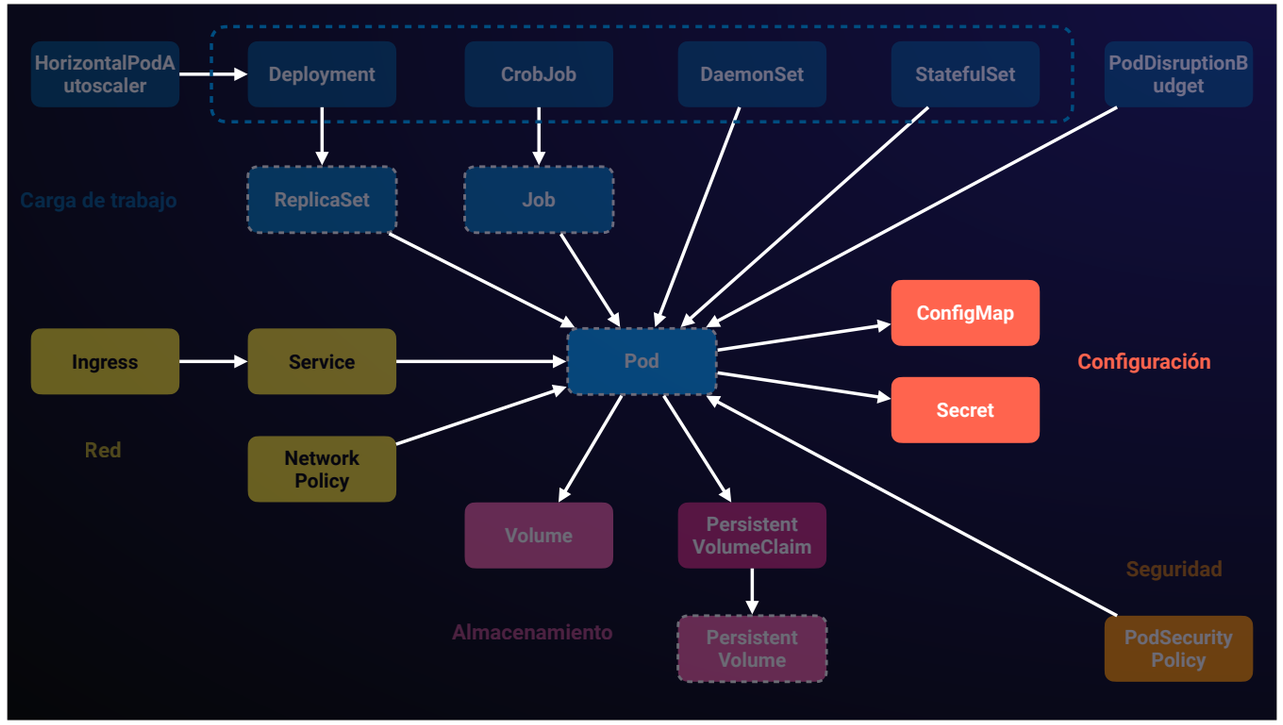
- Es una extensión que **nace** por la necesidad de **ahorrar** costes **en balanceadores** de carga.
- Es un **enrutador** de **alto nivel** (capa de aplicación).
- **Necesita** un **controlador** (un proxy inverso) que no viene por defecto (Traefik o Nginx, por ejemplo).
- **Requiere** un único **balanceador de carga externo** (nivel de transporte) por *cluster*, en lugar de uno por servicio.

```yaml
kind: Ingress
apiVersion: networking.k8s.io/v1

metadata:
  name: landing
  namespace: frontend
  labels:
    app.kubernetes.io/instance: landing
  annotations:
    traefik.ingress.kubernetes.io/router.entrypoints: h2

spec:
  rules:
    - host: brutal.systems
      http:
        paths:
          - backend:
              serviceName: landing
              servicePort: http
```

- **Exponer** todos los **servicios** al exterior **con balanceadores** de carga **externos** tiene un **coste altísimo**. Es un desaprovechamiento de recursos.
- Cuantos **más servicios** estén **en Kubernetes**, **más se aprovechan** los **recursos** de hardware.
- Un ***ingress* permite** tener un **único balanceador** de carga **externo** de nivel de transporte (TCP, UDP) como **punto de entrada** al *cluster*.
- Permite trabajar con rutas, no solo con la cabecera *host*.

