

O'REILLY®

# Infraestructura nativa de la nube con Azure

Crear y administrar aplicaciones nativas de la nube



Nishant Singh y Michael Kehoe

# Infraestructura nativa de la nube con Azure

La nube se está convirtiendo en el hogar de facto para las empresas, desde las grandes organizaciones hasta las empresas emergentes.

Migrar a la nube significa migrar sus aplicaciones de monolito a microservicios. Pero una vez que lo hace, ejecutar y mantener estos servicios conlleva su propio nivel de complejidad. ¿La respuesta? Modularidad, capacidad de implementación, observabilidad y capacidad de recuperación automática a través del desarrollo nativo de la nube.

Con este libro práctico, Nishant Singh y Michael Kehoe le muestran cómo crear una verdadera infraestructura nativa de nube utilizando Microsoft Azure u otra solución de informática en la nube siguiendo las directrices de la Cloud Native Computing Foundation (CNCF). Los ingenieros de DevOps y confiabilidad del sitio aprenderán cómo adaptar las aplicaciones a la nube nativa en las primeras etapas del diseño les ayuda a aprovechar por completo la elasticidad y la naturaleza distribuida de la nube.

**Este libro le ayuda a explorar los siguientes temas:**

- **¿Por qué usar la nube nativa?**
- **Cómo usar la infraestructura como código**
- **Qué necesita para usar contenedores para una aplicación**
- **Por qué y cómo Kubernetes es el "gran orquestador"**
- **Cómo crear un clúster de Kubernetes en Azure**
- **Cómo la observabilidad complementa la supervisión**
- **Cómo usar la detección de servicios y una malla de servicios para descubrir nuevos territorios**
- **Cómo las redes y la administración de directivas actúan como guardianes**
- **Cómo funcionan las bases de datos distribuidas y el almacenamiento**

"La referencia de Azure que debe tener en su escritorio. Este libro desmitifica cómo crear aplicaciones de la manera correcta para la nube".

—Todd Palino  
Ingeniero principal de personal, LinkedIn,  
coautor de *Kafka: The Definitive Guide*

Nishant Singh es ingeniero senior de confiabilidad de sitios en LinkedIn donde trabaja para mejorar la confiabilidad del sitio con un enfoque en reducir el tiempo medio de detección (MTTD) y el tiempo medio de respuesta (MTTR) frente a incidentes. Anteriormente, fue ingeniero de DevOps en Paytm y Gemalto donde dedicaba su tiempo a la creación de soluciones personalizadas para los clientes y la administración y el mantenimiento de servicios a través de la nube pública.

Michael Kehoe es ingeniero senior de seguridad del personal en Confluent. Anteriormente, fue ingeniero senior de confiabilidad de sitios de personal en LinkedIn, donde trabajó en la respuesta ante incidentes, la recuperación ante desastres, la ingeniería de visibilidad y los principios de confiabilidad, y lideró las iniciativas de la empresa para automatizar la migración a Microsoft Azure.

## INFORMÁTICA EN LA NUBE

US \$59.99 CAN \$74.99

ISBN: 978-1-492-09096-0



9 781492 090960

Twitter: @oreillymedia  
[linkedin.com/company/oreilly-media](https://www.linkedin.com/company/oreilly-media)  
[youtube.com/oreillymedia](https://www.youtube.com/oreillymedia)

---

# Infraestructura nativa de la nube con Azure

*Crear y administrar  
aplicaciones nativas de la nube*

*Nishant Singh y Michael Kehoe*

Pekín • Boston • Farnham • Sebastopol • Tokio

O'REILLY®

## **Infraestructura nativa de la nube con Azure**

por Nishant Singh y Michael Kehoe

Copyright © 2022 Nishant Singh y Michael Kehoe. Todos los derechos reservados.

Publicado en los Estados Unidos de América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Es posible que se compren libros de O'Reilly para uso promocional educativo, empresarial o de ventas. Las ediciones en línea para la mayoría de los títulos también están disponibles (<http://oreilly.com>). Para obtener más información, póngase en contacto con nuestro Departamento de ventas corporativo/institucional: 800-998-9938 o [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editora de adquisiciones:** Jennifer Pollock

**Indexadora:** Ellen Troutman-Zaig

**Editora de desarrollo:** Rita Fernando

**Diseñador de interiores:** David Futato

**Editor de producción:** Christopher Faucher

**Diseñadora de portada:** Karen Montgomery

**Correctora de estilo:** Audrey Doyle

**Ilustradora:** Kate Dullea

**Correctora:** Kim Cofer

Febrero de 2022: Primera edición

### **Historial de revisiones para la primera edición**

09-02-2022: Primera versión

Consulte <http://oreilly.com/catalog/errata.csp?isbn=9781492090960> para obtener detalles de la versión.

El logotipo de O'Reilly es una marca registrada de O'Reilly Media, Inc. *La infraestructura nativa de la nube con Azure*, la imagen de portada y la imagen comercial relacionada son marcas comerciales de O'Reilly Media, Inc.

Las opiniones expresadas en este trabajo pertenecen a los autores y no representan las opiniones de la editorial. Si bien la editorial y los autores han realizado esfuerzos de buena fe para garantizar que la información y las instrucciones contenidas en este trabajo sean correctas, la editorial y los autores renuncian a toda responsabilidad por errores u omisiones, incluida sin limitación la responsabilidad por los daños resultantes del uso o la dependencia de este trabajo. Si usa la información y las instrucciones contenidas en este trabajo, es bajo su propia responsabilidad. Si algún ejemplo de código u otra tecnología que este trabajo contiene o describe está sujeto a licencias de open source o a derechos de propiedad intelectual de otros, es su responsabilidad asegurarse de que el uso de los mismos cumpla con dichas licencias y/o derechos.

978-1-492-09096-0

[LSI]

---

# Contenido

<b>Prólogo .....</b>	<b>xí</b>
<b>1. Introducción: ¿Por qué la nube nativa? .....</b>	<b>1</b>
El viaje a la nube	1
Desafíos en la nube	2
Cloud Native Computing Foundation	4
Adopción de una infraestructura nativa de la nube con Azure	4
Resumen	5
<b>2. Infraestructura como código: Configuración del gateway.....</b>	<b>7</b>
Infraestructura como código y su importancia en el mundo nativo de la nube	8
Introducción a Azure y configuración del entorno	11
Fundamentos de Azure y preparación del entorno de Azure	11
Creación de una cuenta de Azure	12
Instalación de la CLI de Azure	13
Herramientas de IaC destacadas	13
Terraform	14
Packer	29
Ansible	31
Azure DevOps y la infraestructura como código	33
Resumen	33
<b>3. Contenedorizar su aplicación:Más que cajas.....</b>	<b>35</b>
¿Por qué usar contenedores?	35
Aislamiento	36
Seguridad	36
Empaque y implementación	37

Primitivos de contenedores básicos	37
Cgroups	38
Espacios de nombre	39
Copy on Write	40
Capacidades	40
Seccomp-BPF	40
Componentes de la ejecución de un contenedor	40
Orquestadores de contenedores	41
Software de contenedores	41
Tiempos de ejecución de contenedores	42
Contenedores	43
Sistema operativo	43
Especificación de Open Container Initiative (OCI)	43
Especificación de imagen de OCI	44
Especificación de tiempo de ejecución de OCI	45
Docker	46
Creación de su primera imagen de Docker	46
Procedimientos recomendados para usar Docker	48
Otras plataformas de contenedores	49
Contenedores Kata	49
LXC y LXD	49
Registros de contenedores	50
Almacenamiento seguro de imágenes con Harbor	51
Almacenamiento seguro de imágenes con Azure Container Registry	55
Almacenamiento de imágenes de Docker en un registro	59
Ejecución de Docker en Azure	60
Azure Container Instances	60
Implementación de una Azure Container Instance	61
Ejecución de un motor de contenedores de Docker	65
Resumen	66
<b>4. Kubernetes: El gran orquestador.....</b>	<b>67</b>
Componentes de Kubernetes	69
Plano de control	70
Nodos de trabajo	71
Objetos del servidor de API de Kubernetes	72
Pods	72
ReplicaSets	73
Implementaciones	73
Servicios	73
Espacios de nombre	74

Etiquetas y selectores	74
Anotaciones	74
Controlador de ingreso	74
StatefulSets	75
DaemonSets	75
Trabajos	75
Observe, opere y administre clústeres de Kubernetes con kubectl	76
Información general y comandos del clúster	76
Administración de pods	78
Kubernetes en producción	85
Resumen	98
<b>5. Creación de un clúster de Kubernetes en Azure.....</b>	<b>99</b>
Creación de un clúster de Kubernetes desde cero	99
Creación de un grupo de recursos	100
Creación de imágenes para máquinas de trabajo y controladores	100
Creación del back-end de la cuenta de almacenamiento	101
Creación de una red virtual de Azure	102
Creación de direcciones IP públicas para el equilibrador de cargas	102
Creación de instancias de controlador y trabajo	104
Uso de Ansible para implementar y configurar los nodos del controlador de Kubernetes	106
Uso de Ansible para implementar y configurar los nodos de trabajo de Kubernetes	109
Configuración del enrutamiento y la red del pod	109
Generar el archivo kubeconfig para el acceso remoto y la validación del clúster	110
Azure Kubernetes Service	111
Implementación de aplicaciones y servicios mediante Helm:	
un administrador de paquetes para Kubernetes	113
Aspectos básicos de Helm	114
Instalación y administración de Helm	114
Administración de lanzamientos de Helm	117
Creación de gráficos para sus aplicaciones	118
Resumen	120
<b>6. Observabilidad: Seguir las huellas.....</b>	<b>121</b>
Introducción a la observabilidad	121
Observabilidad: más que tres pilares	122
Observabilidad: Un superconjunto de supervisión	123
Desarrollo impulsado por la observabilidad	124
Supervisión de métricas con Prometheus en un mundo nativo de la nube	125
Componentes y arquitectura de Prometheus	125
Instalación y configuración de Prometheus	127

node_exporter	129
Instrumentación de aplicaciones	130
Encontrar hosts	134
Prometheus en Kubernetes	136
Crear registros en el mundo nativo de la nube	138
Crear registros con Fluentd	138
Fluentd en Kubernetes	146
Seguimiento distribuido en el mundo nativo de la nube	150
Seguimiento: Conceptos clave	151
Arquitectura del sistema de seguimiento general y ensamblado de seguimiento	153
Normas de seguimiento, herramientas e instrumentación de código	154
Azure Monitor	159
Resumen	161
<b>7. Detección de servicios y malla de servicios: Descubrir nuevos territorios y cruzar fronteras.....</b>	<b>163</b>
Detección de servicios	164
Introducción a CoreDNS	165
Instalación y configuración de CoreDNS	167
Detección de servicios de Kubernetes con CoreDNS	169
Azure DNS	171
La malla de servicios	172
Introducción a Istio	174
Instalación de Istio en Azure Kubernetes Service	175
Inserción automática del proxy sidecar (proxy Envoy)	177
Administración de mallas de servicio de Istio con Kiali	179
Resumen	187
<b>8. Redes y administración de políticas:He aquí los guardianes.....</b>	<b>189</b>
La interfaz de red de contenedores (CNI)	190
¿Por qué usar una CNI?	191
¿Cómo funciona la CNI con Azure?	191
Diversos proyectos de CNI	192
Calico	193
¿Por qué usar Calico?	193
Arquitectura básica	194
Implementación de Calico	195
Análisis detallado de Calico	197
Implementación de la política de seguridad de Calico	198
Cilium	200
Implementación de Cilium	201
Integración de Cilium con la nube	204

Flannel	207
Implementación de Flannel	207
Análisis detallado de Flannel	208
Azure Policy	210
Inicio rápido de Azure Policy	210
Creación de su propia política de Azure	212
Azure Policy para Kubernetes	213
Open Policy Agent	214
Implementación de OPA en Kubernetes	215
Implementación de políticas con OPA	216
Resumen	218
<b>9. Almacenamiento y bases de datos distribuidas: El Banco Central.....</b>	<b>219</b>
La necesidad de contar con bases de datos distribuidas en la arquitectura nativa de la nube	219
Opciones de bases de datos y almacenamiento de Azure	220
Introducción a Vitess: MySQL distribuida y particionada	221
¿Por qué ejecutar Vitess?	221
La arquitectura de Vitess	222
Implementación de Vitess en Kubernetes	223
Introducción a Rook: Orquestador de almacenamiento para Kubernetes	224
La arquitectura de Rook	224
Implementación de Rook en Kubernetes	225
Introducción a TiKV	226
¿Por qué usar TiKV?	226
La arquitectura de TiKV	226
Implementación de TiKV en Kubernetes	228
Más información sobre etcd	229
Plataforma de hardware	230
Escalado automático y corrección automática	230
Disponibilidad y seguridad	231
Resumen	231
<b>10. Recibir el mensaje.....</b>	<b>233</b>
La necesidad de contar con mensajes	233
Ejemplo de caso de uso de mensajería: Ingesta y análisis de registros	235
Generación 1: Sin colas	235
Generación 2: Con colas de la nube y almacenamiento de objetos	236
Generación 3: Con cola Pub/Sub basada en la memoria	237
Conceptos básicos de las plataformas de mensajería	238
La mensajería en comparación con el streaming	238
Fundamentos de la mensajería	238

Productores y consumidores	239
Agentes y clústers	240
Durabilidad y persistencia	241
Entrega de mensajes	241
Seguridad	242
Patrones de mensajería comunes	242
Cola simple	242
Publicar y Suscribir	242
Cola duradera	242
Descripción general de las plataformas populares de mensajería nativa de la nube	243
RabbitMQ	243
Apache Kafka	243
CNCF CloudEvents	244
Análisis de la mensajería en la nube con NATS	244
Arquitectura del protocolo NATS	244
Persistencia de NATS con JetStream	249
Seguridad de NATS	249
Implementación de NATS en Kubernetes	251
Servicios de mensajería de Azure	253
Azure Service Bus	253
Azure Event Hubs	258
Azure Event Grid	261
Resumen	263
<b>11. Informática sin servidor.....</b>	<b>265</b>
Introducción a la informática sin servidor	265
¿Qué es la informática sin servidor?	265
¿Qué es una función sin servidor?	266
El entorno sin servidor	266
Beneficios de la informática sin servidor	267
Posibles desventajas de la informática sin servidor	268
Azure Function Apps	268
Arquitectura de la aplicación de funciones	269
Creación de una aplicación de funciones	270
Knative	272
Arquitectura de Knative	272
Instalación y ejecución de Knative Serving en Kubernetes	272
Instalación y ejecución de Knative Eventing en Kubernetes	274
KEDA	276
Arquitectura de KEDA	276
Instalación de KEDA en Kubernetes	277

OpenFaaS	281
Arquitectura de OpenFaaS	281
Instalación de OpenFaaS	281
Cómo escribir su primera función de OpenFaaS	282
Resumen	283
<b>12. Conclusión.....</b>	<b>285</b>
¿Qué sigue?	287
<b>Índice.....</b>	<b>289</b>



---

# Prólogo

La informática en la nube se ha adoptado ampliamente como un modelo de transformación empresarial digital de vanguardia que impulsa el crecimiento y la innovación. Hoy en día, los clientes buscan un ecosistema y una experiencia que sea rápida y que se integre a la perfección con sus servicios existentes. Desde una perspectiva empresarial, la nube ofrece servicios para los consumidores y las empresas de una manera escalable, altamente confiable y disponible. Desde la perspectiva del usuario final, la nube ofrece un modelo simple para adquirir servicios informáticos sin necesidad de comprender completamente la infraestructura y la tecnología subyacentes.

Para aprovechar al máximo la velocidad y la agilidad de los servicios en la nube, muchas de las aplicaciones existentes se han transformado en aplicaciones nativas de la nube, y se están creando nuevas soluciones para la nube. Las aplicaciones nativas de la nube se crean desde cero para adoptar un cambio rápido, gran escala y resistencia. De forma predeterminada, la infraestructura subyacente para las aplicaciones nativas de la nube desempeña un papel fundamental para atender de forma eficaz las necesidades de una empresa. Si la infraestructura subyacente no está diseñada con los procedimientos correctos, incluso las mejores aplicaciones nativas de la nube fallarán en los entornos de producción.

Este libro explora cómo las infraestructuras nativas de la nube modernas en Azure se pueden crear y administrar en un entorno de producción, junto con diversos requisitos y consideraciones de diseño de las aplicaciones nativas de la nube.

## Quiénes deben leer este libro

Este libro brinda una introducción sencilla pero completa al panorama nativo de la nube y a todas las principales tecnologías que los ingenieros usan para crear tales entornos confiables. El libro es para ingenieros de confiabilidad del sitio, ingenieros de DevOps, arquitectos de soluciones, entusiastas de Azure y cualquier persona que esté involucrada en la creación, migración, implementación y administración de las operaciones diarias de una carga de trabajo nativa de la nube.

El libro supone que usted cuenta con conocimientos básicos de la cultura de la nube y DevOps en general. Pero incluso si no es así y desea comprender mejor la conmoción en torno a la nube nativa y otras tecnologías sofisticadas, este libro sigue siendo un lugar adecuado para comenzar.

## Objetivos de este libro

Cuando termine este libro, podrá seguir y crear su propia infraestructura en Microsoft Azure. Presentamos una introducción secuencial a los componentes principales del mundo nativo de la nube y cómo usarlos, implementarlos y mantenerlos en Azure. Además, aprenderá sobre la necesidad de contar con tecnologías nativas de la nube en este mundo actual más valiente, los problemas que resuelven y los procedimientos prácticos recomendados.

En este libro:

- Aprenderá cómo crear infraestructura nativa de la nube en Azure siguiendo la ruta del panorama nativo de la nube de la fundación Cloud Native Computing Foundation (CNCF)
- Determinará qué tecnologías utilizar en diferentes etapas del diseño
- Descubrirá cómo abordar los problemas que puede enfrentar al administrar y operar la infraestructura nativa de la nube, así como las tecnologías que ayudan a resolverlos

## Cómo navegar por este libro

Este libro se organiza de la siguiente manera:

- El [capítulo 1](#) proporciona una introducción básica a la nube y la necesidad de contar con tecnología nativa de la nube y sus adaptaciones.
- El [capítulo 2](#) abarca los fundamentos de la infraestructura como código (IaC) con Terraform y Packer, y presenta a Azure y Ansible como aprovisionadores/administradores de configuración.
- El [capítulo 3](#) presenta los contenedores y los tiempos de ejecución del contenedor, como containerd, Docker y CRI-O. También se analizan diferentes tipos de registros de contenedores.
- El [capítulo 4](#) analiza Kubernetes e incluye los detalles necesarios para usar la infraestructura en los próximos capítulos.
- El [capítulo 5](#) trata específicamente de Azure Kubernetes Service y del administrador de paquetes Helm.
- El [capítulo 6](#) se centra en cómo se puede hacer observable la infraestructura nativa de la nube moderna.
- El [capítulo 7](#) se trata de la detección de servicios y la malla de servicio. En este capítulo, presentamos el servidor DNS CoreDNS y la malla de servicio de Istio.

- El [capítulo 8](#) se ocupa de la administración de redes y políticas, incluidas las interfaces de redes de contenedores Calico, Flannel, Cilium, Azure Policy y Open Policy Agent.
- El [capítulo 9](#) explica cómo se implementan los sistemas de almacenamiento persistentes en la infraestructura nativa de la nube, con un enfoque en Azure Storage, Vitess, Rook, TiKV y etcd.
- El [capítulo 10](#) se centra principalmente en las plataformas de mensajería y transmisión, como NATS y los servicios de mensajería de Azure.
- El [capítulo 11](#) es una simple introducción a la perspectiva sin servidor de las cosas en el panorama nativo de la nube.
- El [capítulo 12](#) actúa como un resumen de todo lo que analizamos en los capítulos anteriores.

## Convenciones utilizadas en este libro

En este libro, se usan las siguientes convenciones tipográficas:

### *Cursiva*

Indica términos nuevos, direcciones URL, direcciones de correo electrónico, nombres de archivo y extensiones de archivos.

### Espaciado fijo

Se utiliza para las listas de programas, así como dentro de los párrafos para referirse a elementos del programa, como nombres de variables o funciones, bases de datos, tipos de datos, variables del entorno, declaraciones y palabras clave.

### **Negrita con espacio fijo**

Muestra comandos u otro texto que el usuario debe escribir de forma literal.



Este elemento indica un consejo o una sugerencia.



Este elemento indica una nota general.



Este elemento indica una advertencia o precaución.

## Uso de ejemplos de código

El material complementario (ejemplos de código, ejercicios, etc.) está disponible para su descarga en [https://github.com/stormic-nomad-nishant/cloud\\_native\\_azure](https://github.com/stormic-nomad-nishant/cloud_native_azure).

Si tiene alguna pregunta técnica o un problema con los ejemplos de código, envíe un correo electrónico a [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

Este libro está aquí para ayudarlo a hacer su trabajo. En general, si el ejemplo de código se incluye en este libro, puede usarlo en sus programas y documentación. No necesita ponerse en contacto con nosotros para solicitar autorización, a menos que esté reproduciendo un porcentaje significativo del código. Por ejemplo, escribir un programa que use varios fragmentos de código de este libro no requiere permiso. Para vender o distribuir ejemplos de libros de O'Reilly, necesita permiso. Responder una pregunta citando este libro y citando el ejemplo de código no requiere permiso. Para incorporar una cantidad significativa del código de ejemplo de este libro en la documentación de su producto, necesita permiso.

Apreciamos, pero generalmente no requerimos, la atribución. Por lo general, una atribución incluye el título, el autor, la editorial y el ISBN. Por ejemplo: "*Infraestructura nativa de la nube con Azure* de Nishant Singh y Michael Kehoe (O'Reilly). Copyright 2022 Nishant Singh y Michael Kehoe, 978-1-492-09096-0".

Si considera que el uso de ejemplos de código no se encuentra dentro de los usos legítimos o en el permiso anteriormente indicado, no dude en ponerse en contacto con nosotros en [permissions@oreilly.com](mailto:permissions@oreilly.com).

## Aprendizaje en línea de O'Reilly



Durante más de 40 años, *O'Reilly Media* ha proporcionado capacitación tecnológica y empresarial, conocimiento e información para ayudar a las empresas a tener éxito.

Nuestra red única de expertos e innovadores comparte sus conocimientos y experiencia a través de libros, artículos y nuestra plataforma de aprendizaje en línea. La plataforma de aprendizaje en línea de O'Reilly le ofrece acceso a petición a cursos de capacitación en vivo, rutas de aprendizaje profundo, entornos de codificación interactivos y una amplia colección de texto y video de O'Reilly y más de 200 otras editoriales. Para obtener más información, visite <http://oreilly.com>.

# Cómo ponerse en contacto con nosotros

Envíe los comentarios y las preguntas sobre este libro a la editorial:

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472  
800-998-9938 (en Estados Unidos o Canadá)  
707-829-0515 (internacional o local)  
707-829-0104 (fax)

Tenemos una página web para este libro, donde enumeramos las erratas, los ejemplos y cualquier información adicional. Puede acceder a esta página en <https://oreil.ly/cloud-native-azure>.

Envíe un correo electrónico a [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com) para realizar comentarios o preguntas técnicas sobre este libro.

Para obtener noticias e información sobre nuestros libros y cursos, visite <http://oreilly.com>.

Síganos en Facebook: <http://facebook.com/oreilly>.

Síganos en Twitter: <http://twitter.com/oreillymedia>.

Vea nuestros videos en YouTube: <http://youtube.com/oreillymedia>.

## Agradecimientos

Nos gustaría agradecer a Rita Fernando, Nicole Taché y Jennifer Pollock, nuestros editores en O'Reilly, que nos brindaron su increíble apoyo y nos guiaron a través del proceso de escritura. También nos gustaría agradecer a nuestros revisores tecnológicos por sus valiosas contribuciones al libro. Este libro no habría sido posible sin las excelentes revisiones y comentarios exhaustivos de Matt Franz, Peter Jausovec, Liudmila Sinkevich, Steve Machacz y Alexander Kabonov.

Además, nos gustaría destacar a Matt Franz por sus rigurosas revisiones, sugerencias y contribuciones para mejorar la calidad del libro. Matt nos ofreció comentarios importantes sobre los capítulos y contribuyó en el [capítulo 10](#).

Nishant desea agradecer a su madre, su padre y a su esposa, Mahek, por ser tan pacientes y comprensivos mientras trabajaba en el libro. También le gustaría agradecer al equipo de O'Reilly y a sus colegas de LinkedIn que apoyaron la idea desde el principio.

A Michael le gustaría agradecer a su familia, que lo animó a seguir sus sueños y nunca rendirse. También le gustaría agradecer a su mejor amiga, Jennifer, por su apoyo y paciencia mientras trabajaba en este libro, al equipo de O'Reilly por ayudar a llevar el libro de la idea a la realidad, y a todos los revisores por su tiempo.



## CAPÍTULO 1

# Introducción: ¿Por qué la nube nativa?

Aunque usar la nube es una forma de resolver problemas con la escalabilidad, disponibilidad y confiabilidad de nuestras aplicaciones, no es una solución milagrosa. No podemos poner nuestras aplicaciones en la nube y esperar que estén en funcionamiento para siempre, ni tampoco podemos empaquetar nuestras aplicaciones en contenedores para convertirlas en microservicios que puedan ejecutarse sin problemas en la nube. Para aprovechar al máximo lo que la nube ofrece, tenemos que desarrollar infraestructuras y servicios pensando en la nube.

Para comprender realmente el viaje nativo de la nube y su importancia, primero tenemos que observar cómo se veía el mundo de la infraestructura y los servicios durante los primeros días de Internet. Aventurémonos en este viaje.

## El viaje a la nube

En los primeros días de Internet, la infraestructura general de aplicaciones web se hospedaba con servidores físicos que debían aprovisionarse y prepararse antes de que pudieran proporcionar aplicaciones. Los equipos de TI tenían que comprar físicamente los servidores y configurarlos en el entorno local, instalar los sistemas operativos pertinentes para el servidor, preparar los entornos y luego implementar aplicaciones en ellos. Hubo muchos problemas con este enfoque. Por ejemplo, había servidores que no se usaban bien (ya que nunca podría usarlos por completo), era difícil ejecutar varias aplicaciones, y los costos de configuración y mantenimiento eran altos. La *virtualización* se desarrolló para permitir un uso más eficiente de los servidores físicos. La virtualización crea una capa de abstracción sobre el hardware físico que permite que los recursos subyacentes, como los procesadores, la memoria y el almacenamiento, se dividan y compartan.

La virtualización resolvió muchos problemas del uso de recursos y los multiinquilinos, pero todavía necesitaba poseer el hardware para implementar la aplicación y todavía tenía que

mantener toda la sobrecarga de la ejecución de su centro de datos. Esto dio lugar a la necesidad de ejecutar la infraestructura como servicio (IaaS), donde los servidores son propiedad de terceros que son responsables de la infraestructura subyacente de las aplicaciones. Este fue el comienzo de la era de la informática en la nube y permitió que las empresas se centraran en sus aplicaciones y entornos subyacentes sin preocuparse por problemas de hardware, sobrecarga o configuración. A IaaS le siguió la plataforma como servicio (PaaS), que se centró en reducir el esfuerzo aún más al separar el entorno de software subyacente y el tiempo de ejecución. Esto significaba que los desarrolladores solo tenían que centrarse en escribir sus aplicaciones y definir las dependencias y la plataforma de servicio sería completamente responsable de hospedar, ejecutar, administrar y exponer las aplicaciones. PaaS lideró el camino hacia servicios en la nube completamente administrados con la llegada del software como servicio (SaaS), conocido popularmente como "software a petición", que proporciona a los consumidores la aplicación como un servicio que se paga por uso.

A medida que la informática en la nube adquirió popularidad, también se generalizó la idea de contar con tecnologías nativas de la nube que usaran la nube de manera más eficaz, a la vez que aprovecharan todo el potencial de la infraestructura en la nube y sus diversas ofertas. Esto dio lugar al desarrollo de la infraestructura nativa de la nube y el desarrollo de aplicaciones nativas de la nube. La infraestructura nativa de la nube crea una abstracción en la infraestructura subyacente del proveedor de nube y expone la infraestructura con API. Esta filosofía de administración de infraestructura permite que sea muy fácil escalar y reducir la complejidad subyacente, lo que mejora indirectamente la disponibilidad, la resistencia y la capacidad de mantenimiento. Del mismo modo, las aplicaciones nativas de la nube fortalecen el puente entre la aplicación y la infraestructura al incorporar características de soporte, como comprobaciones de estado, telemetría y métricas, resistencia, un entorno de microservicios y recuperación automática.

Ahora demos un vistazo a los desafíos en el entorno de la informática en la nube.

## Desafíos en la nube

Los proveedores de la nube pública se han convertido en una solución empresarial dominante para una gran cantidad de necesidades del sector y requisitos empresariales en constante aumento. Le ofrecen ventajas, como una mayor disponibilidad y escalabilidad, junto con la flexibilidad de diseñar sus aplicaciones de una manera que utiliza los servicios en la nube. Cuando se introdujeron las soluciones en la nube por primera vez, los clientes potenciales se preocuparon por los diversos desafíos, como la seguridad, la administración eficaz de costos, el cumplimiento y el rendimiento. Esos primeros desafíos ahora son cosa del pasado para la mayoría de los consumidores de la nube, ya que se han superado con los avances en las tecnologías de los proveedores de nube y la forma en que las empresas implementan soluciones en la nube.

Aunque hemos logrado muchos avances, esto no significa que la nube sea perfecta. Todavía hay desafíos en el panorama de la nube, pero se ven muy diferentes de los que enfrentamos cuando la nube era nueva. Los clientes actuales tienen que considerar los siguientes desafíos:

### *Demasiadas opciones*

Existen muchos proveedores de la nube con una amplia gama de servicios para elegir. Esto significa que necesita contratar a arquitectos y equipos de ingeniería expertos que sepan cómo operar los servicios y utilizarlos de acuerdo con su caso de uso comercial. No solo es difícil contratar a estos ingenieros, sino que encontrar ingenieros que sean expertos en un campo específico requiere una inversión de tiempo importante.

### *Rápido crecimiento y desarrollo de servicios y tecnologías en la nube*

Los proveedores más importantes de la nube, como Amazon, Microsoft y Google, están lanzando un gran número de nuevos servicios en la nube. Esto se traduce en una mayor necesidad de capacitar a los ingenieros en estos nuevos servicios y una mayor necesidad de mantener estos servicios en el futuro a medida que crecen las aplicaciones. A menudo, el nivel de inversión en estos servicios también provoca indirectamente el bloqueo del proveedor, lo que genera un espiral de restricciones futuras en el diseño de aplicaciones.

### *Varias generaciones de tecnología*

A medida que migramos a la era de la nube, también hemos realizado una migración de lift and shift de nuestras pilas de aplicaciones de varias generaciones de soluciones de infraestructura, que van desde máquinas virtuales hasta contenedores y tecnologías sin servidor. Esta migración de aplicaciones requiere una gran cantidad de esfuerzo para entender las tecnologías subyacentes y admitirlas en el futuro.

### *Mayor complejidad operativa*

Estas tecnologías de rápido crecimiento, combinadas con la migración acelerada de cargas de trabajo a la nube, han dado lugar a una mayor complejidad operativa y a una lista en constante crecimiento de factores que se deben buscar, incluidos sistemas de almacenamiento, modelos de seguridad, modelos de gobernanza y plataformas de administración, entre otros.

### *Evolución de las necesidades y tecnologías empresariales*

Las nuevas áreas de la tecnología y el cambio cultural también han desarrollado rápidamente la arquitectura empresarial. Por ejemplo, con la llegada de la cultura de DevOps, ahora se puede implementar en un par de minutos una nueva aplicación que antes tardaba semanas o meses en desarrollarse. Otras áreas más avanzadas, como la ciencia de datos y machine learning, también han entrado en el panorama, lo que aumenta las necesidades empresariales y la madurez general de la ingeniería.

Por lo tanto, a pesar del poder de la informática en la nube, las empresas tuvieron que sortear diversas complejidades. Finalmente, se hizo evidente que las empresas querían tener la velocidad, la escala y el margen de la nube, pero no la sobrecarga. Para lograrlo, era necesario adoptar el enfoque nativo de la nube para la creación de aplicaciones, lo que ayudaría a las empresas a aprovechar al máximo la nube.

# Cloud Native Computing Foundation

A medida que más empresas adoptan tecnologías nativas de la nube, crean software interno y se asocian estrechamente con otras empresas para llevar productos al mercado rápidamente, se están realizando avances para mejorar el dominio nativo de la nube. Una organización que está ayudando a liderar el camino en este esfuerzo es la [Cloud Native Computing Foundation \(CNCF\)](#). Su misión es la siguiente:

Las tecnologías nativas de la nube permiten a las organizaciones crear y ejecutar aplicaciones escalables en entornos modernos y dinámicos, como nubes públicas, privadas e híbridas. Los contenedores, las mallas de servicio, los microservicios, la infraestructura inmutable y las API declarativas ejemplifican este enfoque.

Estas técnicas permiten contar con sistemas de acoplamiento flexible que son resilientes, administrables y observables. En combinación con una automatización sólida, les permiten a los ingenieros realizar cambios de alto impacto con frecuencia y de forma previsible con un mínimo de esfuerzo.

Cloud Native Computing Foundation busca impulsar la adopción de este paradigma al fomentar y mantener un ecosistema de proyectos de open source y neutrales para los proveedores. Democratizamos los patrones de vanguardia para que estas innovaciones sean accesibles para todos.

La CNCF ha compilado un [panorama nativo de la nube](#) interactivo que muestra la extensión total de las soluciones nativas de la nube actuales. El panorama de CNCF actúa como un mapa para las tecnologías nativas de la nube y ofrece directrices para crear aplicaciones nativas de la nube exitosas. Aunque el panorama nativo de la nube proporciona una gran cantidad de información sobre cómo crear servicios, puede ser un desafío (especialmente para los principiantes) lograrlo, ya que incluye una serie de servicios que se usan en conjunto. En este libro, elegimos lo mejor de la tecnología disponible y seguimos las directrices de la CNCF para navegar mejor en el mundo nativo de la nube.

## Adopción de una infraestructura nativa de la nube con Azure

Ahora que entiende los orígenes de la informática en la nube, los desafíos de operar en la nube y cómo las ofertas nativas de la nube pretendían cambiar la forma en que se estaban desarrollando y entregando los servicios, analicemos cómo este libro puede ayudarlo a avanzar.

Si observa el movimiento nativo de la nube y comienza a trazar su historia desde el principio, la primera tecnología que surgió fue Kubernetes, y encontrará muchas personas que la han adoptado. En la superficie, es posible que observe que muchas empresas tienen Kubernetes en medio de su pila. Esto da la impresión de que Kubernetes es una herramienta esencial que puede resolver todos sus problemas y mágicamente lograr que su entorno se repare automáticamente y sea tolerante a errores. Este malentendido ha llevado a mucha gente a considerar Kubernetes como solución mágica sin entender completamente el significado más profundo. Es importante entender la necesidad de tener estas soluciones y obtener información sobre todo el ecosistema mientras observa el panorama más amplio.

En los capítulos que siguen, proporcionamos orientación sobre cómo crear entornos nativos de la nube siguiendo las directrices que sugiere la CNCF. Escribimos este libro como un manual para ingenieros y entusiastas que acaban de comenzar con las transformaciones nativas de la nube y desean explorar la necesidad general de contar con arquitecturas nativas de la nube. Aprenderá formas prácticas de crear infraestructuras nativas de la nube en Microsoft Azure navegando por el panorama de la CNCF. También aprenderá los principios de la infraestructura nativa de la nube desde la perspectiva de un principiante e implementará soluciones maduras en Azure.<sup>1</sup> Como Azure es uno de los actores clave en el ecosistema de la nube pública, proporciona una comprensión madura de la pila de infraestructura. Lo guiaremos por los conceptos básicos de la infraestructura de la nube y explicaremos cómo ser nativos de la nube a través de diversas tecnologías, a la vez que destacamos por qué estas tecnologías son necesarias. Le daremos una amplia descripción de las tecnologías nativas de la nube y los servicios de Azure a través de una aplicación práctica, y le mostraremos las ventajas de basarse en la nube (con Azure) en comparación con la nube nativa. Nuestro objetivo final es brindarle un libro que muestre todas las principales tecnologías nativas de la nube y su importancia para que pueda entender el razonamiento lógico detrás de las ventajas del uso de estas tecnologías.

## Resumen

En este capítulo introductorio, sentamos las bases de qué es la informática en la nube y cómo las tecnologías nativas de la nube ayudan a mejorar la adopción de la nube. Aprendió cómo la nube se hizo popular y cómo evolucionó la informática en la nube desde el hardware físico hasta un entorno sin servidor. También aprendió sobre los desafíos que presenta la informática en la nube y la creciente necesidad de adaptarse a las tecnologías nativas de la nube. Explicamos qué significa la nube nativa y cómo el resto del libro seguirá la ruta hacia la nube nativa en Azure. Esperamos que el viaje que emprenderemos le parezca interesante y que este esfuerzo lo ayude a adaptar las tecnologías nativas de la nube de forma más eficaz.

---

<sup>1</sup> Azure también ofrece muchas directrices con respecto a la arquitectura que muestran los procedimientos recomendados para crear servicios en Azure: consulte <https://docs.microsoft.com/azure/architecture> y <https://azure.microsoft.com/blog/azure-application-architecture-guide/>.



## CAPÍTULO 2

# Infraestructura como código: Configuración del Gateway

La nube, con su rica variedad de soluciones que ofrecen todo tipo de características, desde crear redes sin esfuerzo hasta informática "a escala mundial", se está convirtiendo en el hogar de facto para la mayoría de las organizaciones, independientemente de su tamaño. El ecosistema de la mayoría de los proveedores de nube ahora es más grande que nunca e incluye máquinas virtuales simples, complicados clústeres administrados e incluso una infraestructura altamente sofisticada. Microsoft Azure, por ejemplo, ofrece una variedad de servicios diseñados para los usuarios finales. Estas soluciones se basan en la infraestructura que es resiliente por naturaleza y tienen acuerdos de nivel de servicio (SLA)<sup>1</sup> bien definidos para satisfacer las necesidades de todos los clientes, desde pequeñas empresas emergentes hasta grandes empresas.

Si bien la naturaleza dinámica de la informática en la nube es una ventaja para las organizaciones, puede ser un desafío mantener los servicios en entornos de nube a medida que su organización crece. A medida que aprovecha más y más servicios basados en la nube para respaldar sus necesidades empresariales cada vez más grandes, rápidamente se da cuenta de que ya no puede mantener sus aplicaciones y la infraestructura subyacente simplemente haciendo clic en la consola web cada vez que desee llevar a cabo una acción nueva. El diseño evolutivo de la nube se creó en base a generaciones anteriores de hardware e infraestructura virtualizada con un plano de control programable que proporciona flexibilidad en forma de capas de API. Estas API se aprovechan mediante las herramientas de infraestructura como código (IaC) para permitir que los profesionales de la nube mantengan fácilmente sus entornos.

---

<sup>1</sup> Un SLA es un acuerdo entre el proveedor de la nube y el cliente (usted) sobre las métricas mensurables, como el tiempo de actividad, la capacidad de respuesta y las responsabilidades.

Cuando desea crear una infraestructura a nivel de producción desde cero, debe comenzar a tratar sus definiciones de infraestructura de la misma manera en que trata su código. Aprovisionar y escalar su infraestructura de forma manual no solo es engoroso y lento. También es un proceso propenso a errores: no puede crear la misma infraestructura dos veces con el mismo nivel de confianza. Cuando trata su infraestructura como código, también hereda los principios de ingeniería de software, como el control de versiones, la capacidad de prueba, la automatización y la velocidad de desarrollo, como parte de la estrategia de DevOps. IaC garantiza que su infraestructura se pueda crear de manera coherente y repetible. A medida que más equipos de desarrollo de software han adoptado metodologías ágiles, ahora están obligados a migrar características y soluciones con más frecuencia y rapidez a la nube. Como tal, la infraestructura se ha conectado estrechamente con las aplicaciones, y los equipos tienen que administrar las aplicaciones y la infraestructura a través de un proceso fluido. Además, los equipos necesitan iterar e implementar en repetidas ocasiones nuevas características y la infraestructura subyacente con más frecuencia. Una forma rápida de implementar IaC es usar las plantillas escritas previamente que ofrecen los proveedores de nube, que dan instrucciones sobre cómo crear una infraestructura. Azure admite de forma nativa las plantillas de Azure Resource Manager (ARM), que se usan para definir y crear recursos de infraestructura que arrojan luz directamente sobre la filosofía de IaC. Esta combinación de la nube y la IaC permite a las organizaciones ingresar cambios de forma más rápida y eficaz.

Este capítulo actúa como base para comprender de forma sólida IaC y Azure. En primer lugar, presentaremos el concepto de IaC y su importancia en la creación de infraestructura nativa de la nube. A continuación, presentaremos Microsoft Azure, que es el proveedor de nube que utilizaremos en todo el libro.

A continuación, profundizaremos en Terraform como una herramienta para implementar IaC. También trataremos Packer, que es principalmente una herramienta para crear imágenes de máquina, y Ansible, para presentarle la administración de configuración mientras crea aplicaciones nativas de la nube. Por último, nos referiremos a Azure DevOps antes de cerrar el capítulo. Manos a la obra.

## Infraestructura como código y su importancia en el mundo nativo de la nube

Antiguamente, cuando un equipo de TI quería obtener nuevo hardware para la implementación de aplicaciones, primero tenía que iniciar una solicitud de aprovisionamiento de hardware, que tardaba días o incluso semanas en completarse. Incluso después de que el hardware finalmente se entregaba localmente, tardaba unos días en llegar a un estado activo antes de que se pudieran implementar las aplicaciones en él. Hoy en día, con el crecimiento de DevOps y la nube, estos procesos lentos han desaparecido. Los ciclos de lanzamiento se han acortado y la velocidad con la que se pueden entregar los requisitos empresariales ha aumentado.

Sin embargo, incluso con la introducción de la nube, administrar la infraestructura sigue siendo un problema. Aunque los servidores físicos fueron reemplazados con servidores virtuales y la administración de infraestructura compleja se convirtió en responsabilidad del proveedor de la nube, aumentar y escalar la infraestructura seguían siendo tareas complejas

y propensas a errores. Los equipos de ingeniería necesitaban crear constantemente nuevas infraestructuras y pilas de software, y seguir manteniendo la infraestructura antigua. Para aliviar el tedio y la fatiga asociados con estas tareas, se automatizó la creación y administración de sistemas basados en la nube, lo que generó un entorno sólido de infraestructura y aplicaciones.