Services e ingresses




HorizontalPodAutoscaler
Deployment
CrobJob
DaemonSet
StatefulSet
PodDisruptionBudget
Carga de trabajo
ReplicaSet
Job
ConfigMap
Ingress
Service
Pod
Configuración
Secret
Red
Network Policy
Volume
Persistent VolumeClaim
Seguridad
Almacenamiento
Persistent Volume
PodSecurity Policy

Objetos: ConfigMap Secret

- Ambos permiten **almacenar** variables de **configuración**.
- Los ***config maps*** almacenan **configuración no sensible** que puede (y debe) estar en los repositorios de software.
- Los ***secrets*** almacenan **configuración sensible** como contraseñas, *tokens* o claves.
- Los **contenedores no deben contener** ningún tipo de **configuración**. Esta debe ser **inyectada** en los ***pods*** en tiempo de ejecución.

La **configuración forma parte del entorno**, no del software, por lo que no debería distribuirse junto con él. Sí que es recomendable que resida en el repositorio junto al código fuente, especialmente si se quiere trabajar con infraestructura como código (IaC).

Es un **problema** grave de **seguridad** construir **contenedores** Docker que contengan **secretos**. Dicha configuración debería estar guardada siempre en *secrets*, los cuales no deberían versionarse.

Tampoco deberían guardarse **secretos** en **texto plano** en los **repositorios** de software. Hay alternativas, como **cifrarlos** con una clave que conozca el software de CI/CD (para poder aplicar los manifiestos en los despliegues) o no guardar nada en los repositorios, en cuyo caso puede hacerse que la gestión de secretos sea **manual** o que se inyecten automáticamente con algún software del estilo de **Hashicorp Vault**.

Objetos: ConfigMap Secret

```yaml
kind: ConfigMap
apiVersion: v1

metadata:
  name: superapi
  namespace: backend
  labels:
    app.kubernetes.io/instance: superapi

data:
  APP_ENV: production
  APP_URL: https://superapi.example.com
  DB_HOST: mariadb-0.vlc1.example.net
  DB_DATABASE: superapi
  DB_USERNAME: superapi
  REDIS_HOST: redis-0.vlc1.example.net
```

```yaml
kind: Secret
apiVersion: v1

metadata:
  name: superapi
  namespace: backend
  labels:
    app.kubernetes.io/instance: superapi

type: Opaque

stringData:
  APP_KEY: v3ryS3cr3tK3y
  DB_PASSWORD: dbP4ssw0rd
  AWS_SECRET_ACCESS_KEY: 4wsS3cr3t
  RECAPTCHA_SECRET_KEY: r3c4ptch4S3cr3t
  STRIPE_SECRET: str1p3S3cr3t
  STRIPE_ENDPOINT_SECRET: an0th3rOn3
```

Como **curiosidad**, los ***secrets*** almacenan la **información** codificada en **base 64** porque **pueden contener datos en binario** (como claves) y que, por tanto, no son sintácticamente válidos en YAML.

Sin embargo, es posible guardarlos sin codificar dentro del mapa `stringData` y que Kubernetes se encargue de ello.

Kubectl

- **Interfaz de línea de comandos** (CLI) para interactuar con la API de Kubernetes.
- Permite crear objetos y aplicar cambios en los mismos dada su especificación (**configuración declarativa**).
- También es posible interactuar con el *cluster* a través de comandos específicos (**configuración imperativa**).

Herramientas de Kubernetes.

Helm

- **Gestionar** los objetos **manualmente** es **tedioso** y **repetitivo**.
- **Helm** es el **gestor de paquetes** de Kubernetes.
- Los **paquetes** se llaman ***charts*** y tienen una estructura concreta.
- Permite el uso de **plantillas** para las especificaciones de los objetos.
- Dispone de un **repositorio** de paquetes **oficial**.
- Es recomendable utilizarlo únicamente como software de plantillas en el lado del cliente, aunque ofrece otras opciones.

Tiene dos usos posibles: como gestor de paquetes completo o como sistema de templating. La segunda opción permite utilizar todo el potencial de ambas herramientas sin acoplarlas.

cert-manager

- Es un controlador nativo para Kubernetes que se encarga de la **gestión de certificados** X.509.
- Utiliza el **protocolo ACME** para emitir y renovar certificados automáticamente.
- Hace uso de las definiciones personalizadas de recursos de Kubernetes (**CRD**).
- Guarda las **claves privadas en secretos** que otro software, normalmente el *ingress controller*, puede consumir.
- Se integra con **proveedores de DNS** para las **verificaciones**.

Herramientas del ecosistema.

- Al utilizar ACME, es compatible con **Let's Encrypt**.
- Sigue la misma filosofía de **conciliación** que Kubernetes: dados los certificados deseados, se encarga de mantenerlos vigentes.
- Kubernetes permite definir recursos (Pod o Deployment lo son) personalizados mediante las **CRD**. Por ejemplo, define el recurso **Certificate**.
- También se integra con **proveedores de DNS** para la automatización de las **verificaciones** (como CloudFlare o Route 53).
- Es compatible con ***wildcards***.

Traefik Proxy

- Es un ***ingress controller***, es decir, el ***proxy* inverso** que recibe y enruta las peticiones HTTP que llegan a un *cluster*.
- **Intregración nativa** con Kubernetes: lee los objetos ***ingress*** y enruta hacia los servicios según su especificación.
- Trabaja en las **capas** de **aplicación** y de **transporte**.
- Es el **punto de terminación TLS**. Usa los certificados que *cert-manager* deja en los secretos.
- Hay otras opciones, como Nginx, HAProxy, Istio o Envoy.

- **Descubre la configuración** automáticamente de los propios objetos de Kubernetes.
- Tiene ***middlewares*** que permiten añadir **cabeceras** (como HSTS), automatizar **redirecciones** (por ejemplo, para quitar “www.” o forzar TLS), añadir **autenticación** (básica HTTP o de proveedores externos) o filtrar tráfico por listas de IPs, entre muchos otros.
- Exporta **métricas de Prometheus**.
- Se integra con **Jaeger** para ***tracing***.
- Es compatible con **PROXY protocol**. Lo normal es que haya un balanceador de carga físico por delante, de forma que con este protocolo puede obtener la dirección real de la petición de forma estándar y eficiente.
- Compatible con **TLS 1.3**.
- Escrito en **Go**.

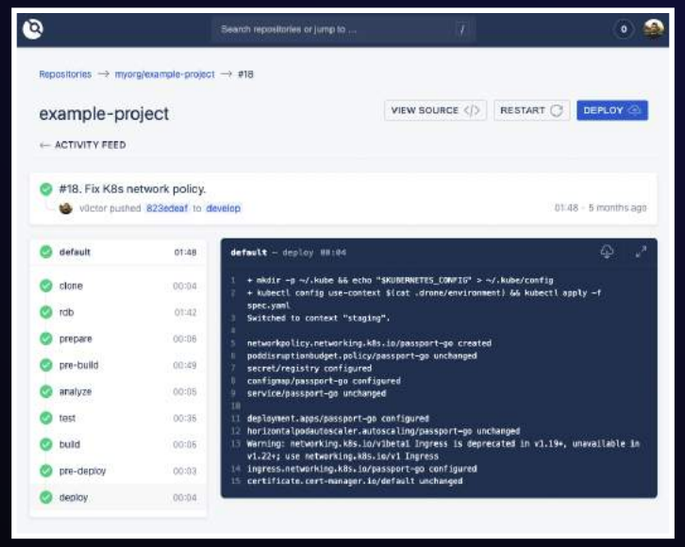
Drone

- Es un **software de integración continua** que se puede desplegar en Kubernetes. También puede ejecutar las *pipelines* en el *cluster*.
- **Basado en contenedores.** Cada paso de la *pipeline* son una serie de comandos ejecutados en un contenedor.
- Las ***pipelines*** se definen en **YAML** y se guardan en el repositorio de software.
- Únicamente requiere un servidor externo con Docker Engine para la construcción de imágenes.

Ventajas frente a otras alternativas como Jenkins:
- Facilidad de despliegue.
- Simplicidad a la hora de configurarlo y mantenerlo.
- Persistencia en disco o en una base de datos.
- Integración total con GitHub, GitLab y otros. Crea, mantiene y elimina los *webhooks* automáticamente para lanzar las *pipelines* cuando proceda.
- Compatible con *plugins* y extensiones.

Construir imágenes Docker estando dentro de un contenedor Docker tiene implicaciones de seguridad, ya que requiere montar el *socket* Docker de la máquina huésped. Sin embargo, es trivial desplegar una máquina independiente con Docker Engine que sea utilizada por los pasos de las *pipelines* que construyan imágenes. También existen otras soluciones como Kaniko, que permite construir imágenes dentro de Docker de forma segura, pero son considerablemente más complejas e implican el uso de herramientas de terceros.