Finalmente, a medida que el código se adaptó para aprovisionar las capas de informática, redes y almacenamiento de la pila, se hicieron evidentes los beneficios de tratar la infraestructura como código. Esto condujo al movimiento de adaptación de IaC en entornos de nube.

La *infraestructura como código* es la filosofía que consiste en tratar su infraestructura como instrucciones programables. Tal como se protegen los archivos de control de origen, se protegen las definiciones de infraestructura. Tratar la infraestructura como código ofrece numerosas ventajas, entre las que se incluyen las siguientes:

- La capacidad de proteger una versión de las definiciones de su infraestructura y revertir a una versión específica en caso de que haya un problema
- La idempotencia, que le permite reproducir exactamente la misma infraestructura con el mínimo esfuerzo
- El registro estandarizado, la supervisión y la solución de problemas en toda la pila
- Reducir puntos únicos de error en sus equipos de ingeniería (es decir, donde solo un ingeniero sabe cómo crear un tipo específico de sistema)
- Mayor productividad de los desarrolladores (los ingenieros de confiabilidad del sitio [SRE] y/o los ingenieros de DevOps pueden centrarse en escribir código que los desarrolladores pueden usar para crear un entorno de prueba)
- Interacción manual mínima, que genera una infraestructura menos propensa a errores
- Tiempo promedio de recuperación (MTTR) en caso de errores drásticamente reducido
- La capacidad de probar su infraestructura incluso antes de que se cree

Con los beneficios de IaC, es evidente que tales herramientas desempeñan un papel importante en los entornos nativos de la nube moderna, donde las aplicaciones deben cambiarse sobre la marcha. Los cambios pueden ser menores, como agregar una nueva etiqueta de configuración, o significativos, como agregar nuevos clústeres para mantenerse al día con los requisitos de capacidad. Una estrategia de IaC confiable ayuda a los desarrolladores a administrar estos cambios fácilmente sin comprometer la velocidad del proceso de desarrollo de software. Una vez que IaC se integra en el núcleo del proceso de compilación de la infraestructura, también ayuda a crear una infraestructura antifragilidad.



### Infraestructura antifragilidad

La infraestructura antifragilidad son sistemas que pueden soportar el estrés y mostrar elasticidad junto con resiliencia. La antifragilidad tiene como objetivo crear una infraestructura que pueda manejar eventos impredecibles y anormales mientras se fortalece.

Las aplicaciones nativas de la nube actuales se consideran aplicaciones en movimiento, ya que la infraestructura subyacente también sigue cambiando y actualizándose. La nube le permite tratar a sus infraestructuras como entidades de corta duración en lugar de entidades permanentes. El enfoque nativo de la nube promueve crear infraestructura que sea reemplazable en lugar de corregir la infraestructura que no funciona. Dado que el enfoque nativo de la nube desacopla las aplicaciones de su infraestructura, les ofrece a los ingenieros el control y la confianza para hacer cambios en la infraestructura, sabiendo que pueden revertir y avanzar fácilmente.



### Filosofía de la infraestructura nativa de la nube

Mientras cree la infraestructura nativa de la nube, siempre trate la infraestructura subyacente como entidades de corta duración en lugar de sistemas de larga vida que pueden mantenerse. Esto significa que si un servidor falla, será fácil eliminarlo y crear un nuevo servidor al instante, en lugar de intentar corregirlo.

El cambio de la arquitectura monolítica a los microservicios también ayudó a que IaC ganara impulso al garantizar que las aplicaciones modernas pudieran seguir la **metodología de "aplicación de doce factores"**, lo que permite que las decisiones sobre la infraestructura se realicen independientemente del desarrollo de aplicaciones. Otra ventaja importante de IaC en la creación de la infraestructura nativa de la nube es que permite la infraestructura inmutable; es decir, una vez que se implementa la infraestructura, no se puede cambiar ni configurar. En un escenario típico de implementación de aplicaciones mutable, una aplicación se desarrolla, se implementa en la infraestructura subyacente y, finalmente, se configura en la infraestructura subyacente ([Figura 2-1](#)). Este proceso presenta la *alteración de la configuración*, que da lugar a la infraestructura que comienza a alejarse del estado inicial. Por otro lado, la infraestructura inmutable se centra en implementar de nuevo la infraestructura y las aplicaciones cada vez que se inserta un nuevo cambio, lo que mantiene firmemente la infraestructura en su estado exacto. La inmutabilidad se logra al desarrollar y configurar la aplicación en un estado preparado previamente (por ejemplo, una imagen de máquina) para luego implementarla ([Figura 2-2](#)).



Figura 2-1. El flujo mutable



Figura 2-2. El flujo inmutable

Estas son las principales ventajas de crear infraestructura inmutable:

- Infraestructura/sistemas predecibles en lugar de infraestructura/sistemas reactivos
- Implementaciones que ahora son de naturaleza atómica
- Más control sobre su infraestructura, lo que resulta en una mejor telemetría

Ahora que analizamos la IaC y su valor al crear la infraestructura nativa de la nube, veamos cómo puede usar Microsoft Azure en la infraestructura nativa de la nube.

## Introducción a Azure y configuración del entorno

Microsoft Azure es un proveedor de nube con muchas regiones en todo el mundo. Azure le permite crear e implementar rápidamente infraestructura a través de medios manuales y automatizados, como analizaremos a lo largo de este libro. Azure como PaaS (plataforma como servicio) ofrece una gran flexibilidad a la comunidad de desarrolladores para innovar, crear e implementar aplicaciones. Elegimos Azure como nuestro entorno de nube debido a su velocidad, flexibilidad, seguridad, recuperación ante desastres y curva de aprendizaje simple.

En esta sección, explicaremos los conceptos básicos en torno a Azure y cómo crear una cuenta de Azure.

## Fundamentos de Azure y preparación del entorno de Azure

Antes de comenzar con Azure, es importante entender algunos de sus **conceptos básicos**. Comprender estos conceptos le ayudará a tomar decisiones fundamentadas sobre las políticas en el futuro:

### Inquilino

Un inquilino de Azure es una representación de Azure Active Directory (AAD) de una organización. Probablemente ya esté familiarizado con los servicios tradicionales de Active Directory (AD). Cuando una organización crea una cuenta con Microsoft, se crea una instancia de AAD.

Todas las instancias de AAD están completamente separadas entre sí, al igual que todas las identidades (cuentas de usuarios o de servicios). Si tiene un inquilino de Microsoft existente (por ejemplo, para Office 365), puede usarlo para su huella de Azure. También puede crear su propio inquilino de AAD a través del [portal de Azure](#). Recuerde **habilitar la autenticación multifactor** (MFA)<sup>2</sup> en su inquilino.

---

<sup>2</sup> La MFA agrega una capa de protección al proceso de inicio de sesión. Al acceder a cuentas o aplicaciones, los usuarios proporcionan una verificación de identidad adicional, como escanear una huella dactilar o ingresar un código que reciben en su teléfono.

## *Suscripciones*

Las suscripciones de Azure son básicamente un contenedor de facturación para sus recursos. También puede establecer directivas específicas de Azure en el nivel de suscripción.

## *Grupos de administración*

Los grupos de administración de Azure ofrecen un medio para organizar y administrar suscripciones. Puede utilizar grupos de administración para aplicar políticas/gobernanza a todos los recursos (incluidas las suscripciones) dentro del grupo de administración. De forma predeterminada, todas las suscripciones dentro del grupo de administración heredarán la política del grupo de administración.

## *Grupos de recursos*

Un grupo de recursos es un contenedor que contiene recursos relacionados para una solución de Azure. El grupo de recursos incluye los recursos que desea administrar como un grupo. Usted decide qué recursos pertenecen a un grupo de recursos según lo que tenga más sentido para su organización. Además, los recursos pueden ubicarse en diferentes regiones desde el grupo de recursos.

## *Regiones*

Una región de Azure contiene dos o más zonas de disponibilidad. Una zona de disponibilidad es un centro de datos físicamente aislado en una región geográfica.

En la **Figura 2-3** se muestra cómo se asocian entre sí estos conceptos básicos.

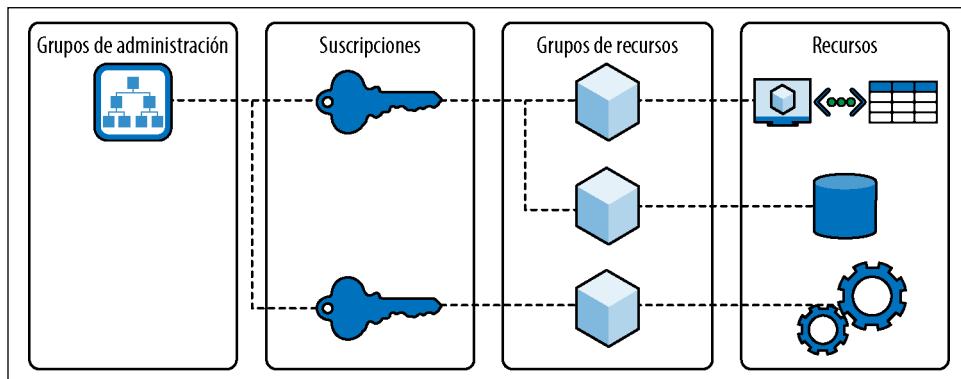


Figura 2-3. Conceptos básicos de Azure

## **Creación de una cuenta de Azure**

Para crear una cuenta de Azure, ingrese a <https://azure.microsoft.com> y use el plan gratuito. Tendrá que iniciar sesión con una cuenta de Microsoft. También puede usar una cuenta de Hotmail, Outlook u Office 365, pero recomendamos encarecidamente usar MFA también. Siga las instrucciones en el sitio web de Azure para crear su cuenta.

Para continuar, haga clic en el vínculo Mi cuenta en la esquina superior derecha o acceda directamente al [portal de Microsoft Azure](#).

Ahora hablaremos sobre cómo usar la CLI de Azure.

## Instalación de la CLI de Azure

La CLI de Azure es un conjunto de comandos que se pueden usar para crear y administrar recursos de Azure. Puede utilizar la CLI en el navegador web en el portal de Azure o puede [descargar una versión adecuada](#) para instalarla localmente en su equipo. Por ejemplo, en macOS puede instalar la CLI de Azure con el comando `brew install azure-cli`.

Puede interactuar con Azure a través del portal con el comando `az` de la siguiente manera:

```
$ az account show
{
  "environmentName": "AzureCloud",
  "id": "b5627140-9189-4305-a94c-3b199afe86791",
  "isDefault": true,
  "name": "Visual Studio Enterprise",
  "state": "Enabled",
  "tenantId": "baeb0498-d28a-41cd-a20d-d9409814a06k",
  "user": {
    "name": "cloudnative@xyz.com",
    "type": "user"
  }
}
```

Incluso puede usar la CLI de Azure para crear un script de forma declarativa de su infraestructura de nube. Estos scripts se pueden ejecutar en PowerShell o Bash. Aunque los scripts de Azure funcionan bien para tareas como la creación, el desmontaje y la nueva implementación de infraestructura, otras tareas como la actualización de un entorno existente no son sencillas debido a la falta de idempotencia en los comandos de la CLI de Azure.

Además de Azure, hay otras herramientas disponibles para crear y administrar entornos de nube. A continuación, veremos algunas de las más importantes.

## Herramientas de IaC destacadas

Como se mencionó anteriormente, se pueden usar otras herramientas para crear infraestructura nativa de la nube siguiendo el enfoque de IaC. Aquí nos centraremos en tres de las herramientas de open source más populares que pueden usarse con diversos proveedores de nube:

- Terraform, una herramienta de IaC basada en CLI
- Packer, una herramienta de CLI para crear imágenes de máquinas
- Ansible, una herramienta de administración de configuración

También puede usar todas estas herramientas en configuraciones de integración continua/implementación continua (CI/CD). En este escenario, apenas escribe una nueva configuración, se puede integrar de forma continua en la canalización de implementación.

## Terraform

Terraform es una herramienta de IaC de open source de HashiCorp, escrita en el lenguaje de programación Go. La palabra *terraform* significa transformar el entorno de un planeta para que se asemeje profundamente a la atmósfera de la Tierra de manera tal que la especie humana pueda vivir en él. Esta analogía es válida si considera los proveedores de nube como entornos que necesitan transformarse en una infraestructura administrable para ayudar a que su aplicación aproveche al máximo la nube y alcance su verdadero potencial.

Terraform le ayuda a crear, cambiar y controlar la versión de la infraestructura en la nube de una manera segura y eficaz. Utiliza una sintaxis declarativa, que significa que solo tiene que indicarle *lo* que desea compilar en lugar de *cómo* desea compilarlo. Una de las principales ventajas de Terraform es su compatibilidad con un lenguaje específico del dominio (DSL) fácil de entender llamado Lenguaje de Configuración HashiCorp (HCL), que es relativamente fácil de interpretar para los seres humanos. Como alternativa, puede usar JSON con el mismo propósito. Además, Terraform es independiente de la nube y, por lo tanto, compatible con varios proveedores de nube, lo que hace que sea realmente fácil de aprender ya que no tiene que aprender el lenguaje (HCL/JSON) de nuevo para un proveedor de nube diferente. En esta sección, analizaremos cómo puede aprovechar Terraform para crear infraestructura en Azure mediante la filosofía de IaC.

Para que entienda cómo funciona Terraform, primero tenemos que revisar algunos términos básicos:

### *Proveedores*

Un proveedor de Terraform es generalmente (pero no siempre) un proveedor de nube. La responsabilidad del proveedor es exponer los recursos (por ejemplo, recursos de la nube) e interpretar la API. Algunos de los proveedores compatibles con Terraform son Azure, Amazon Web Services (AWS), Google Cloud Platform (GCP) y PagerDuty.

### *Planificar*

Cuando crea su infraestructura mediante el comando `terraform plan`, Terraform crea un plan de ejecución para usted que se basa en el código que escribe mediante los archivos de configuración de Terraform (`.tf`). El plan de ejecución incluye los pasos que seguirá Terraform para compilar la infraestructura solicitada.

### *Aplicar*

Cuando esté satisfecho con el plan que creó Terraform, puede usar el comando `terraform apply` para crear la infraestructura, que finalmente llegará al estado deseado (si no hay conflictos o errores).

### Estado

Terraform necesita una forma de almacenar el estado de su infraestructura para que pueda administrarla. Los estados se almacenan en *archivos de estado* basados en texto que contienen la asignación de sus recursos de la nube a su configuración. Por lo general, hay dos formas en las que Terraform puede almacenar el estado de su infraestructura: de forma local en el sistema donde se inicializa o de forma remota. Almacenar el estado de forma remota es el método recomendado porque mantiene una copia de seguridad del estado de su infraestructura y más ingenieros pueden reunirse para crear infraestructura, ya que ahora hay una única fuente de verdad que permite conocer el estado actual de la infraestructura. Los archivos de estado no deben incluirse en un repositorio de Git ya que pueden contener secretos. En su lugar, puede utilizar el almacenamiento de blobs de Azure. Terraform también admite el bloqueo de forma predeterminada para los archivos de estado.

### Módulo

Un módulo de Terraform es un grupo de muchos recursos que se usan en conjunto. Los módulos ayudan a crear una infraestructura reutilizable al abstraer los recursos que se configuran una vez y se usan varias veces. Una de las principales ventajas de usar módulos es que le permiten reutilizar el código que, después, se puede compartir entre los equipos para diversos proyectos.

### Back-end

Un back-end describe cómo se cargará un estado. Es donde el archivo de estado se almacenará finalmente. Para este propósito, puede usar el almacenamiento de archivos local o el almacenamiento de blobs de Azure.

En la [Figura 2-4](#) se muestran los bloques del *proveedor* y de los *recursos*. El bloque del proveedor menciona la versión del administrador de recursos de Azure, mientras que los dos bloques de recursos crean el grupo de recursos de Azure y la red virtual.

```
provider "azurerm" {
    version = "2.0.0"
    features {}
}

resource "azurerm_resource_group" "example" {
    name      = "dummy-resource-group"
    location = "West Europe"
}

resource "azurerm_virtual_network" "example" {
    name          = "example-network"
    resource_group_name = azurerm_resource_group.example.name
    location       = azurerm_resource_group.example.location
    address_space  = ["10.0.0.0/16"]
}
```

Figura 2-4. Un archivo Terraform simple que representa la creación de recursos y el proveedor de Azure

Ahora que presentamos los fundamentos de Terraform, veamos cómo confluyen para aprovisionar la infraestructura nativa de la nube en Azure. En la [Figura 2-5](#) se muestra el flujo de trabajo general.

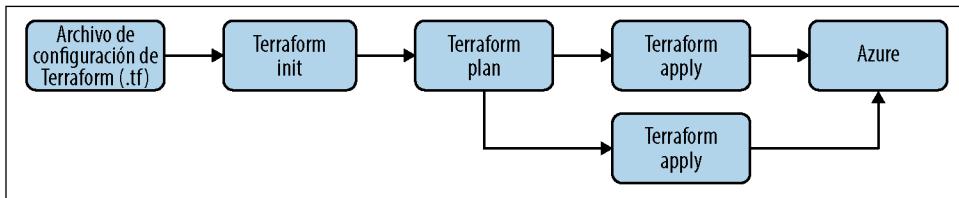


Figura 2-5. Flujo de trabajo de Terraform

Para empezar a usar Terraform, siga estos pasos:

1. Inicialice Terraform con el comando `terraform init`. No puede ejecutar los comandos `plan` y `apply` si no ha inicializado el directorio.
2. Una vez que haya inicializado correctamente Terraform, cree el plan de ejecución o un simulacro utilizando el comando `terraform plan`.
3. Cuando esté satisfecho con el resultado del plan, cree su infraestructura con el comando `terraform apply`.

Según el proveedor que utilice, la infraestructura se aprovisionará en el entorno correspondiente del proveedor de nube (en nuestro caso, es Microsoft Azure). Si desea destruir su infraestructura, simplemente puede emitir el comando `terraform destroy`, que eliminará la infraestructura correspondiente en el directorio actual. Es muy importante tener mucho cuidado al eliminar su infraestructura con Terraform.



Tenga mucho cuidado cuando elimine su infraestructura con Terraform. Una vez que se elimina una infraestructura, el archivo de estado se actualiza con los valores actualizados. Antes de eliminar su infraestructura, compruebe visualmente el resultado de `terraform destroy` para no eliminar accidentalmente información importante.

Ahora que comprende de forma general el flujo de trabajo y los detalles operativos, veamos de forma más práctica Terraform.

## Instalación de Terraform

Para instalar Terraform en su máquina,<sup>3</sup> primero debe buscar el **binary package** adecuado para su sistema operativo. Si usa Azure Cloud Shell, aparecerá la última versión de Terraform de forma predeterminada. En el momento de escribir este libro, la versión actual de Terraform es 0.12.28. Para hallar el paquete binario adecuado:

```
# For Linux-based systems:  
$ wget https://releases.hashicorp.com/terraform/0.12.28/terraform_0.12.28_linux_amd64.zip  
  
# For macOS-based systems:  
$ wget https://releases.hashicorp.com/terraform/0.12.28/terraform_0.12.28_darwin_amd64.zip
```

Una vez que tome el archivo ZIP, descomprima el paquete binario y mueva el binario solo a `/usr/local/bin`. Puede comprobar que Terraform está instalado mediante la ayuda de Terraform del shell:

```
$ ~ terraform --version  
Terraform v0.12.28  
$ ~ terraform --help  
Usage: terraform [-version] [-help] <command> [args]
```

```
The available commands for execution are listed below.  
The most common, useful commands are shown first, followed by  
less common or more advanced commands. If you're just getting  
started with Terraform, stick with the common commands. For the  
other commands, please read the help and docs before usage.
```

Common commands:

apply	Builds or changes infrastructure
console	Interactive console for Terraform interpolations
destroy	Destroy Terraform-managed infrastructure
env	Workspace management
fmt	Rewrites config files to canonical format
get	Download and install modules for the configuration
graph	Create a visual graph of Terraform resources
import	Import existing infrastructure into Terraform
init	Initialize a Terraform working directory
login	Obtain and save credentials for a remote host
logout	Remove locally-stored credentials for a remote host
output	Read an output from a state file
plan	Generate and show an execution plan
providers	Prints a tree of the providers used in the configuration
refresh	Update local state file against real resources
show	Inspect Terraform state or plan
taint	Manually mark a resource for recreation
untaint	Manually unmark a resource as tainted
validate	Validates the Terraform files
version	Prints the Terraform version
workspace	Workspace management

All other commands:

---

<sup>3</sup> Azure Cloud Shell incluye la versión más reciente de Cloud Shell. Para obtener información sobre cómo empezar a usar Terraform en Cloud Shell, visite <https://docs.microsoft.com/azure/developer/terraform/get-started-cloud-shell>.

```
0.12upgrade      Rewrites pre-0.12 module source code for v0.12
debug           Debug output management (experimental)
force-unlock    Manually unlock the terraform state
push            Obsolete command for Terraform Enterprise legacy (v1)
state           Advanced state management
$ ~
```

## Configuración del acceso de Terraform a una cuenta de Microsoft Azure

Ahora que configuró su cuenta de Azure y Terraform localmente, puede configurar Terraform para acceder a su cuenta de Azure:

1. Haga clic en el ícono de Cloud Shell en la esquina superior derecha de su cuenta del portal de Azure y, luego, haga clic en "Create storage". Esto creará una nueva cuenta de almacenamiento para usted en unos minutos.
2. Después de seleccionar Cloud Shell, puede elegir Bash o PowerShell. También puede cambiar el shell más adelante si lo desea. A lo largo de este libro, usaremos el shell Bash. Una vez completada la configuración, verá lo siguiente:

```
Requesting a Cloud Shell.Succeeded.
Connecting terminal...
Welcome to Azure Cloud Shell

Type "az" to use Azure CLI
Type "help" to learn about Cloud Shell

$
```

3. Ahora debe obtener una lista de los valores de ID de suscripción y ID de inquilino. Para ello, ejecute el siguiente comando en Cloud Shell para enumerar todos los nombres de cuentas de Azure, ID de suscripción y ID de inquilino:

```
$ az account list --query "[].{name:name, subscriptionId:id, tenantId:tenantId}"
[
  {
    "name": "Visual Studio Enterprise",
    "subscriptionId": "b1234567-89017-6135-v94s-3v16ifk86912",
    "tenantId": "ba0198-d28a-41ck-a2od-d8419714a098"
  }
]
```

4. Anote el valor de `subscriptionId` del resultado JSON. Luego, mientras sigue en Cloud Shell, reemplace el valor de `subscriptionId` por el siguiente comando:

```
$ az account set --subscription="b1234567-89017-6135-v94s-3v16ifk86912"
```

El comando anterior debería ejecutarse sin mostrar ningún resultado.