Drone



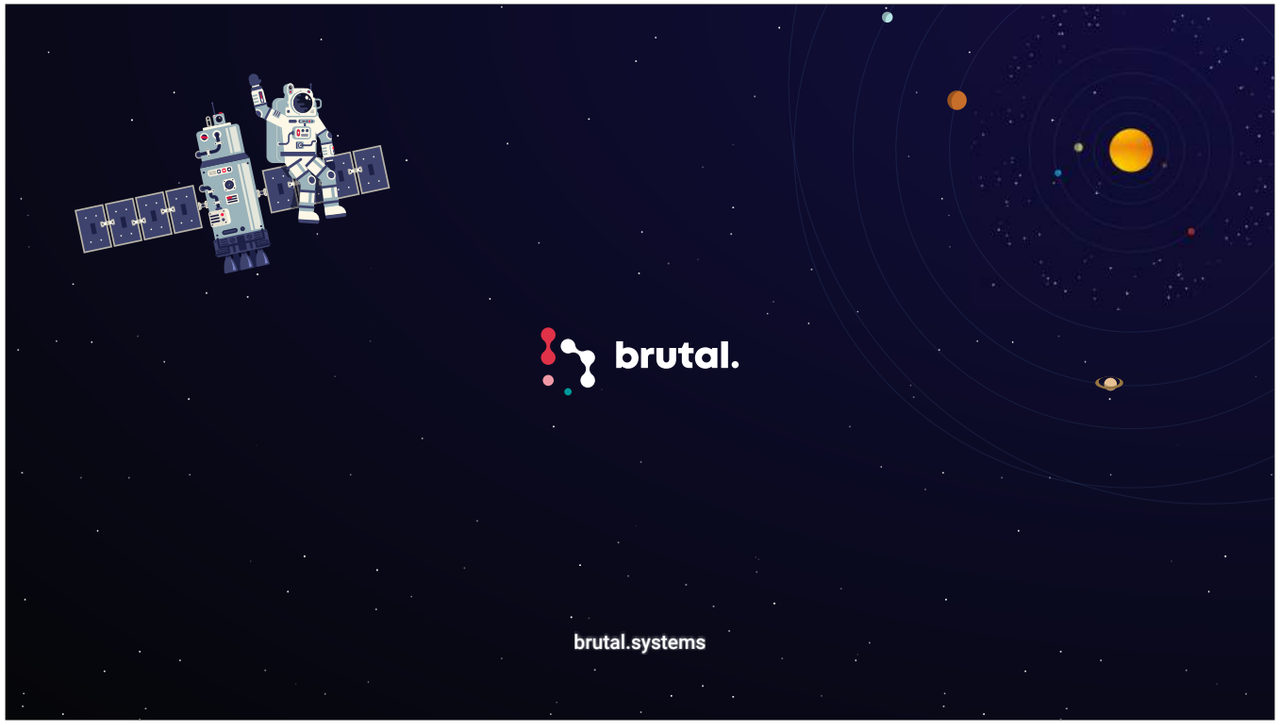
Ejemplo de *pipeline* que provisiona una base de datos (para las pruebas funcionales), realiza análisis estático, pasa las pruebas (unitarias y funcionales) y despliega el software en el *cluster* Kubernetes correspondiente en función de la rama Git y, por tanto, del entorno (preproducción o producción).

De esta forma, el software está autocontenido, es decir, contiene todo lo necesario para funcionar y ser desplegado en cualquier entorno:

- El código fuente.
- El entorno (o entornos) Docker.
- Los manifiestos para orquestar el software en local (con Docker Compose) y en entornos de preproducción o producción (con Kubernetes).
- Las instrucciones de despliegue, es decir, la *pipeline* de Drone.

Conclusiones

- La **infraestructura como código** (IaC) permite tratar la infraestructura como una parte más del software.
- **Kubernetes y su ecosistema** proporcionan herramientas de alta calidad, con licencia libre y de código abierto para implementar IaC.
- Las herramientas presentadas permiten ofrecer servicios **altamente disponibles**, **resilientes** y **escalables**.
- Gran parte de los servicios de las organizaciones puede ser desplegado en Kubernetes, lo que supone un **ahorro de costes** muy significativo al reservar solo los recursos que se necesitan.
- La **automatización** que aportan estas prácticas proporciona un ahorro de tiempo que se puede **reinvertir** en más **innovación**.


brutal.
brutal.systems