5. Cree una entidad de servicio<sup>4</sup> para usarla con Terraform con el siguiente comando en Cloud Shell:

```
$ az ad sp create-for-rbac --role="Contributor" |
  --scopes="/subscriptions/b1234567-89017-6135-v94s-3v16ifk86912"

Creating a role assignment under the scope of \
  "/subscriptions/b1234567-89017-6135-v94s-3v16ifk86912"
  Retrying role assignment creation: 1/36
{
  "appId": "b0b88757-6ab2-3v41-1234-23ss224vd41e",
  "displayName": "azure-cli-2020-07-03-18-24-17",
  "name": "http://azure-cli-2020-07-03-18-24-17",
  "password": "ah2cK.asfbbashfbjafsADNknvsaklvQQ",
  "tenant": "ba0198-d28a-41ck-a2od-d8419714a098"
}
$
```

Este comando devuelve el `appId`, `displayName`, `name`, , `password` y `tenant`.

6. Use los siguientes datos para configurar las variables de entorno de Terraform en la máquina local:

```
ARM_SUBSCRIPTION_ID=<subscription>
ARM_CLIENT_ID=<appId>
ARM_CLIENT_SECRET=<password>
ARM_TENANT_ID=<tenant>
```

7. En su `.bash_profile` (o, si prefiere, `envvar.sh`), reemplace las variables en el código anterior de la siguiente manera:

```
#!/bin/sh
echo "Setting environment variables for Terraform"
export ARM_SUBSCRIPTION_ID=b1234567-89017-6135-v94s-3v16ifk86912
export ARM_CLIENT_ID=b0b88757-6ab2-3v41-1234-23ss224vd41e
export ARM_CLIENT_SECRET=ah2cK.asfbbashfbjafsADNknvsaklvQQ
export ARM_TENANT_ID=ba0198-d28a-41ck-a2od-d8419714a098

# Not needed for public, required for usgovernment, german, china
export ARM_ENVIRONMENT=public
```

8. Utilice `source .bash_profile` en su terminal para leer y ejecutar comandos desde el archivo al entorno de shell actual.

Ahora puede usar Cloud Shell para interactuar con los recursos de Azure o, si lo prefiere, utilizar la CLI de Azure en su equipo local para interactuar con Azure.

---

<sup>4</sup> Una entidad de servicio de Azure es una identidad creada para usar con aplicaciones, servicios hospedados y herramientas automatizadas para acceder a los recursos de Azure. Este acceso está restringido por los roles asignados a la entidad de servicio, lo que le permite controlar a qué recursos se puede acceder y en qué nivel.

En este punto, debe iniciar sesión en Azure con la línea de comandos, que lo redirigirá al navegador para autenticarse. Después de autenticarse correctamente, podrá iniciar sesión:

```
$ ~ az login
You have logged in. Now let us find all the subscriptions to which you have access...
[
  {
    "cloudName": "AzureCloud",
    "id": "b1234567-89017-6135-v94s-3v16ifk86912",
    "isDefault": true,
    "name": "Visual Studio Enterprise",
    "state": "Enabled",
    "tenantId": "ba0198-d28a-41ck-a2od-d8419714a098",
    "user": {
      "name": "nishant7@hotmail.com",
      "type": "user"
    }
  }
]
```

Ahora que ha iniciado sesión correctamente y se ha autenticado con Azure, puede revisar el directorio `~/.azure`, que contiene la información de autenticación y autorización:

```
$ ls -al
total 44
drwxr-xr-x 1 nsingh nsingh 270 Apr 13 21:13 .
drwxr-xr-x 1 nsingh nsingh 624 Apr 13 21:13 ..
-rw------- 1 nsingh nsingh 7842 Apr 13 21:13 accessToken.json
-rw-r--r-- 1 nsingh nsingh 5 Apr 13 21:13 az.json
-rw-r--r-- 1 nsingh nsingh 5 Apr 13 21:13 az.sess
-rw-r--r-- 1 nsingh nsingh 420 Apr 13 21:21 azureProfile.json
-rw-r--r-- 1 nsingh nsingh 66 Apr 13 21:21 clouds.config
-rw-r--r-- 1 nsingh nsingh 5053 Apr 13 21:13 commandIndex.json
drwxr-xr-x 1 nsingh nsingh 318 Apr 13 21:21 commands
-rw------- 1 nsingh nsingh 51 Apr 13 21:13 config
drwxr-xr-x 1 nsingh nsingh 26 Apr 13 21:13 logs
drwxr-xr-x 1 nsingh nsingh 10 Apr 13 21:13 telemetry
-rw-r--r-- 1 nsingh nsingh 16 Apr 13 21:13 telemetry.txt
-rw-r--r-- 1 nsingh nsingh 211 Apr 13 21:13 versionCheck.json
```

## Configuración básica de infraestructura y uso con Terraform

Ahora que configuró Terraform correctamente para comunicarse con Azure, puede crear alguna infraestructura básica. A medida que avance por estos pasos, podrá conectar los puntos y usar Terraform de manera eficaz.

Todo el código de Terraform relacionado con este capítulo está disponible en el [repositorio de GitHub](#) del libro, que puede clonar en su máquina local.

Vaya al directorio `Test` y encontrará un archivo llamado `test.tf`. En el [Ejemplo 2-1](#) se muestra el contenido de este archivo. En este ejemplo, estamos usando un archivo de estado local para almacenar el estado actual de la infraestructura.

### Ejemplo 2-1. El archivo test.tf

```
provider "azurerm" {
}
resource "azurerm_resource_group" "rg" {
    name = "testResourceGroup"
    location = "westus"
}
```

El código creará un grupo de recursos llamado `testResourceGroup` en `westus`.

Para ejecutar este archivo, siga estos pasos:

1. Ejecute `terraform init` para inicializar Terraform y descargue el proveedor AzureRM:

```
$ Test git:(master) terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "azurerm" (hashicorp/azurerm) 2.17.0...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.azurerm: version = "~> 2.17"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

2. Despues de inicializar de forma correcta, ejecute `terraform plan`:

```
$ Test git:(master) X terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

-----
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create
Terraform will perform the following actions:
```

```

# azurerm_resource_group.rg will be created
+ resource "azurerm_resource_group" "rg" {
    + id      = (known after apply)
    + location = "westus"
    + name     = "testResourceGroup"
}

Plan: 1 to add, 0 to change, 0 to destroy.

-----
Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.

```

- Como puede ver en el resultado anterior, estamos tratando de crear un grupo de recursos con el nombre `testResourceGroup`. Ahora puede finalmente crear el recurso mediante la emisión de `terraform apply`. Recuerde que debe escribir explícitamente `yes` para poder continuar:

```

$ Test git:(master) ✘ terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# azurerm_resource_group.rg will be created
+ resource "azurerm_resource_group" "rg" {
    + id      = (known after apply)
    + location = "westus"
    + name     = "testResourceGroup"
}

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

Enter a value: yes

azurerm_resource_group.rg: Creating...
azurerm_resource_group.rg: Creation complete after 3s
[id=/subscriptions/b5627140-9087-4305-a94c-3b16afe86791/resourceGroups/ \
testResourceGroup]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
$ Test git:(master) ✘

```

Esto debería crear un grupo de recursos, como se muestra en la [Figura 2-6](#). Puede revisar el portal de Azure para verificarlo.

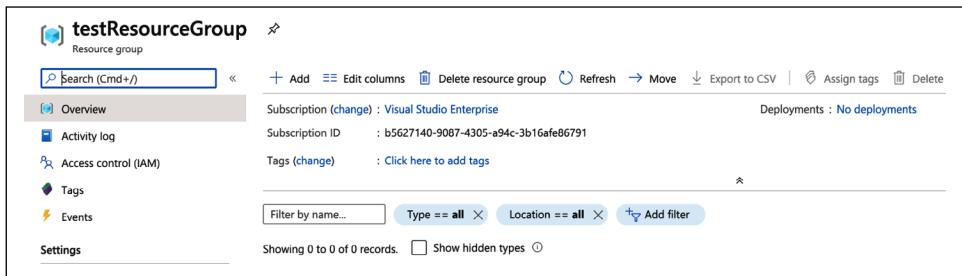


Figura 2-6. Creación de un grupo de recursos simple con Terraform

También debería verlo desde la CLI. Tenga en cuenta que la operación de shell *no* es específica de la región. Puede ver los recursos en varias ubicaciones. Use el siguiente comando para crear una lista del grupo de recursos:

```
$ az group list --output table
Name          Location    Status
-----
testResourceGroup    westus      Succeeded
```

En el directorio *Test*, es posible que observe un archivo llamado *terraform.tfstate* que se creó después de aprovisionar el grupo de recursos. Este es el archivo de estado que contiene el estado actual de la infraestructura.

### Explorar la infraestructura de Azure con Terraform

Ahora que creamos una infraestructura básica con Terraform, profundicemos y creemos infraestructura adicional.

En el [Ejemplo 2-1](#), le mostramos cómo almacenar su estado localmente. Aunque este no es un proceso recomendado, es suficiente para el uso básico, como en el caso del ejemplo donde creamos un grupo de recursos (*test.tf*). Es mejor usar el almacenamiento remoto, como el almacenamiento de blobs de Azure, que proporciona durabilidad y seguridad para las configuraciones de infraestructura.



Los estados de Terraform son la fuente de verdad y siempre deben almacenarse en un back-end, como Consul o Amazon DynamoDB o el almacenamiento de Azure, porque esto almacenará de forma segura el estado de la infraestructura junto con el control de versiones para facilitar la administración o la reversión en caso de un desastre. Los estados no se deben verificar en el sistema de control de versiones de Git, porque Git contiene secretos.

En las siguientes subsecciones, crearemos almacenamiento de blobs en nuestra cuenta de Azure para almacenar todos los estados de nuestra infraestructura. También seguiremos el principio de "No se repita" (DRY) al asegurarnos de usar el módulo de Terraform. Vamos a repasar la creación de redes virtuales y máquinas virtuales. Puede ver los módulos en este [repositorio de GitHub](#) de este capítulo.

**Creación de almacenamiento de blobs de Azure.** Para crear almacenamiento de blobs de Azure, debe usar el siguiente módulo: <https://bit.ly/3bM6jCx>.

Ahora, siga los pasos descritos en "[Configuración básica de infraestructura y uso con Terraform](#)" en la página 20 para implementar la cuenta de almacenamiento. Una vez que emita el comando `terraform plan`, verá una gran cantidad de información sobre el plan entrelazada con los módulos. También notará que los módulos (p. ej., `Storage_accounts`) se abstraen, mientras que el archivo de recursos se ve así:

```
module "CloudNativeAzure-strg-backend" {  
    source = "../Modules/Services/Storage_accounts"  
    resource-grp-name = "CloudNativeAzure-group"  
    storage-account-name = "cnabookprod"  
    azure-dc = "westus"  
    storage-account-tier = "Standard"  
    storage-replication-type = "LRS"  
    storage-container-name = "cloud-native-devs"  
    storage-container-access = "private"  
    blob-name = "cloud-native-with-azure"  
}
```

Esto demuestra el poder que tiene abstraer la infraestructura repetible detrás de los módulos. Una vez que use `terraform apply`, se ejecutará el programa y, después de unos minutos, verá una cuenta de almacenamiento llamada `cnabookprod` y un blob con el nombre `cloud-native-with-azure`:

```
.  
. .  
module.CloudNativeAzure-strg-backend.azurerm_storage_container.generic-container: Creation  
complete after 3s [id=https://cnabookprod.blob.core.windows.net/cloud-native-devs]  
module.CloudNativeAzure-strg-backend.azurerm_storage_blob.generic-blob: Creating...  
module.CloudNativeAzure-strg-backend.azurerm_storage_blob.generic-blob: Creation complete  
after 4s [id=https://cnabookprod.blob.core.windows.net/cloud-native-devs/cloud-native-with-  
azure]  
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.  
Outputs:  
Blob-ID = https://cnabookprod.blob.core.windows.net/cloud-native-devs/cloud-native-with-  
azure  
Blob-URL = https://cnabookprod.blob.core.windows.net/cloud-native-devs/cloud-native-  
with-azure  
Primary-Access-Key =  
Bfc9g/piV3XkJbGosJkjsjn13iLHevR3y1cuPyM8giGT4J0vXeAKAKvjsduTI1GZ45ALACLAWPAXA==
```

En la [Figura 2-7](#) se muestra el blob y la cuenta de almacenamiento de Azure.

Figura 2-7. Blob y cuenta de almacenamiento de Azure

Anote el resultado de la operación `terraform apply` en el paso anterior y copie el valor de la clave de acceso principal. Deberá actualizar el valor de la clave principal en el mismo `.bash_profile` en su máquina local de la siguiente manera:

```
#!/bin/sh
echo "Setting environment variables for Terraform"
export ARM_SUBSCRIPTION_ID=b1234567-89017-6135-v94s-3v16ifk86912
export ARM_CLIENT_ID=b0b88757-6ab2-3v41-1234-23ss224vd41e
export ARM_CLIENT_SECRET=aH2cK.asfbashfbjafsADNknvsaklvQQ
export ARM_TENANT_ID=ba0198-d28a-41ck-a2od-d8419714a098

# Not needed for public, required for usgovernment, german, china
export ARM_ENVIRONMENT=public
export
ARM_ACCESS_KEY=Bfc9g/piV3XkJbGosJkjjsjn13iLHevR3y1cuPyM8giGT4J0vXeAKAKvjdsuTI1GZ45AL
ACLAWPAXA==
```

Ahora podemos crear de forma segura más infraestructura y almacenar el estado de la infraestructura en el almacenamiento de blobs.

**Creación de una red virtual e instancias de Azure.** Ahora crearemos una red virtual en Azure a partir de la configuración de Terraform en <https://bit.ly/3bHE342>.

También explicaremos el flujo sintáctico del módulo de red virtual, que le permitirá comprender mejor la simplicidad del código.

Observe más de cerca el módulo `Virtual_network` en el directorio `/Modules/Services/Virtual_network` y verá el proceso paso a paso de la creación de la red virtual:

1. Cree un grupo de seguridad de red mediante `azurerm_network_security_group`.
2. Agregue la regla de seguridad y adjúntela al grupo de seguridad en el paso 1.
3. Cree un plan de protección DDoS.
4. Cree una red virtual.
5. Cree cuatro subredes diferentes dentro de la red virtual.
6. Cree una interfaz de red y adjúntela a una subred (privada).
7. Cree direcciones IP públicas.
8. Cree una subred pública adjuntando una IP pública a ella.

El resultado final de `apply` de Terraform se ve así:

```
Apply complete! Resources: 12 added, 0 changed, 0 destroyed.  
Releasing state lock. This may take a few moments...  
Outputs:  
  
private-nic-id = /subscriptions/b5627140-9087-4305-a94c-3b16afe86791/resourceGroups/  
CloudNativeAzure-group/providers/Microsoft.Network/networkInterfaces/private-nic  
private-subnet-a-id = /subscriptions/b5627140-9087-4305-a94c-3b16afe86791/resourceGroups/  
CloudNativeAzure-group/providers/Microsoft.Network/virtualNetworks/cna-prod/subnets/  
linkedin-private-a  
private-subnet-b-id = /subscriptions/b5627140-9087-4305-a94c-3b16afe86791/resourceGroups/  
CloudNativeAzure-group/providers/Microsoft.Network/virtualNetworks/cna-prod/subnets/  
linkedin-private-b  
pub-nic-id = /subscriptions/b5627140-9087-4305-a94c-3b16afe86791/resourceGroups/  
CloudNativeAzure-group/providers/Microsoft.Network/networkInterfaces/public-nic  
public-subnet-a-id = /subscriptions/b5627140-9087-4305-a94c-3b16afe86791/resourceGroups/  
CloudNativeAzure-group/providers/Microsoft.Network/virtualNetworks/cna-prod/subnets/  
linkedin-public-a  
public-subnet-b-id = /subscriptions/b5627140-9087-4305-a94c-  
3b16afe86791/resourceGroups/CloudNativeAzure-group/providers/Microsoft.Network/  
virtualNetworks/cna-prod/subnets/linkedin-private-b  
vpc-id = /subscriptions/b5627140-9087-4305-a94c-3b16afe86791/resourceGroups/  
CloudNativeAzure-group/providers/Microsoft.Network/virtualNetworks/cna-prod
```

En la [Figura 2-8](#), puede ver la red virtual que se está creando en el portal de Azure.

Figura 2-8. Creación de la red virtual

Como se muestra en la Figura 2-9, Terraform ha almacenado el estado de la red virtual en la cuenta de almacenamiento de blobs que creamos anteriormente.

```

1 ion": 4,
2 aform-version": "0.12.28",
3 a": 3,
4 age": "3436fc85-69de-ebb5-22d2-fe73a280be10",
5 ur:
6 vate-ric-id": {
7 value": "/subscriptions/b5627140-9887-4385-a94c-3b16afe86791/resourceGroups/CloudNativeAzure-group/provid
8 type": "string"
9
10 ivate-subnet-a-id": {
11 value": "/subscriptions/b5627140-9887-4385-a94c-3b16afe86791/resourceGroups/CloudNativeAzure-group/provid
12 type": "string"
13
14 ivate-subnet-b-id": {
15 value": "/subscriptions/b5627140-9887-4385-a94c-3b16afe86791/resourceGroups/CloudNativeAzure-group/provid
16 type": "string"
17
18 b-ric-id": {
19 value": "/subscriptions/b5627140-9887-4385-a94c-3b16afe86791/resourceGroups/CloudNativeAzure-group/provid
20 type": "string"
21
22 blate-subnet-a-id": {
23 value": "/subscriptions/b5627140-9887-4385-a94c-3b16afe86791/resourceGroups/CloudNativeAzure-group/provid
24 type": "string"
25
26

```

Figura 2-9. Estado de la red virtual cargado por Terraform en la cuenta de almacenamiento

Del mismo modo, puede seguir los pasos anteriores y lanzar una máquina virtual dentro de una de las subredes mediante el módulo de Terraform en <https://bit.ly/3bQjTEV>.



Una vez que haya creado los recursos en este ejemplo, asegúrese de destruirlos con el comando `terraform destroy` en cada directorio de recursos en caso de tener habilitada la facturación para su cuenta de Azure.

## Plantillas de Terraform y ARM

Antes de pasar a la siguiente sección, veamos lo que Azure ofrece de forma nativa para implementar la infraestructura como código. Como se mencionó antes, Azure usa plantillas de ARM para implementar la infraestructura como código para las cargas de trabajo basadas en Azure. Las plantillas de ARM vienen en un archivo JSON<sup>5</sup> que define la infraestructura y la configuración para su proyecto. El archivo usa una sintaxis declarativa similar a Terraform, que le permite definir la configuración de la infraestructura prevista. Además de la sintaxis declarativa, las plantillas de ARM también heredan idempotencia; es decir, puede implementar la misma plantilla varias veces y obtener los mismos tipos de recursos en el mismo estado. El administrador de recursos es responsable de convertir la plantilla de ARM en una operación de API de REST. Por ejemplo, si implementa la siguiente plantilla de ARM:

```
"resources": [
  {
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2019-04-01",
    "name": "teststorageaccount",
    "location": "eastus",
    "sku": {
      "name": "Standard_LRS"
    },
    "kind": "StorageV2",
    "properties": {}
  }
]
```

el administrador de recursos convertirá el JSON en una operación de API de REST, que se enviará al proveedor de recursos `Microsoft.Storage` de la siguiente manera:

```
PUT
https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/ \
{resourceGroupName}/providers/Microsoft.Storage/storageAccounts/ \
teststorageaccount?api-version=2019-04-01
REQUEST BODY
{
  "location": "eastus",
  "sku": {
    "name": "Standard_LRS"
  },
  "kind": "StorageV2",
  "properties": {}
}
```

También puede usar Terraform para implementar plantillas de ARM mediante el recurso `azurerm_resource_group_template_deployment` para el proveedor AzureRM.

---

5 Azure ofrece JSON (JavaScript Object Notation) de forma predeterminada para escribir plantillas de ARM. Azure también ofrece un lenguaje específico del dominio llamado Bicep que también usa la sintaxis declarativa para implementar recursos de Azure. Puede usar Bicep en lugar de JSON, ya que reduce la complejidad y mejora la experiencia general de desarrollo/administración.

# Packer

Al principio de este capítulo, presentamos el término *infraestructura inmutable*, que representa el cambio en el aprovisionamiento y la implementación de las aplicaciones nativas de la nube. Una de las herramientas más utilizadas para crear una imagen de máquina preparada previamente<sup>6</sup> es Packer. Las imágenes de máquina son principalmente recursos informáticos que tienen todas las configuraciones, metadatos, artefactos y archivos relacionados preinstalados/configurados. **Packer** es una herramienta de open source de HashiCorp que se usa para crear imágenes de máquina a partir de una configuración definida. Automatiza todo el proceso de creación de imágenes de máquina, lo que aumenta la velocidad de implementación de la infraestructura. Al igual como Terraform se utiliza para crear infraestructura, Packer ayuda a crear imágenes de máquina al preparar todo el software esencial, los binarios, etc., en la imagen de la máquina.

Además, dado que todo el software de la imagen de máquina está instalado y configurado antes de que se cree la imagen, Packer ayuda a mejorar la estabilidad general de la infraestructura. También es compatible con diferentes proveedores de nube, que finalmente crean imágenes de máquina idénticas en varias plataformas. Profundicemos un poco más en cómo se puede usar Packer para crear imágenes en Azure.

## Instalación de Packer

Para empezar a trabajar con Packer, descargue el **binario de Packer** adecuado para su máquina local<sup>7</sup>. Descomprima el paquete binario y muévalo a `/usr/local/bin`. De esta forma, se debería configurar Packer en su máquina. Para verificarlo, puede ejecutar `packer --help` en su shell:

```
$ ~ packer --help
Usage: packer [--version] [--help] <command> [<args>]

Available commands are:
  build      build image(s) from template
  console    creates a console for testing variable interpolation
  fix        fixes templates from old versions of packer
  inspect   see components of a template
  validate  check that a template is valid
  version   Prints the Packer version
```

## Creación de una imagen de Linux en Azure

Packer utiliza un archivo de configuración que define la imagen de la máquina. El archivo de configuración se denomina *plantilla* de Packer y está escrito en JSON. La plantilla incluye *creadores* y *aprovisionadores*. Los creadores crean una imagen de máquina para una plataforma mediante la lectura de las claves de configuración y autenticación. Los aprovisionadores son responsables de instalar y

---

<sup>6</sup> *Preparado* previamente es un término que se usa para referirse a la preinstalación de software en una imagen de máquina, que se puede usar más adelante para poner en marcha más instancias o contenedores.

<sup>7</sup> Packer está disponible en Azure Cloud Shell de forma predeterminada.

configurar el software sobre la imagen de la máquina antes de que la imagen se convierta en immutable. En la [Figura 2-10](#) se muestra el proceso de creación de imágenes de Packer.



*Figura 2-10. Proceso de creación de imágenes con Packer*

Puede consultar el código relacionado con la creación de imágenes de máquina de Packer en <https://bit.ly/3GRgswf>. Para ejecutar este código, primero debe actualizar el código con los detalles de autenticación en el archivo *example.json*. Puede usar los detalles de autenticación de su cuenta de Azure desde el archivo *bash\_profile*. El resultado de la ejecución debería ser similar a lo siguiente:

```
$ 1.2.2.2 git:(master) ✘ packer build example.json
azure-arm: output will be in this color.

==> azure-arm: Running builder ...
==> azure-arm: Getting tokens using client secret
==> azure-arm: Getting tokens using client secret
    azure-arm: Creating Azure Resource Manager (ARM) client ...
==> azure-arm: WARNING: Zone resiliency may not be supported in East US, checkout the
docs at https://docs.microsoft.com/en-us/azure/availability-zones/
==> azure-arm: Creating resource group ...
==> azure-arm:   -> ResourceGroupName : 'pkr-Resource-Group-k9831ev1uk'
==> azure-arm:   -> Location           : 'East US'
==> azure-arm:
.
.
.
.

==> azure-arm:
Build 'azure-arm' finished.

==> Builds finished. The artifacts of successful builds are:
--> azure-arm: Azure.ResourceManagement.VMImage:

OSType: Linux
ManagedImageResourceGroupName: CloudNativeAzure-group
ManagedImageName: myfirstPackerImage
ManagedImageId: /subscriptions/b1234567-89017-6135-v94s-3v16ifk86912/resourceGroups/
    CloudNativeAzure-group/providers/Microsoft.Compute/images/myfirstPackerImage
ManagedImageLocation: East US
```

En la [Figura 2-11](#) se muestra la imagen que creamos en el portal de Azure. Puede comprobar que la imagen tendrá el servidor web Nginx y el equilibrador de carga y el servidor proxy de HAProxy incorporados al poner en marcha las máquinas virtuales usando la imagen.

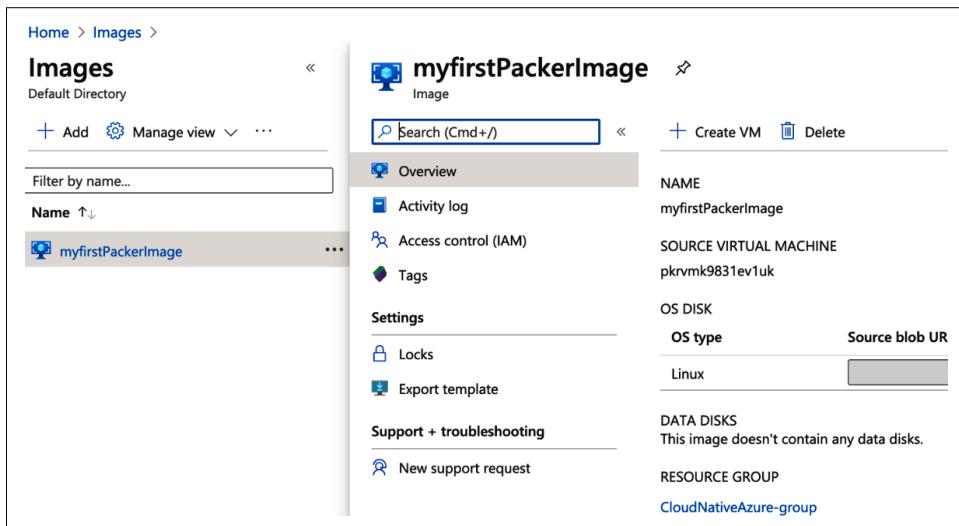


Figura 2-11. Creación de imágenes de máquina con Packer

Ahora que entiende el rol de Packer como una herramienta para la creación de imágenes, veamos Ansible, que es un gran complemento para la administración de entornos nativos de nube complejos.

## Ansible

Ansible es una herramienta de administración de configuración simple que le permite aprovisionar, configurar y administrar servidores. Ansible facilita la administración de servidores que no tienen ningún agente preinstalado en ellos, aunque también puede configurar Ansible como un sistema basado en un agente. Además, puede aprovechar Ansible como una herramienta una vez que haya creado la infraestructura utilizando Terraform y las imágenes de máquina preparadas previamente con Packer.

Los conceptos básicos de Ansible son los siguientes:

### Manuales

Los manuales de Ansible son una lista de tareas o *movimientos* que deben ejecutarse en una instancia. Un ejemplo simple sería:

- Agregar el repositorio de Nginx.
- Instalar Nginx.
- Crear el directorio raíz.

Esta lista ordenada se puede convertir en un manual de Ansible de la siguiente manera:

```
---
- hosts: local
  vars:
    - docroot: /var/www/serversforhackers.com/public
```

```

tasks:
  - name: Add Nginx Repository
    apt_repository: repo='ppa:nginx/stable' state=present

  - name: Install Nginx
    apt: pkg=nginx state=installed update_cache=true

  - name: Create Web Root
    file: dest=/etc/nginx
      mode=775
      state=directory
      owner=www-data
      group=www-data
    notify:
      - Reload Nginx

```

### *Nodo de control*

Este es normalmente el nodo desde el que ejecutará el manual de Ansible. Este podría ser cualquier nodo donde esté instalado Ansible. Para ejecutar un manual de Ansible, puede ejecutar el siguiente comando en su terminal:

```
ansible-playbook -vi path_to_host_inventory_file playbook.yaml
```

### *Nodos administrados*

Los nodos o hosts que desea administrar se denominan nodos administrados. Estos nodos suelen ser los servidores remotos. Lo mejor de Ansible es que no necesita instalar un agente en las instancias remotas para administrarlos. Todo lo que necesita es un demonio de Secure Shell (SSH) que escuche y Python instalado en los hosts. Ansible crea una conexión SSH para ejecutar los manuales.

### *Archivo de inventario*

El archivo de inventario contiene la lista de todos los hosts que administra Ansible. Por lo general, el inventario contiene direcciones IP, nombres de host, el nombre de usuario de SSH y las claves para conectarse a la máquina remota. Este es un ejemplo del inventario:

```

mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com

[midtier servers]
10.0.1.2
10.2.3.4

```

Dado que Ansible no requiere mucha sobrecarga y está escrito en YAML, actúa como una gran herramienta posterior a la configuración. A veces, utilizaremos Ansible en este libro para aprovisionar partes de los servicios.

Antes de concluir este capítulo, veamos rápidamente Azure DevOps, que nos ayuda a combinar todas estas herramientas de forma automatizada para crear una canalización de CI/CD.

## Azure DevOps y la infraestructura como código

Azure ofrece *Azure DevOps*, que proporciona una cadena de herramientas de DevOps completa para desarrollar e implementar aplicaciones, y es principalmente un servicio hospedado para implementar canalizaciones de CI/CD. También puede usar Terraform, Ansible y Packer juntos para crear canalizaciones en varias etapas para sus proyectos. Azure DevOps tiene un conjunto de servicios, incluido *Azure Pipeline*, que se utiliza esencialmente para compilar, probar e implementar las canalizaciones de CI/CD. No profundizaremos en Azure DevOps, ya que va más allá del alcance de este libro, pero puede obtener más información sobre cómo configurar Azure DevOps en la [documentación oficial](#).

## Resumen

En este capítulo, presentamos los pasos clave para crear un entorno nativo de la nube moderna con Azure. Creó una cuenta de Azure, que será útil en los próximos capítulos a medida que comprenda con mayor profundidad las tecnologías nativas de la nube. También se presentó Terraform, Packer y Ansible, tres útiles orquestadores nativos de la nube que lo ayudarán a implementar y administrar su infraestructura en la nube. Terraform es una herramienta independiente de la nube para el desarrollo de la infraestructura como código desde cero. Packer se usa para crear imágenes de máquina para el desarrollo de artefactos inmutables y Ansible es una herramienta para la administración de configuración.

Ahora que maneja esta información básica de Azure y las tecnologías relacionadas, podemos pasar al siguiente capítulo, donde hablaremos sobre los contenedores, el registro de contenedores y la creación de contenedores en su aplicación.



# Contenedorizar su aplicación: Más que cajas

En los últimos años, la popularidad de los contenedores ha aumentado. No solo ofrecen baja sobrecarga, alta seguridad y alta portabilidad, sino que también siguen los principios recomendados de la nube, como la inmutabilidad, la fugacidad y el escalado automático.

En este capítulo, explicaremos qué son los contenedores y hablaremos sobre las plataformas populares de contenedorización. Además, veremos más allá del revuelo general del sector y explicaremos por qué debería usar contenedores. También destacaremos algunas de sus ventajas, especialmente cuando se trata de la nube.

## ¿Por qué usar contenedores?

Los contenedores se ejecutan sobre el sistema operativo del host y, por lo general, se orquestan mediante software, como Docker (como hablaremos más adelante en este capítulo) o Kubernetes (consulte la [Figura 3-1](#)).

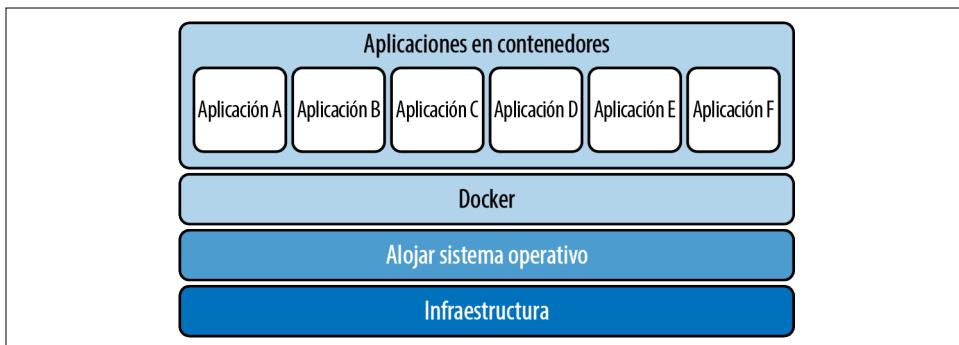


Figura 3-1. El paradigma de la operación de contenedores

Una de las ventajas más importantes de los contenedores es que proporcionan aislamiento similar a una máquina virtual, pero sin la sobrecarga de ejecutar una instancia del sistema operativo host para cada contenedor. A escala, esto puede ahorrar una cantidad significativa de recursos del sistema.

## Aislamiento

Los contenedores proporcionan aislamiento a través de una serie de métodos de una manera flexible que el administrador configura. Los contenedores se pueden aislar de las siguientes maneras:

### *Recursos del sistema*

El aislamiento se logra a través de CPU, memoria y grupos de control (cgroup) del disco.

### *Espacios de nombre*

El aislamiento se logra al permitir espacios de nombres independientes para cada contenedor.

### *Límites de POSIX*

El aislamiento con este método le permite establecer rlimits en su contenedor.

Estos mecanismos le permiten ejecutar de forma segura varias cargas de trabajo juntas sin preocuparse por la contención de recursos o que un sistema moleste a otro. Si tiene un orquestador de contenedores que reconozca la carga de trabajo, puede usar todos los recursos de modo eficaz en un equipo host sin degradación del rendimiento (conocido como *empaqueado en contenedores*). Si se hace bien a escala, esto puede crear ahorros de costos significativos a medida que aumenta la eficiencia de la infraestructura.

## Seguridad

A partir de las características de aislamiento de los contenedores, la **especificación de tiempo de ejecución del contenedor OCI** (que se explica más adelante en este capítulo) proporciona una serie de características relacionadas con la seguridad del tiempo de ejecución que aseguran que si una aplicación dentro de un contenedor está comprometida, se reduce significativamente el riesgo de movimiento lateral en toda la red.

Algunas de estas características incluyen lo siguiente:

### *Capacidades de Linux*

Esto reduce el acceso del contenedor a las API del kernel.

### *Atributos de usuario/grupo*

Establecen los atributos de usuario y grupo del contenedor, que luego pueden limitar el acceso a los recursos del sistema de archivos.

### *Dispositivos*

Controlan a qué dispositivos tiene acceso un contenedor.

### *Etiquetas de SELinux*

Al igual que otros componentes de un sistema Linux (archivos, procesos, canalizaciones), las etiquetas de SELinux se pueden aplicar a los contenedores.

### *Sysctl*

Esto agrega controles adicionales (limitados) al contenedor.

Estas capas de características de seguridad del tiempo de ejecución proporcionan una forma determinista y administrada para implementar aplicaciones con un riesgo significativamente menor para la infraestructura del host y las aplicaciones vecinas. Dentro del ecosistema de contenedores también hay un conjunto de paradigmas de seguridad de artefactos, que trataremos más adelante en este capítulo.

Tenga en cuenta que si desea aplicar la política del sistema a través de un mecanismo de orquestación (Puppet/Chef, etc.), es mucho más trabajo implementarla por aplicación que simplemente usar las características que proporciona el contenedor.

## **Empaquetado e implementación**

Como analizaremos más adelante en este libro, la estandarización de las imágenes de contenedores y la portabilidad de esas imágenes es extremadamente útil. Es relativamente sencillo tomar un contenedor que se ejecuta en su escritorio e implementarlo en producción sin una sobrecarga operativa significativa o sin tener que realizar pasos adicionales de configuración y administración. El lema de Docker es "crear, enviar y ejecutar", lo que resalta la mentalidad de escritorio a producción que la contenedorización aporta al ciclo de vida de desarrollo de software.

Además, teniendo en cuenta la especificación OCI de las imágenes, como veremos más adelante en este capítulo, puede estar seguro de que el sistema de implementación se realizará correctamente sin problemas, siempre y cuando su registro de contenedor esté disponible.

## **Primitivas de contenedores básicos**

Puede que le sorprenda saber que los contenedores no son un concepto de primera clase dentro de Linux. De hecho, no existía una definición formal de contenedor hasta que se creó la [especificación OCI en 2017](#) (hablaremos más sobre las especificaciones más adelante en este capítulo). Un contenedor básico, por lo general, se compone de los siguientes primitivos:

- Cgroups
- Espacios de nombre
- Copy on write (CoW)
- Seccomp-BPF

Los contenedores de Linux son una evolución natural de tecnologías similares, como las cárceles de FreeBSD, las zonas de Solaris y las máquinas virtuales. De muchas maneras, los contenedores reúnen lo mejor de estas tecnologías.

Debido a que los contenedores no son un concepto primitivo, hay disponible una serie de plataformas de software de contenedores. Las exploraremos más adelante en este capítulo. Por ahora, revisemos cada uno de los primitivos de contenedores básicos.

## Cgroups

Un grupo de control (cgroup) es un primitivo de Linux que proporciona la capacidad de limitar los recursos a los que un grupo de programas dentro de un cgroup puede acceder/asignar. Esto brinda protección contra el problema del "vecino ruidoso". A continuación, hay una lista de controles que proporcionan los cgroups:

### CPU

El control del subsistema de CPU le permite asignar una cantidad determinada de tiempo de CPU (`cpuacct`) o un número específico de núcleos (`cpuset`) a un cgroup, para que pueda asegurarse de que un proceso no sature todos los recursos de CPU en una máquina física.

### Memoria

El subsistema de memoria le permite registrar y establecer límites para el uso de memoria en un cgroup. Esto funciona para la memoria de usuario y la memoria de intercambio.

### Blkio

El subsistema de controlador blkio (se pronuncia bloquio) ofrece un mecanismo para limitar las operaciones de entrada/salida por segundo (IOPS) o el ancho de banda (bps) de un cgroup a un dispositivo de bloque.

### PID

El controlador PID (identificador de proceso) establece el número máximo de procesos que se pueden ejecutar en el cgroup.

### Dispositivos

Este control, que solo es válido para cgroups v1, permite al administrador establecer listas para permitir/denegar qué dispositivos puede tener acceso un cgroup en la máquina host.

### Red

El cgroup de clasificador de red (`net_cls`) proporciona una interfaz para etiquetar paquetes de red con un identificador de clase (`classid`).

El cgroup de prioridad de red (`net_prio`) proporciona una interfaz para establecer dinámicamente la prioridad del tráfico de red generado por varias aplicaciones.

En su mayoría, **eBPF** proporciona una mejor funcionalidad de control de red en las versiones más recientes del kernel de Linux.

## Espacios de nombre

Los **espacios de nombres** son una característica de Linux que le permite limitar los recursos no informáticos para que un grupo de procesos vea un conjunto de recursos y un segundo grupo de procesos vea un conjunto diferente de recursos. En el momento de escribir este libro, Linux contiene el siguiente conjunto de tipos de espacio de nombres:

### *Montaje (mnt)*

Controla los montajes de sistema de archivos que están disponibles para el contenedor. Esto significa que los procesos en diferentes espacios de nombres de montaje ven diferentes vistas de una jerarquía de directorios.

### *PID (pid)*

Le proporciona al contenedor un conjunto de PID numerados de forma independiente. Esto significa que el primer proceso dentro de un contenedor con su propio espacio de nombre de PID tendrá un PID de 1. Cualquier descendiente de PID 1 actuará como un sistema Unix normal.

### *Red (net)*

Es una característica excepcionalmente útil que le permite virtualizar su infraestructura de red. Cada espacio de nombres tendrá un conjunto privado de **direcciones IP** y su propia **tabla de enrutamiento**, lista de **sockets**, tabla de seguimiento de conexiones, **firewall**, y otros recursos relacionados con la red.

### *Comunicación entre procesos (ipc)*

Aísla los procesos de la comunicación entre procesos de Linux (IPC). Los ejemplos incluyen canalizaciones y señales.

### *Uso compartido de tiempo de Unix (uts)*

Permite que un solo sistema parezca tener diferentes **nombres de host** y **de dominio** para diferentes procesos.

### *ID de usuario (user)*

Es similar al espacio de nombres de PID. Le permite tener privilegios elevados en el contenedor, pero no en el sistema general. El espacio de nombres **user** proporciona segregación de identificación de usuarios entre espacios de nombres. El espacio de nombres **user** contiene una tabla de asignación que convierte los ID de usuarios del punto de vista del contenedor al punto de vista del sistema. Esto permite, por ejemplo, que el usuario raíz tenga el ID de usuario 0 en el contenedor, pero que el sistema lo trate como ID de usuario 1.400.000 para las comprobaciones de propiedad.

### *Grupo de control (cgroup)*

Oculta la identidad del **grupo de control** del que el proceso es un miembro (esto es independiente del primitivo cgroup).

### *Tiempo (time)*

Permite que los procesos vean diferentes tiempos del sistema de manera similar al espacio de nombres UTS.

## **Copy on Write**

Copy on Write (CoW) es una técnica de administración de memoria que solo copia memoria cuando se modifica (o se escribe) un recurso. Esta técnica reduce la cantidad de memoria necesaria.

## **Capacidades**

Las capacidades de Linux permiten al administrador controlar de forma más estrecha qué capacidades tiene el proceso o cgroup en el sistema host. Puede ver una lista de capacidades en la [página man de capacidades\(7\)](#). Por ejemplo, si quisiera ejecutar un servidor web en el puerto 80, solo tendría que dar la capacidad CAP\_NET\_BIND\_SERVICE al cgroup/process. Si el servidor web se ve comprometido, las acciones del atacante estarían limitadas, ya que el proceso no tiene permisos del sistema para llevar a cabo otros comandos administrativos.

## **Seccomp-BPF**

Seccomp (SECure COMPUTing) es útil para crear restricciones absolutas en todos los sistemas. Seccomp-BPF permite un control detallado en cada aplicación. Seccomp-BPF ofrece granularidad por subprocesso (en modo estricto). Se agregó al kernel en la versión 3.5 (2012).

Seccomp-BPF permite que los programas de Berkeley Packet Filter (BPF) (programas que se ejecutan en el espacio del kernel) filtren syscalls (y sus argumentos) y devuelvan un valor sobre lo que debería ocurrir después de que se cierra el programa Seccomp-BPF.

## **Componentes de la ejecución de un contenedor**

Como hablamos anteriormente en este capítulo, los contenedores no son un concepto de primera clase en Linux (no hay un primitivo de contenedor de Linux), lo que hace que sean levemente difíciles de definir. Esto significa que antes era difícil hacer que los contenedores fueran portátiles. Sin embargo, para ejecutar un contenedor, debe haber algún tipo de interfaz definida entre la construcción del contenedor y el software del sistema operativo (o software host).

En esta sección, analizaremos los puntos más precisos del software de contenedor, incluidos los tiempos de ejecución de contenedores (el software que ejecuta el contenedor), las plataformas de contenedores y los orquestadores de contenedores.

En primer lugar, veamos rápidamente las capas de abstracción en el ecosistema de contenedores ([Figura 3-2](#)).



*Figura 3-2. Las capas de abstracción de contenedores*

En esta sección, veremos estas capas de abstracción de arriba a abajo para ver cómo trabajamos desde contenedores perfectamente orquestados hasta conceptos de Linux de bajo nivel que permiten que los contenedores operen.

## Orquestadores de contenedores

En la capa superior, se encuentran los orquestadores de contenedores que llevan a cabo la mágica tarea de administrar el escalado automático, el reemplazo de instancias y la supervisión de un ecosistema de contenedores, lo que hace que los contenedores sean un concepto tan atractivo. Como analizaremos en el [Capítulo 4](#) y [Capítulo 5](#), Kubernetes es la plataforma de programación y orquestación graduada de la Cloud Native Computing Foundation (CNCF).

Los orquestadores de contenedores generalmente proporcionan la siguiente funcionalidad:

- Escalado automático de instancias de clúster en función de la carga de instancias
- Aprovisionamiento e implementación de instancias
- Funcionalidad básica de supervisión
- Detección de servicios

Otros orquestadores de contenedores de la CNCF incluyen:

- Azure Kubernetes Service (AKS)
- Azure Service Fabric
- Amazon Elastic Container Service (ECS)
- Docker Swarm
- Apache Mesos
- HashiCorp Nomad

Los buenos orquestadores de contenedores facilitan la creación y el mantenimiento de clústeres de informática. En el momento de escribir este libro, no existe una configuración estándar para los orquestadores de contenedores.

## Software de contenedores

Los demonios de contenedores, ubicados justo bajo los orquestadores, proporcionan el software para ejecutar un demonio. Según el orquestador que use, esto puede ser transparente para usted. Básicamente, los demonios de contenedor administran el ciclo de

vida del tiempo de ejecución del contenedor. En muchos casos, esto será tan simple como iniciar o detener el contenedor o más complicado, como interactuar con complementos como Cilium (que analizaremos en el [Capítulo 8](#)).

Las plataformas de software de contenedores comunes incluyen:

- Docker
- Agente Mesos (Mesos)
- Kubelet (Kubernetes)
- LXD (LXC)
- Rkt

Como abordaremos más adelante en este capítulo, Docker le permite tomar una imagen y, a continuación, crear instancias de contenedores a partir de ella, esencialmente brindando la interfaz de usuario como una acción no orquestada. Los agentes de Mesos o kubelets son ligeramente diferentes, ya que los generan las CLI o API de REST.

## Tiempo de ejecución de contenedores

Los tiempos de ejecución de contenedores proporcionan una interfaz con el fin de ejecutar instancias de contenedores. En la industria, hay dos categorías estándares de interfaces:

- Tiempo de ejecución del contenedor (runtime-spec)
- Especificación de imagen (image-spec)

El tiempo de ejecución del contenedor especifica la configuración del contenedor (por ejemplo, capacidades, montajes y configuración de red). La especificación de imagen contiene información sobre el diseño del sistema de archivos y el contenido de la imagen.

Si bien cada demonio u orquestador de contenedor tiene su propia especificación de configuración, el tiempo de ejecución del contenedor es un área que tiene estandarización en la imagen.

Tres de los tiempos de ejecución de contenedores más destacados son los siguientes:

- Containerd
- CRI-O
- Docker (hasta Kubernetes v1.2)

### Containerd

Containerd es un tiempo de ejecución de contenedor compatible con OCI que se usa en Docker. Containerd actúa como una capa de abstracción entre todas las syscalls de Linux que hacen que un contenedor opere y la configuración estándar de OCI que configura un contenedor. Containerd tiene un subsistema de eventos que permite que otros sistemas,

como etcd, se suscriban a los cambios del sistema y actúen en consecuencia. En especial, containerd es compatible con los formatos de imagen OCI y Docker. "[What is containerd?](#)" es una excelente entrada de blog con más información.

## CRI-O

CRI-O es una implementación de la [interfaz de tiempo de ejecución \(CRI\)](#) de contenedor de Kubernetes que habilita los plazos compatibles con OCI. Se considera que es una alternativa ligera al uso de Docker y containerd. CRI-O también admite la ejecución de contenedores Kata (un contenedor similar a una máquina virtual creado por VMWare) y es extensible a cualquier otro tiempo de ejecución compatible con OCI. Esto hace que sea un tiempo de ejecución atractivo para diversos casos de uso.

## Docker

Docker, que no debe confundirse con la plataforma completa de Docker, en efecto contiene un motor de tiempo de ejecución que, paradójicamente, es en esencia containerd. Kubernetes aceptó Docker como un tiempo de ejecución de contenedor [hasta la versión v1.20 de Kubernetes](#).

## Contenedores

La instancia de contenedor es la implementación y operación de software tal como se define en la especificación de un contenedor. El contenedor ejecuta el software definido (por lo general, proporcionado por una imagen) que contiene una serie de configuraciones y límites.

## Sistema operativo

El sistema operativo es la última de las construcciones, que es donde se ejecutan nuestros contenedores. En la mayoría de los tiempos de ejecución de contenedores, el kernel se comparte entre todas las instancias del contenedor.

# Especificación de Open Container Initiative (OCI)

Como se mencionó antes, en un principio los contenedores no eran un concepto bien definido. La [Open Container Initiative](#) (OCI) se estableció en 2015 para formalizar las especificaciones de [tiempo de ejecución](#) e [imagen](#) de las que hablamos anteriormente.

La configuración de tiempo de ejecución de un contenedor se especifica en un archivo *config.json* que usa el tiempo de ejecución del contenedor para configurar los parámetros operativos del contenedor. El uso de un contenedor compatible con OCI permite que la misma imagen/configuración de contenedor se ejecute en varios orquestadores de contenedores (por ejemplo, Docker y rkt) sin necesidad de modificar nada.

## Especificación de imagen de OCI

La [especificación de imagen de OCI](#) describe cómo se definen las capas del contenedor. Algunas de las propiedades de la imagen incluyen las siguientes:

- Autor
- Arquitectura y sistema operativo
- Usuario/grupo para ejecutar el contenedor
- Puertos expuestos fuera del contenedor
- Variables de entorno
- Puntos de entrada y comandos
- Directorio de trabajo
- Etiquetas de imagen

El siguiente es un ejemplo de una imagen de contenedor:

```
{  
    "created": "2021-01-31T22:22:56.015925234Z",  
    "author": "Michael Kehoe <michaelk@example.com>",  
    "architecture": "x86_64",  
    "os": "linux",  
    "config": {  
        "User": "alice",  
        "ExposedPorts": {  
            "5000/tcp": {}  
        },  
        "Env": [  
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",  
        ],  
        "Entrypoint": [  
            "/usr/bin/python"  
        ],  
        "Cmd": [  
            "app.py"  
        ],  
        "Volumes": {  
            "/var/job-result-data": {},  
            "/var/log/my-app-logs": {}  
        },  
        "WorkingDir": "/app",  
        "Labels": {  
            "com.example.project.git.url": "https://example.com/project.git",  
            "com.example.project.git.commit": "45a939b2999782a3f005621a8d0f29aa387e1d6b"  
        }  
    },  
    "rootfs": {  
        "diff_ids": [  
            "sha256:c6f988f4874bb0add23a778f753c65efe992244e148a1d2ec2a8b664fb66bbd1",  
            "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef"  
        ],  
        "type": "layers"  
    },  
}
```

```

"history": [
  {
    "created": "2021-01-31T22:22:54.690851953Z",
    "created_by": "/bin/sh -c #(nop) ADD file:a3bc1e842b69636f9df5256c49c5374fb4eef1e28
      1fe3f282c65fb853ee171c5 in /"
  },
  {
    "created": "2021-01-31T22:22:55.613815829Z",
    "created_by": "/bin/sh -c #(nop) CMD [\"sh\"]",
    "empty_layer": true
  }
]
}

```

## Especificación de tiempo de ejecución de OCI

En un nivel alto, la especificación de configuración de tiempo de ejecución brinda el control de los siguientes atributos de contenedor:

- Sistemas de archivos predeterminados
- Espacios de nombres
- Asignaciones de ID de usuario y ID de grupo
- Dispositivos
- Configuración del grupo de control
- Sysctl
- Seccomp
- Montaje de rootfs
- Rutas enmascaradas y de solo lectura
- Montaje de etiquetas de SELinux
- Comandos de enganche
- Otros atributos especializados (por ejemplo, personalidad, unificado e Intel RDT)

La especificación de OCI también **aborda** cómo se define el estado de un contenedor y el ciclo de vida del contenedor. El estado del contenedor se puede consultar mediante una función `state` (en `containerd`, `in` y `runc`) y devuelve los siguientes atributos:

```

{
  "ociVersion": "0.2.0",
  "id": "oci-container1",
  "status": "running",
  "pid": 4422,
  "bundle": "/containers/redis",
  "annotations": {
    "myKey": "myValue"
  }
}

```

El **ciclo de vida del contenedor OCI** reúne todas las especificaciones de tiempo de ejecución, incluida la forma en que la especificación de tiempo de ejecución interactúa con la especificación de imagen. Define cómo se inicia y configura el contenedor y, a continuación, cómo interactúa con los enganches de inicio/detención de la imagen.

## Docker

**Docker** proporciona una plataforma como servicio (PaaS) que permite a los usuarios ejecutar aplicaciones en contenedores en un escritorio de manera simplificada y estandarizada. Docker ejecuta imágenes compatibles con OCI y, a menudo, es un medio para probar una implementación de código antes de implementarla en un entorno de producción. Además, la disponibilidad general de Docker del software de escritorio proporciona a los desarrolladores un mecanismo fácil para probar el empaquetado y la implementación de su código, lo que lo ha convertido en un sistema atractivo para los desarrolladores.

### Creación de su primera imagen de Docker

Las imágenes de Docker por sí solas son imágenes que no son compatibles con OCI (su estructura es diferente de la especificación OCI) que se ejecutan en Docker y en sistemas que admiten el tiempo de ejecución de Docker (containerd). Lo hermoso de las imágenes de Docker está en lo rápido y sencillo que es crear. Cada imagen de contenedor se define mediante un Dockerfile. Este archivo consta de propiedades y comandos para compilar la imagen, ejecutar la aplicación, exponerla a la red y realizar comprobaciones de estado. Define qué dependencias y archivos se copian en la imagen del contenedor, así como qué comandos se ejecutan para iniciar las aplicaciones dentro del contenedor. Este enfoque de creación de la imagen de Docker con un formato simple (imperativo) similar a un script de shell (ejecutando comando tras comando) era el favorito de los desarrolladores que se enfrentaban a una administración compleja de paquetes o a la agrupación de aplicaciones en RPM o DEB (o archivos comprimidos).

Docker ofrece una [referencia de configuración completa](#) que puede ayudarlo a crear su propio contenedor.



#### Creación de su propio contenedor

Una excelente forma de entender cómo funcionan los contenedores es crear su propio contenedor básico. Le recomendamos probar el [taller de rubber-docker](#). Presenta los componentes básicos de un contenedor y muestra cómo se implementan.

Si bien el Dockerfile permite configuraciones complejas, es excepcionalmente fácil crear una imagen en solo unas pocas líneas. En el siguiente ejemplo, ejecutaremos una aplicación web de Python Flask y expondremos el puerto 5000:

1. Cree un archivo *requirements.txt*:

```
Flask >2.0.0
```

2. Cree un archivo *app.py*:

```
from Flask import flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello, World"
```

3. Cree su Dockerfile, llamado *Dockerfile*:

```
FROM ubuntu:18.04          ①
COPY . /app                ②
WORKDIR /app               ③
RUN pip3 install -r requirements.txt      ④
ENTRYPOINT ["python3"]      ⑤
CMD ["app.py"]             ⑥
EXPOSE 5000                ⑦
```

- ➊ Tomamos una imagen base para usar. En este caso, vamos a usar Ubuntu 18.04 desde el registro de Docker Hub.
- ➋ Copiamos el directorio actual en un nuevo directorio de imágenes llamado *app*.
- ➌ Establecemos el directorio de trabajo actual en */app*.
- ➍ Usamos *pip* para instalar las dependencias definidas de la aplicación.
- ➎ Establecemos la aplicación entrypoint como *python3* (este es un procedimiento recomendado de Docker/Python).
- ➏ Iniciamos la aplicación mediante la ejecución de *app.py*.
- ➐ Exponemos los servicios del puerto 5000 fuera del contenedor.

En su totalidad, esta imagen de contenedor se copia sobre nuestra aplicación y luego instala los requisitos de la aplicación, se inicia y permite conexiones de red en el puerto 5000 (el puerto de Flask predeterminado).

Para permitir que los usuarios u otras aplicaciones accedan a su contenedor, el Dockerfile contiene una directiva, **EXPOSE**, que permite que la pila de red del contenedor escuche en el puerto especificado en el tiempo de ejecución. Analizaremos la conectividad de red en el [Capítulo 8](#).



Técnicamente, un Dockerfile es una mezcla de un tiempo de ejecución del contenedor y una especificación de imagen, ya que proporciona semántica sobre el contenido de la imagen, así como la forma en que se ejecuta (es decir, limitaciones de cgroup de la memoria o CPU).

Una vez que haya creado su Dockerfile, puede crear su imagen de contenedor con el comando `docker build`. Esto se realiza normalmente con la ruta de archivo actual (por ejemplo, `docker build`), pero también puede usar una URL (por ejemplo, `docker build https://github.com/abcd/docker-example`). Puede consultar una referencia completa sobre cómo usar `docker build` [en la documentación](#).

Debe crear una etiqueta para su imagen a fin de que pueda hacer referencia a ella de forma rápida y fácil en el futuro cuando cree la imagen. Para ello, agregue la opción `--tag` al compilar:

```
docker build --tag example-flask-container
```

Cuando ejecute `docker build`, verá un resultado similar al siguiente:

```
[internal] load build definition from Dockerfile
=> transferring dockerfile: 203B
[internal] load .dockerrcignore
=> transferring context: 2B
[internal] load metadata for docker.io/library/ubuntu:18.04
[1/6] FROM docker.io/library/ubuntu:18.04
[internal] load build context
=> transferring context: 953B
CACHED [2/6] WORKDIR /app
[3/6] COPY requirements.txt requirements.txt
[4/6] RUN pip3 install -r requirements.txt
[5/6] COPY .
[6/6] CMD [ "python3", "app.py" ]
exporting to image
=> exporting layers
=> writing image sha256:8cae92a8fb6d091ce687b71b31252056944b09760438905b726625831564c4c
=> naming to docker.io/library/example-flask-container
```

## Procedimientos recomendados para usar Docker

La [aplicación de doce factores](#) hace seguimiento a una serie de características de aplicaciones basadas en Docker y la nube. Esto significa que la aplicación debe poder implementarse, iniciarse, detenerse y destruirse sin causar problemas. Aunque hablaremos sobre las soluciones de almacenamiento de contenedores más adelante en este libro, debe intentar evitar la conservación de datos tanto como sea posible, a menos que esté leyendo datos de un volumen persistente o utilizando un servicio de datos nativos de la nube (consulte el [Capítulo 9](#) para obtener más información al respecto).

Hay otros procedimientos recomendados que debería seguir:

- Utilice las **opciones** de `docker build`. Estas opciones le permiten personalizar la imagen de Docker para agregar características, como controles de cgroup, etiquetas, configuraciones de red y mucho más.
- Utilice un archivo `.dockerignore`. Similar a `.gitignore`, el archivo `.dockerignore` garantizará que ciertos archivos o carpetas no se incluyan en la compilación de imágenes.
- Para reducir la complejidad, las dependencias, los tamaños de archivo y los tiempos de compilación, evite instalar paquetes adicionales o innecesarios.
- **Use compilaciones en varias etapas**. Si necesita instalar dependencias para compilar la aplicación, puede hacerlo durante el proceso de compilación de la imagen, pero copie *solo* el resultado en la imagen del contenedor. Esto reduce el tamaño de la compilación de la imagen.

Puede obtener más información sobre los procedimientos recomendados de Dockerfile en el [sitio web de Docker](#).

## Otras plataformas de contenedores

Si bien la plataforma de Docker y las imágenes de OCI siguen siendo líderes de la industria, de todas formas existe otras ofertas de contenedores disponibles.

### Contenedores Kata

Los contenedores Kata son contenedores compatibles con OCI que tienen como objetivo mejorar la seguridad. Los contenedores Kata tienen las siguientes características de contenedor ligero:

- Cgroups
- Espacios de nombres
- Filtros de capacidad
- Filtrado de seccomp
- Control de acceso obligatorio

Los contenedores Kata también poseen algunos conceptos de máquinas virtuales, en especial un kernel invitado independiente por contenedor y un mejor aislamiento de hardware. El proceso `kata-runtime` controla la creación de instancias del contenedor compatible con OCI. La diferencia más grande es que hay un kernel invitado independiente por contenedor en lugar de un kernel compartido entre las instancias de contenedor.

### LXC y LXD

**LXC (Linux Containers)** es uno de los primeros mecanismos de contenedores que se lanzaron. LXC usa principalmente las capacidades de cgroup y espacio de nombres del kernel, así como un conjunto estándar de API de biblioteca para controlar los contenedores.

**LXD** es el software de administración de contenedores y las herramientas que ayudan a orquestar las API de LXC. El demonio LXD proporciona una API de REST sobre un socket/red Unix que le permite usar las herramientas de línea de comandos proporcionadas o crear las suyas.

El ecosistema de LXC y LXD está más orientado hacia ejecutar infraestructura que el desarrollo de aplicaciones debido a la falta de herramientas de desarrollo e implementación.

## Registros de contenedores

Después de crear sus imágenes de contenedor, necesitará lugares para almacenarlas y publicarlas. Estos se conocen como *registros de contenedores*. La popularidad de los registros de contenedores ha aumentado debido a sus sencillos mecanismos de implementación y sus mecanismos de replicación integrados, que reemplazan las complicadas configuraciones de rsync necesarias para los repositorios de paquetes DEB y RPM.

Al implementar un contenedor, el sistema de implementación de contenedores descargará la imagen del registro a la máquina host. Básicamente, un registro de contenedor actúa como un servidor de archivos de imágenes del contenedor de alta disponibilidad (con algunas protecciones). Todas las plataformas de contenedores comunes le permiten configurar un repositorio público o privado como el origen de las imágenes de contenedor.

Si bien estas responsabilidades suenan razonablemente triviales, hay muchas características de los registros de contenedores que se pasan por alto y son partes importantes de las implementaciones de contenedores seguras.

Estas son algunas características:

### *Replicación de imágenes*

Replicar las imágenes en otros registros de contenedores (globalmente)

### *Autenticación*

Permitir solo a los usuarios autorizados el acceso al registro y su contenido

### *Control de acceso basado en roles*

Restringir quién puede modificar las imágenes y los datos dentro del registro

### *Análisis de vulnerabilidades*

Revisar las imágenes para buscar vulnerabilidades conocidas

### *Recolección de basura*

Eliminar de forma periódica las imágenes antiguas

### *Auditoría*

Suministro de información confiable sobre los cambios en el registro de contenedores con fines de seguridad de la información

Los productos nativos de la nube actuales incluyen (vea el [panorama de la CNCF](#) para obtener más información):

- Harbor

- Dragonfly
- Alibaba Container Registry
- Amazon Elastic Container Registry (Amazon ECR)
- Azure Registry
- Docker Registry
- Google Container Registry
- IBM Cloud Container Registry
- JFrog Artifactory
- Kraken
- Portus
- Quay

Muchos de estos sistemas están vinculados a proveedores específicos. Sin embargo, Harbor es uno de los sistemas de registro de contenedores de open source más populares, por lo que lo veremos con más detalle.

## Almacenamiento seguro de imágenes con Harbor

Harbor es la plataforma principal de registro de contenedores nativos de la nube de open source. Creada en 2016, Harbor almacena imágenes de contenedores y proporciona funcionalidad de análisis de firmas y seguridad. Además, Harbor admite la alta disponibilidad y la funcionalidad de replicación, así como una serie de funciones de seguridad, incluida la administración de usuarios, el control de acceso basado en roles y la auditoría de actividades. La ventaja de Harbor es que proporciona características de primer nivel, aunque es de open source, lo que hace que sea una gran opción de plataforma.

### Instalación de Harbor

Cuando usa Harbor, esencialmente se convierte en un componente crítico de las canalizaciones de compilación e implementación. Debe ejecutarlo en un host que cumpla con estas especificaciones recomendadas:

- CPU Quad-Core
- 8 GB de memoria
- Disco mínimo de 160 GB
- Ubuntu 18.04/CentOS7 o posterior



Le recomendamos evaluar el rendimiento de la configuración o unidades de disco para que coincida con el rendimiento que desea. Harbor es básicamente un servidor de archivos, y el rendimiento del disco afectará en gran medida sus tiempos de compilación e implementación.

Harbor admite varios métodos de instalación, incluidos Kubernetes y Helm. Como aún no hemos presentado Kubernetes, vamos a instalar Harbor manualmente.

Harbor tiene dos versiones (o tipos de instalaciones): un instalador en línea y un instalador sin conexión. El instalador en línea es de menor tamaño y se basa en una conexión a Internet para descargar la imagen completa de Docker Hub (directamente de Internet). El instalador sin conexión tiene un tamaño más grande y no requiere una conexión a Internet. Si su postura de seguridad es restrictiva, el instalador sin conexión puede ser una mejor opción.

Las versiones de Harbor aparecen en [GitHub](#).

## Comprobación del instalador

Se recomienda que verifique la integridad del paquete descargado. Tendrá que abrir el acceso a Internet saliente al puerto TCP/11371 para poder llegar al servidor de claves GPG. Para comprobar el instalador:

1. Descargue el archivo `*.asc` correspondiente para la versión del instalador.
2. Obtenga la clave pública para las versiones de Harbor:  
`gpg --keyserver https://keyserver.ubuntu.com --receive-keys 644FF454C0B4115C`
3. Compruebe el paquete mediante la ejecución de:

```
gpg -v --keyserver https://keyserver.ubuntu.com -verify <asc-file-name>
```

Debería ver `gpg: Good signature from "Harbor-sign (The key for signing Harbor build)` en el resultado del comando anterior.

## Configuración de Harbor para la instalación

Antes de configurar Harbor, debe habilitar HTTPS mediante la instalación de un certificado de capa de sockets seguros (SSL) a través de su propia entidad de certificación (CA) o a través de una CA pública, como Let's Encrypt. Puede encontrar más información sobre cómo hacerlo en la [documentación de Harbor](#).

Todos los parámetros de instalación de Harbor se definen en un archivo llamado `harbor.yml`. Los parámetros definidos en este archivo configuran Harbor para su primer uso o cuando se está reconfigurando a través del archivo `install.sh`. Puede encontrar los detalles sobre todas las opciones de configuración en [GitHub](#). Para empezar, utilizaremos los siguientes [parámetros requeridos](#):

```

hostname: container-registry.example.com

http:
  # port for http, default is 80. If https enabled, this port will redirect to https port
  port: 80

https:
  port: 443
  certificate: /your/certificate/path
  private_key: /your/private/key/path

harbor_admin_password: Harbor12345

# DB configuration
database:
  password: root123
  max_idle_conns: 50
  max_open_conns: 100

# The default data volume
data_volume: /data

clair:
  updaters_interval: 12

jobservice:
  max_job_workers: 10

notification:
  webhook_job_max_retry: 10

chart:
  absolute_url: disabled

log:
  level: info
  local:
    rotate_count: 50
    rotate_size: 200M
    location: /var/log/harbor

proxy:
  http_proxy:
  https_proxy:
  no_proxy: 127.0.0.1,localhost,.local,.internal,log,db,redis,nginx,core,portal, \
  postgresql,jobservice,registry,registryctl,clair

```

## Creación de una imagen de Packer

Con lo que aprendimos en el [Capítulo 2](#), vamos a crear una imagen de Packer para implementar Harbor. Puede usar el siguiente archivo `harbor.json`:

```
{
  "builders": [
    {
      "type": "azure-arm",
      "client_id": "<place-your-client_id-here>",
      "client_secret": "<place-your-client_secret-here>",
      "tenant_id": "<place-your-tenant-id-here>",
      "image_publisher": "microsoft-oss",
      "image_offer": "ubuntu-server",
      "image_sku": "16.04-lts-v2"
    }
  ]
}
```

```

"subscription_id": "<place-your-subscription-here>",

"managed_image_resource_group_name": "CloudNativeAzure-group",
"managed_image_name": "harborImage",

"os_type": "Linux",
"image_publisher": "Canonical",
"image_offer": "UbuntuServer",
"image_sku": "16.04-LTS",

"azure_tags": {
    "env": "Production",
    "task": "Image deployment"
},

"location": "East US",
"vm_size": "Standard_DS1_v2"
}],
"provisioners": [
    "execute_command": "chmod +x {{ .Path }}; {{ .Vars }} sudo -E sh '{{ .Path }}'",

    "inline": [
        "apt-get update",
        "apt-get upgrade -y",
        "/usr/sbin/waagent -force -deprovision+user && export HISTSIZE=0 && sync"
    ],
    "inline_shbang": "/bin/sh -x",
    "type": "shell"
},
{
{
    "type": "shell",
    "inline": [
        "wget https://github.com/goharbor/harbor/releases/download/v2.1.3/ \
        harbor-online-installer-v2.1.3.tgz -O /tmp/harbor-online-installer-v2.1.3.tgz",
        "tar -xvf /tmp/harbor-online-installer-v2.1.3.tgz"
    ]
},
{
    "type": "file",
    "source": "{{template_dir}}/harbor.yml",
    "destination": "/tmp/harbor"
},
{
    "type": "shell",
    "inline": [
        "sudo chmod 0777 /tmp/harbor/install.sh",
        "sudo /tmp/harbor/install.sh"
    ]
}
]
}

```

Cuando ejecute `packer build harbor.json`, su imagen se creará con Harbor instalado y configurado. Si usa HTTPS (y debe hacerlo), deberá modificar el archivo de Packer para incluir la copiar los archivos de la clave privada y el certificado.

# Almacenamiento seguro de imágenes con Azure Container Registry

Azure ofrece su propio registro de contenedores, conocido como Azure Container Registry. Dado que Azure proporciona el registro, la configuración inicial y los requisitos de mantenimiento continuos consumen mucho menos tiempo que cuando se ejecuta software, como Harbor.

## Instalación de Azure Container Registry

Implementar una instancia de contenedor es extraordinariamente simple:

1. Inicie sesión en el portal de Azure y haga clic en "Create a resource" (Crear un recurso), como se muestra en la [Figura 3-3](#).

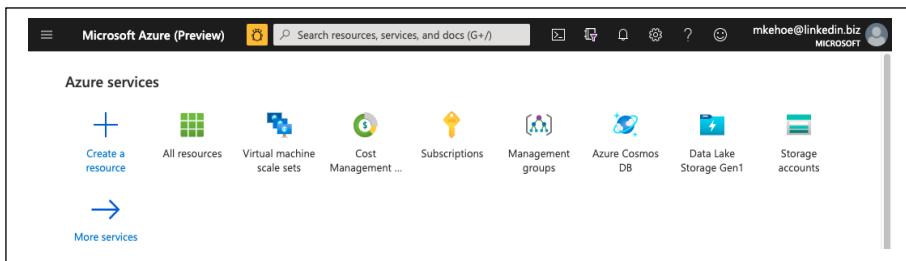


Figura 3-3. Página principal del portal de Azure

2. Busque "container registry" (registro de contenedores) en la barra de búsqueda y haga clic en la oferta de Container Registry de Microsoft (consulte la [Figura 3-4](#)).

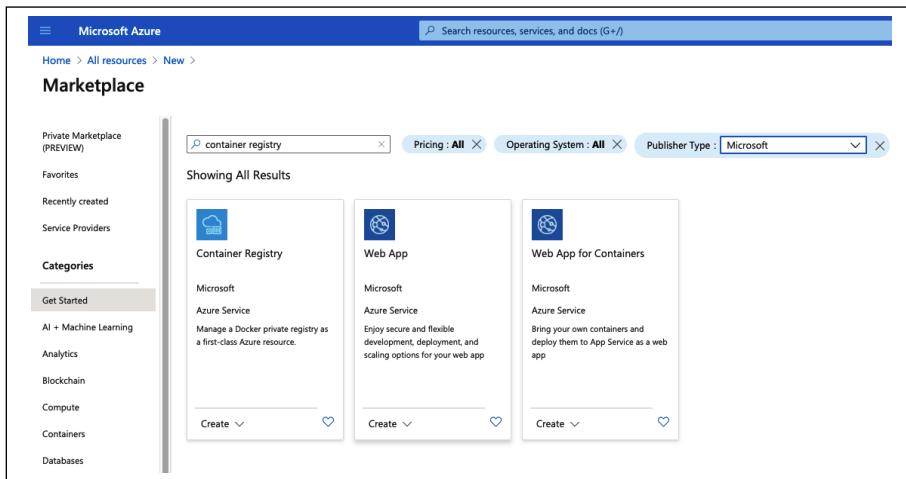


Figura 3-4. Búsqueda de Azure Container Registry

Verá un formulario para llenar con la información básica sobre el registro que va a crear ([Figura 3-5](#)).

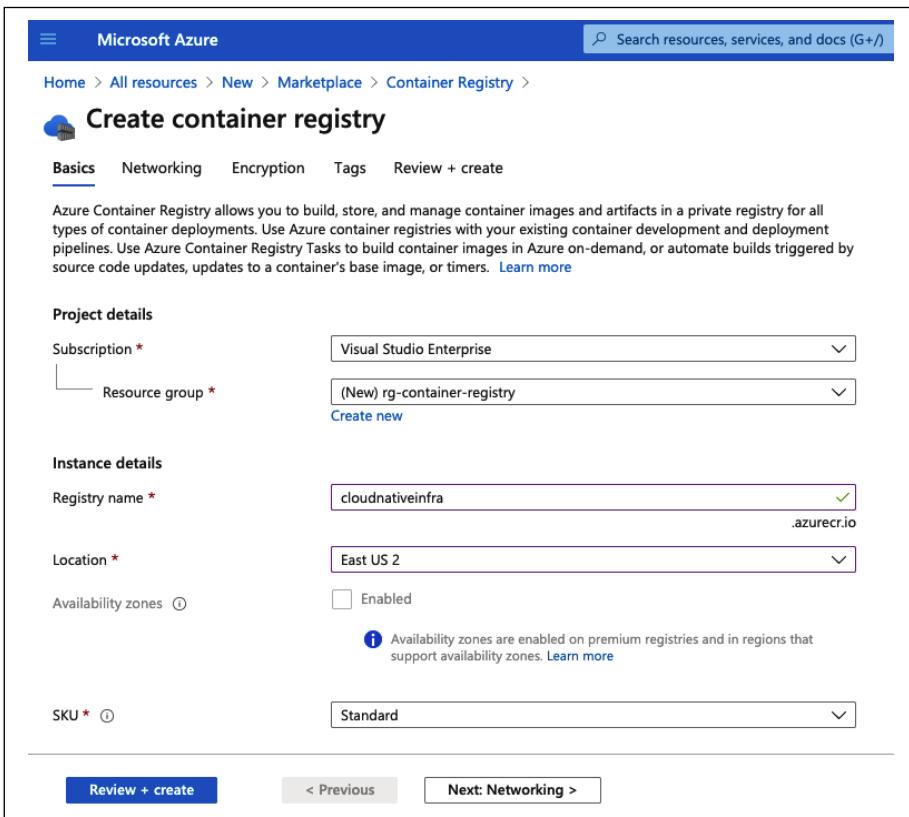


Figura 3-5. Primera página para crear un registro de contenedores

3. Ingrese los siguientes detalles básicos del registro de contenedores:

#### *Suscripción*

Elija la suscripción en la que desea realizar la implementación.

#### *Grupo de recursos*

Puede usar un grupo de recursos existente o crear uno nuevo. En este caso, vamos a crear uno nuevo, llamado **rg-container-registry**.

#### *Nombre del registro*

Esto configurará la URL utilizada para acceder al registro. Usaremos **cloudnativeinfra.azurecr.io**. Esto es único a nivel global, por lo que tendrá que elegir el suyo y asegurarse de que no reciba un mensaje de error que indique que ya se ha ocupado.

#### *Región*

Elija su región. Vamos a implementar en la región Este de EE. UU. 2.

## SKU

Hay tres niveles disponibles. Puede ver una comparación en la [página de precios de Container Registry](#). Vamos a usar el nivel Estándar.

4. Elija la configuración de red ([Figura 3-6](#)). Dado que estamos usando el nivel Estándar, no podemos hacer ningún cambio de configuración aquí. Por lo tanto, el registro de contenedores que se creará estará disponible a través de Internet y no solo dentro de nuestra red de Azure.

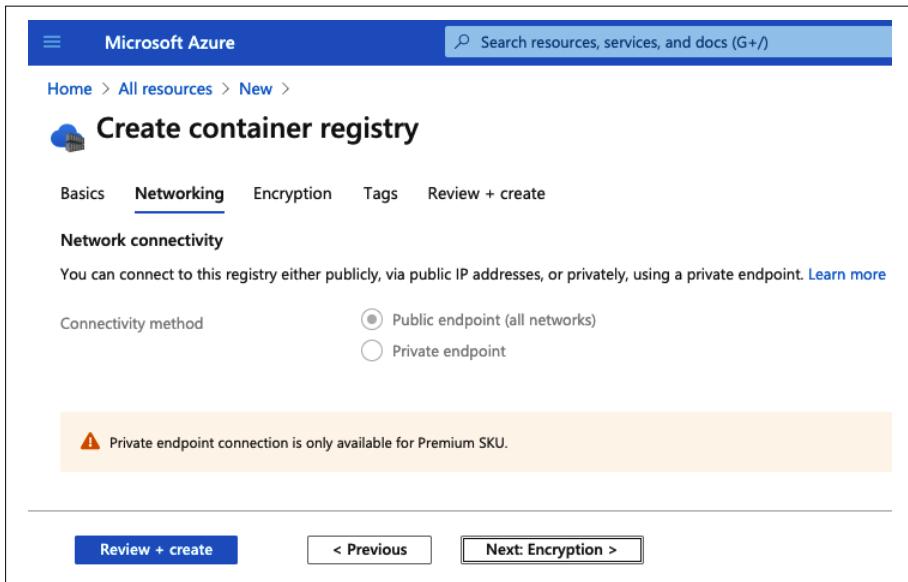


Figura 3-6. Configuración de red del registro de contenedores

5. Elija la configuración de cifrado ([Figura 3-7](#)). El nivel Estándar proporciona cifrado en reposo para los datos almacenados, mientras que el nivel Premium permite a los usuarios establecer sus propias claves de cifrado.

Figura 3-7. Cifrado del registro de red de contenedores

- Haga clic en "Review + create" (Revisar y crear) para crear su registro de contenedores. Esto debería implementarse en menos de dos minutos.



### Puntos de conexión privados

Los puntos de conexión privados son un concepto de Azure que permite que los recursos de la nube estén disponibles a través del espacio de direcciones RFC1918 en lugar de una IP pública. Esto significa que su recurso solo está disponible a través de una red conectada localmente y no a través de Internet.

Si desea implementar su registro de contenedores a través de Terraform, el siguiente código le permitirá hacerlo:

```
resource "azurerm_resource_group" "rg" {
    name = "example-resources"
    location = "West Europe"
}

resource "azurerm_container_registry" "acr" {
    name = "containerRegistry1"
    resource_group_name = azurerm_resource_group.rg.name
    location = azurerm_resource_group.rg.location
    sku = "Premium"

    identity {
        type = "UserAssigned"
        identity_ids = [ azurerm_user_assigned_identity.example.id ]
    }
}
```

```

    encryption {
        enabled = true
        key_vault_key_id = data.azurerm_key_vault_key.example.id
        identity_client_id = azurerm_user_assigned_identity.example.client_id
    }
}

resource "azurerm_user_assigned_identity" "example" {
    resource_group_name = azurerm_resource_group.example.name
    location = azurerm_resource_group.example.location name = "registry-uai"
}

data "azurerm_key_vault_key" "example" {
    name = "super-secret"
    key_vault_id = data.azurerm_key_vault.existing.id
}

```

## Almacenamiento de imágenes de Docker en un registro

Como vimos anteriormente, debe poder almacenar y publicar imágenes de contenedores de forma confiable. Una vez que haya creado su imagen personalizada, debe insertarla en el registro. Además, y como analizamos antes, hay muchas opciones para almacenar imágenes, incluido Docker Registry y los registros de contenedores de los diversos proveedores de nube (por ejemplo, Azure Container Registry). En este caso, vamos a usar el registro de Harbor que creamos anteriormente en este capítulo.

En primer lugar, accederemos al registro:

```
$ docker login <harbor_address>
```

Por ejemplo:

```
$ docker login https://myharborinstallation.com
```

Ahora etiquetaremos la imagen:

```
$ docker tag example-flask-container <harbor_address>/demo/example-flask-container
```

Y ahora la insertamos:

```
$ docker push <harbor_address>/demo/example-flask-container
```

Una vez que esto finalice, Harbor podrá publicar nuestra imagen en cualquier software de contenedor que la descargue.

# Ejecución de Docker en Azure

En Azure, hay dos maneras en que puede ejecutar su imagen de contenedor de Docker: con Azure Container Instances o ejecutando su propia máquina virtual con Docker instalado. En esta sección, mostraremos ambos métodos.

## Azure Container Instances

Azure Container Instances (ACI) le permite poner en marcha rápidamente una instancia de contenedor en Azure sin tener que administrar la infraestructura subyacente. Esto proporciona un servicio similar a Docker, pero en la nube pública. ACI permite la implementación de imágenes de Azure Container Registry o imágenes de Docker desde repositorios privados o públicos.

ACI también permite que estas instancias se coloquen en redes privadas o públicas, en función del uso de las imágenes del contenedor.

Estas son algunas características adicionales de ACI:

- No es necesario configurar una especificación de tiempo de ejecución de OCI
- La capacidad de pasar variables del entorno al contenedor
- Administración básica del firewall
- Directivas de reinicio de contenedores administradas

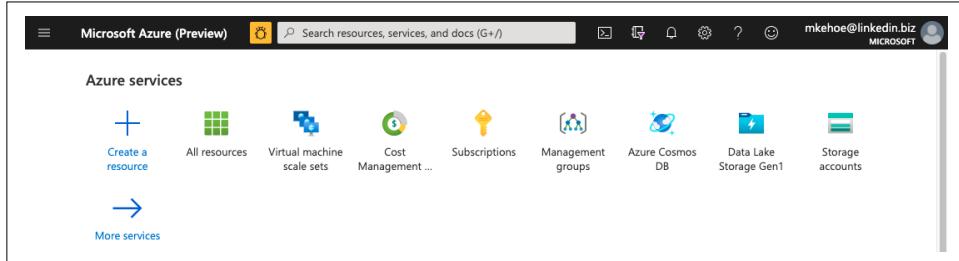
En pocas palabras, ACI es la forma más fácil de empezar a usar contenedores en Azure. ACI también le ofrece la posibilidad de probar nuevas imágenes de contenedores sin crear un entorno totalmente independiente.

Debe tener en cuenta algunas salvedades cuando ejecuta una ACI:

- Hay un límite de quad-core/16 GB para cualquier instancia de contenedor individual.
- La configuración de ACI para la CPU y memoria anulará cualquier ajuste en el tiempo de ejecución que haya aplicado a la imagen del contenedor.

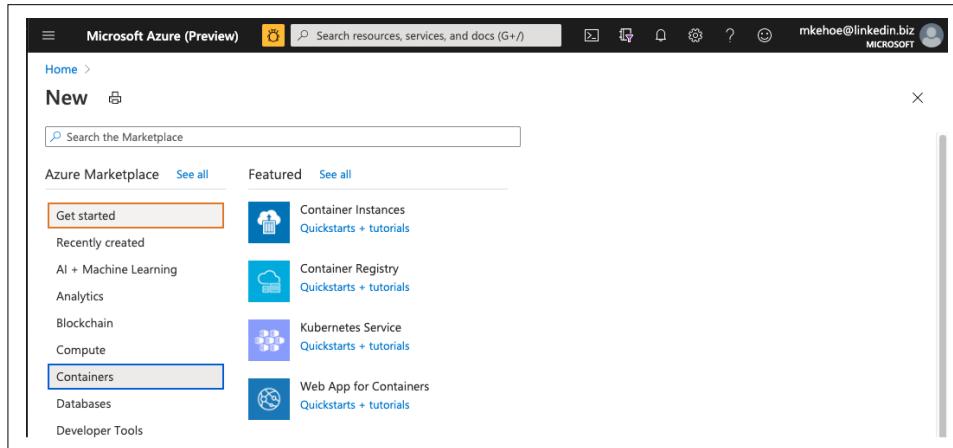
## Implementación de una Azure Container Instance

Implementar una instancia de contenedor de Azure es extraordinariamente simple. Primero, inicie sesión en el portal de Azure y, luego, haga clic en "Create a resource" (Crear un recurso), como se muestra en la [Figura 3-8](#).



*Figura 3-8. Página principal del portal de Azure*

Baje hasta llegar a "Containers" en el lado izquierdo y verá una oferta de productos relacionados con contenedores (consulte la [Figura 3-9](#)). Haga clic en el vínculo de instancias de contenedores.



*Figura 3-9. Búsqueda de productos de contenedores*

Aparecerá un formulario para llenar con la información básica sobre la instancia de contenedor que va a crear ([Figura 3-10](#)).

The screenshot shows the 'Create container instance' wizard in the Azure portal. The 'Basics' tab is selected. The page header includes a search bar and icons for refresh, help, and notifications. The main content area is titled 'Create container instance'.

**Project details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

**Subscription \***: Visual Studio Enterprise

**Resource group \***: (dropdown menu) | Create new

**Container details**

**Container name \***: (input field)

**Region \***: (US) East US

**Image source \***: Quickstart images (selected), Azure Container Registry, Docker Hub or other registry

**Image \***: mcr.microsoft.com/azuredocs/aci-helloworld:latest (Linux)

**Size \***: 1 vcpu, 1.5 GiB memory, 0 gpus | Change size

At the bottom are buttons for 'Review + create', '< Previous', and 'Next : Networking >'.

Figura 3-10. Configuración básica para una instancia de contenedor de Azure

Rellene los siguientes campos:

#### Suscripción

Elija la suscripción en la que desea realizar la implementación.

#### Grupo de recursos

Puede usar un grupo de recursos existente o crear uno nuevo. En este caso, vamos a crear uno nuevo, llamado `rg-aci-test`.

#### Nombre del contenedor

Asigne un nombre a la instancia del contenedor. La nuestra se llamará `aci-demo`.

#### Región

Elija su región. Vamos a implementar en la región Este de EE. UU.

## Origen de la imagen

Aquí es donde elige la imagen del contenedor. Se le presentan tres opciones:

- Usar una imagen de inicio rápido de demostración que proporciona Azure.
- Usar una imagen que se haya cargado en Azure Container Registry.
- Usar una imagen que se haya cargado en Docker Hub u otro registro. Si ha implementado Harbor, puede configurarlo aquí. En este caso, vamos a elegir la **imagen de inicio rápido de Nginx**.

## Tamaño

Aquí elige qué tipo de máquina virtual desea usar. Tenga en cuenta que cuanto más grande sea la máquina virtual, más tendrá que pagar para ejecutarla. En este caso, vamos a ejecutar una configuración de "1 vCPU, memoria de 1,5 GiB de memoria, 0 GPU". Esto significa que tenemos un núcleo virtual, 1 GB de memoria y ninguna tarjeta gráfica.

Una vez que esto se haya configurado, haga clic en el botón "Next: Networking" (Siguiente: Redes) para configurarla red (**Figura 3-11**).

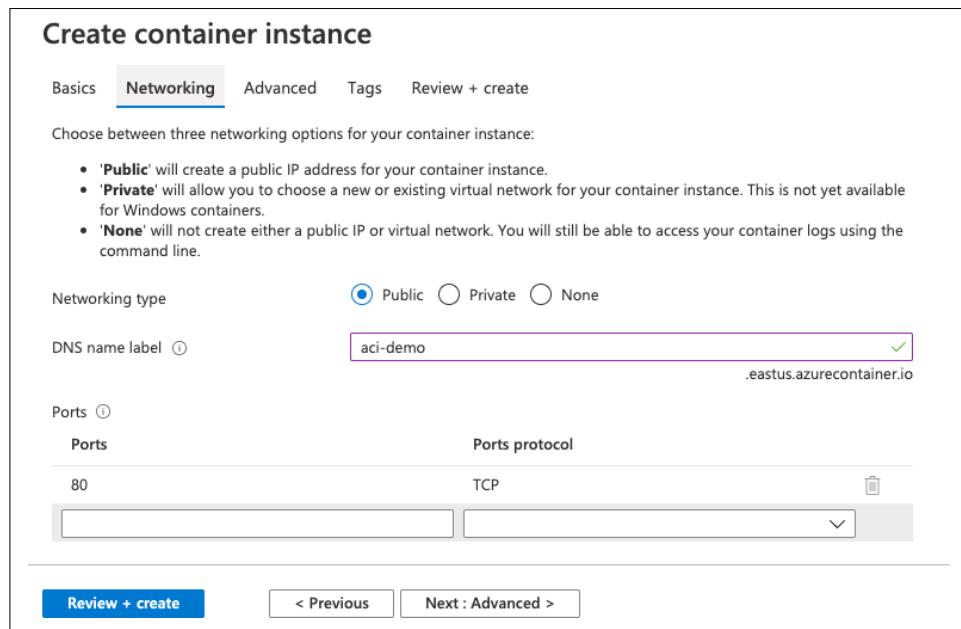


Figura 3-11. Configuración de red para una instancia de contenedor de Azure

En esta demostración, vamos a asignar una IP pública a nuestra instancia de contenedor (para que podamos acceder a ella desde Internet) y, a continuación, dar a la dirección IP el nombre DNS de *aci-demo.eastus.azurecontainer.io* (le recomendamos usar un nombre único para esto). A continuación, tenemos la opción de hacer que los puertos TCP o UDP estén disponibles desde fuera del contenedor.

Como estamos usando un contenedor Nginx, se abre el puerto 80 automáticamente. También puede crear una red virtual (o usar una existente) y una subred si está utilizando direcciones IP privadas.

Si hace clic en el botón "Next: Advanced" (Siguiente: Opciones avanzadas), verá que hay algunas otros parámetros de configuración que puede aplicar para hacer frente a las directivas de error y las variables del entorno. Para esta instancia, no vamos a realizar ningún cambio ([Figura 3-12](#)).

**Create container instance**

Basics Networking **Advanced** Tags Review + create

Configure additional container properties and variables.

Restart policy ⓘ On failure

Environment variables

Mark as secure	Key	Value
No		

Command override ⓘ [ ] Example: [ "/bin/bash", "-c", "echo hello; sleep 100000" ]

**Review + create** < Previous Next : Tags >

*Figura 3-12. Configuración avanzada para una instancia de contenedor de Azure*

Finalmente, haga clic en "Review + create" (Revisar y crear) y se creará la imagen del contenedor. Tomará alrededor de dos minutos que se implemente una imagen de inicio rápido.

Una vez que se implemente su contenedor, podrá ir al nombre DNS de su contenedor en un navegador de Internet y ver el banner "Welcome to nginx!" (consulte la [Figura 3-13](#)). A través del portal de Azure, también puede obtener una conexión de la consola a la instancia.



*Figura 3-13. Página de bienvenida predeterminada de Nginx*

Puede obtener más información sobre la implementación de ACI en la [documentación de Microsoft](#).

## Ejecución de un motor de contenedores de Docker

Si desea tener un control más preciso de su entorno del contenedor, una opción es ejecutar su propio motor de contenedores (CE) de Docker. La contrapartida es que tiene que administrar la máquina virtual subyacente en la que se ejecuta Docker.

1. Instale Docker CE en una máquina virtual de Azure:

Centos/Red Hat:

```
$ sudo yum install -y yum-utils  
  
$ sudo yum-config-manager \  
    --add-repo \  
    https://download.docker.com/linux/centos/docker-ce.repo  
$ sudo yum install docker-ce docker-ce-cli containerd.io
```

Debian/Ubuntu:

```
$ sudo add-apt-repository \  
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \  
    $(lsb_release -cs) \  
    stable"  
  
$ sudo apt-get update  
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Otra opción puede ser instalar Docker CE a través de una imagen de Packer. A continuación, puede ver un ejemplo de la configuración de Packer:

```
{  
  "builders": [{  
    "type": "azure-arm",  
    "client_id": "<place-your-client_id-here>",  
    "client_secret": "<place-your-client_secret-here>",  
    "tenant_id": "<place-your-tenant-id-here>",  
    "subscription_id": "<place-your-subscription-here>",  
    "managed_image_resource_group_name": "CloudNativeAzure-group",  
    "managed_image_name": "DockerCEEngine",  
  
    "os_type": "Linux",  
    "image_publisher": "Canonical",  
    "image_offer": "UbuntuServer",  
    "image_sku": "16.04-LTS",  
  
    "azure_tags": {  
      "env": "Production",  
      "task": "Image deployment"  
    },  
  
    "location": "East US",  
    "vm_size": "Standard_DS1_v2"  
  }],  
  "provisioners": [{  
    "execute_command": "chmod +x {{ .Path }}; {{ .Vars }} sudo -E sh '{{ .Path }}'",  
    "inline": [  
      "apt-get update",  
      "apt-get upgrade -y",  
      "/usr/sbin/waagent -force -deprovision+user && export HISTSIZE=0 && sync"  
    ]  
  }]
```

```

        ],
        "inline_shebang": "/bin/sh -x",
        "type": "shell"
    },
    {
        "type": "shell",
        "inline": [
            "sudo apt-get remove docker docker-engine",
            "sudo apt-get install apt-transport-https ca-certificates curl \
                software-properties-common",
            "curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add \
                -sudo apt-key fingerprint 0EBFCD88",
            "sudo add-apt-repository \"deb [arch=amd64] https://download.docker.com/ \
                linux/ubuntu      $(lsb_release -cs) stable\"",
            "sudo apt-get update",
            "sudo apt-get -y upgrade",
            "sudo apt-get install -y docker-ce",

            "sudo groupadd docker",
            "sudo usermod -aG docker ubuntu",
            "sudo systemctl enable docker"
        ]
    }
}

```

2. Una vez que Docker CE se esté ejecutando, inicie su contenedor ejecutando lo siguiente:

```
$ docker run -d <container-name>
```

3. Si usamos nuestro ejemplo de Flask anterior, podría iniciar su contenedor ejecutando:

```
$ docker run -d -p 5000:5000 http://myharborinstallation.com/demo/ \
example-flask-container
```

4. Si ejecuta `curl localhost:5000/`, verá que le devuelve "Hello, world".

## Resumen

En este capítulo, analizamos brevemente las capas de abstracción que componen el ecosistema de contenedores. Como el ecosistema de contenedores ahora está firmemente centrado en la especificación OCI y Kubernetes, está prosperando y vemos una proliferación de productos de seguridad y red que se basan en la norma OCI. En el [Capítulo 8](#), exploraremos otro estándar de contenedor, la Interfaz de red de contenedores (CNI), y veremos cómo aprovecharla para la configuración de red de contenedores.

## CAPÍTULO 4

# Kubernetes: El gran orquestador

A medida que las aplicaciones monolíticas se desglosaron en microservicios, los contenedores se convirtieron en el hogar de facto para estos microservicios. Los microservicios son un enfoque arquitectónico nativo de la nube en el que una sola aplicación se compone de muchos componentes o servicios más pequeños, de acoplamiento flexible e independientes que se pueden implementar. Los contenedores aseguran que el software se ejecute correctamente cuando se mueva entre diferentes entornos. A través de los contenedores, los microservicios trabajan al unísono con otros microservicios para formar una aplicación completamente funcional.

Si bien el desglose de las aplicaciones monolíticas en servicios más pequeños resuelve un problema, crea problemas más grandes con respecto a la administración y el mantenimiento de una aplicación sin tiempo de inactividad significativo, la creación de redes de los diversos microservicios, el almacenamiento distribuido, y así sucesivamente. Los contenedores ayudan al desacoplar las aplicaciones en bases de código más pequeñas que se mueven rápidamente con un enfoque en el desarrollo de características. Sin embargo, aunque este desacoplamiento es limpio y fácil al principio porque hay menos contenedores que administrar, a medida que aumenta la cantidad de microservicios en una aplicación, se hace casi imposible depurar, actualizar o incluso implementar los microservicios contenedorizados de forma segura en la pila sin romper algo o generar tiempo de inactividad.

La contenedorización de aplicaciones fue el primer gran paso hacia la creación de entornos que se reparan automáticamente con cero tiempo de inactividad, pero esta práctica necesitaba evolucionar más, especialmente en términos de desarrollo de software y entrega en entornos nativos de la nube. Esto condujo al desarrollo de programadores y motores de orquestadores como Mesos, Docker Swarm, Nomad y Kubernetes. En este capítulo y en el siguiente, nos centraremos en Kubernetes debido a su madurez y a que ha sido ampliamente adoptado en la industria.

Google presentó Kubernetes al mundo en 2014, después de dedicar casi una década a perfeccionar y aprender de su sistema de administración de clústeres interno, *Borg*. En palabras sencillas, Kubernetes es un sistema de orquestación de contenedores de open source. El término *orquestación de contenedores* se refiere principalmente al ciclo de vida completo de la administración de contenedores en entornos dinámicos,

como la nube, donde las máquinas (servidores) van y vienen según sea necesario. Un orquestador de contenedores automatiza y administra una variedad de tareas, como el aprovisionamiento y la implementación, la programación, la asignación de recursos, el escalado, el equilibrio de carga y la supervisión del estado del contenedor. En 2015, Google donó Kubernetes a la Cloud Native Computing Foundation (CNCF).

Kubernetes (también conocido como K8s) es una de las piezas más adoptadas de una solución nativa de la nube ya que se encarga de la implementación, la escalabilidad y el mantenimiento de aplicaciones contenedorizadas. Kubernetes ayuda a los ingenieros de confiabilidad del sitio (SRE) y a los ingenieros de DevOps a ejecutar su carga de trabajo en la nube con resiliencia mientras se encarga del escalado y la conmutación por error de aplicaciones que abarcan varios contenedores en clústeres.



### ¿Por qué se le llama K8s a Kubernetes?

Kubernetes es una palabra derivada del griego que significa *timonel* o *capitán del barco*. Kubernetes también se llama K8s. Este numerónimo se creó al reemplazar las ocho letras entre K y S ("ubernete") con un 8.

A continuación, presentamos algunas de las características clave que Kubernetes proporciona desde el inicio:

#### *Recuperación automática*

Una de las características más destacadas de Kubernetes es el proceso de poner en marcha nuevos contenedores cuando un contenedor falla.

#### *Detección de servicios*

En un entorno nativo de la nube, los contenedores se mueven de un host a otro. El proceso de averiguar realmente cómo conectarse a un servicio o una aplicación que se ejecuta en un contenedor se denomina *detección de servicios*. Kubernetes expone los contenedores automáticamente mediante DNS o la dirección IP de los contenedores.

#### *Equilibrio de carga*

Para mantener la aplicación implementada en un estado estable, Kubernetes equilibra la carga y distribuye el tráfico entrante de forma automática.

#### *Implementaciones automáticas*

Kubernetes funciona en una sintaxis declarativa, lo que significa que no tiene que preocuparse por cómo implementar las aplicaciones. Por el contrario, usted determina lo que debe implementarse y Kubernetes se encarga de ello.

#### *Empaquetado en contenedores*

Para aprovechar al máximo los recursos informáticos, Kubernetes implementa automáticamente los contenedores en el mejor host posible sin desperdiciar ni deteriorar la disponibilidad general de otros contenedores.

En este capítulo, profundizaremos en los componentes principales y en los conceptos subyacentes de Kubernetes. Aunque este capítulo no tiene como objetivo enseñar a usar Kubernetes por completo puesto que ya existe una amplia gama de libros<sup>1</sup> que abarcan el tema en profundidad, queremos crear una base sólida. Este enfoque práctico lo ayudará a comprender mejor lo esencial del entorno general. Comencemos hablando sobre cómo funciona el clúster de Kubernetes.

## Componentes de Kubernetes

El clúster de Kubernetes contiene dos tipos de componentes de nodo:

### Plano de control

Es el componente rector del clúster de Kubernetes. Garantiza que siempre se ejecuten un par de servicios importantes (como, programación, inicio de pods nuevos<sup>2</sup>). El objetivo principal del nodo del plano de control es asegurarse de que el clúster esté siempre en buen estado y funcionando correctamente.

### Nodos de trabajo

Estas son las instancias de procesamiento que ejecutan la carga de trabajo en el clúster de Kubernetes, que hospeda todos sus contenedores.

En la [Figura 4-1](#) se muestran los componentes de alto nivel de Kubernetes. Las líneas muestran las conexiones, como los nodos de trabajo que aceptan una conexión que habla con el equilibrador de carga, que distribuye el tráfico.

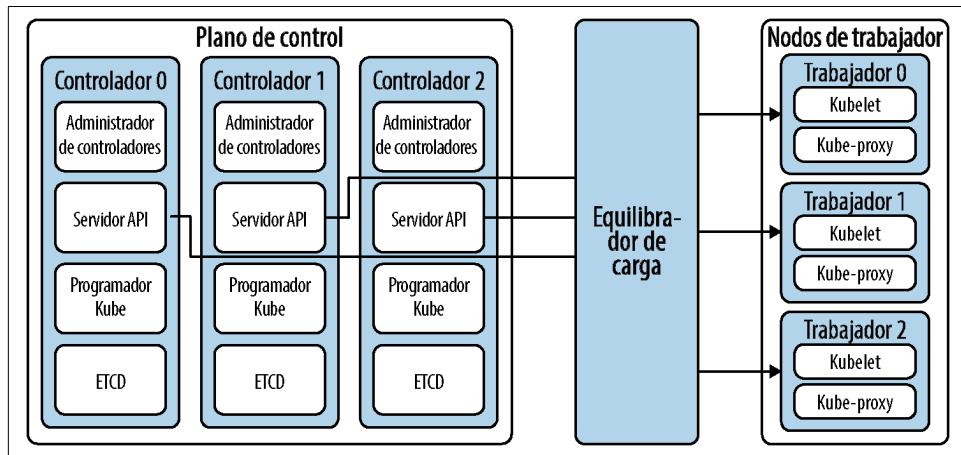


Figura 4-1. Componentes de Kubernetes

1 *Managing Kubernetes* de Brendan Burns y Craig Tracey (O'Reilly, 2019); *Kubernetes in Action* de Marko Lukša (Manning, 2018).

2 Los pods son las unidades de informática más pequeñas que se pueden implementar y que puede crear y administrar en Kubernetes.

Ahora veamos con detalle cada componente.

## Plano de control

El plano de control es principalmente responsable de tomar decisiones globales sobre el clúster de Kubernetes, como detectar el estado del clúster, programar pods en los nodos y administrar el ciclo de vida de los pods. El plano de control de Kubernetes tiene varios componentes, que describimos en las siguientes subsecciones.

### kube-apiserver (servidor de API)

El servidor de API es el front-end del plano de control de Kubernetes y es el único componente con acceso directo a todo el clúster de Kubernetes. Como vimos en la [Figura 4-1](#), el servidor de API sirve como punto central para todas las interacciones entre los nodos de trabajo y los nodos del controlador. Los servicios que se ejecutan en un clúster de Kubernetes utilizan el servidor de API para comunicarse entre sí. Puede ejecutar varias instancias del servidor de API porque está diseñado para que se pueda escalar horizontalmente.

### Programador Kube

El programador Kube es responsable de determinar qué nodo de trabajo ejecutará un pod (o unidad de trabajo básica; explicaremos los pods con más detalle más adelante en este capítulo). El programador Kube se comunica con el servidor de API, que determina qué nodos de trabajo están disponibles para ejecutar el pod programado de la mejor manera posible. El programador busca pods recién creados que se tienen que asignar a un nodo y, a continuación, encuentra nodos *factibles* como candidatos potenciales y los califica en función de diferentes factores, como la capacidad de recursos y el requisito de hardware del nodo, para garantizar que se toma la decisión correcta para la programación. Se elige el nodo con la puntuación más alta para ejecutar el pod. El programador también notifica al servidor de API sobre esta decisión en un proceso que se conoce como *enlace*.

### Administrador de controladores Kube

Kubernetes tiene una característica central incorporada que implementa las capacidades de recuperación automática en el clúster. Esta característica se denomina administrador de controladores Kube y se ejecuta como un demonio. El administrador de controladores ejecuta un bucle de control llamado bucle de conciliación, que es un bucle de no terminación que es responsable de lo siguiente:

- Determinar si un nodo dejó de funcionar y, si es así, tomar medidas. Esto lo realiza el controlador del nodo.
- Mantener el número correcto de pods. Esto lo realiza el controlador de replicación.
- Unir los objetos del punto de conexión (es decir, servicios y pods). Esto lo realiza el controlador de punto de conexión.

- Garantizar que se creen cuentas y puntos de conexión predeterminados para los nuevos espacios de nombres. Esto lo realizan los controladores de cuenta de servicio y tokens.

El bucle de conciliación es la fuerza impulsora tras la capacidad de recuperación automática de Kubernetes. Kubernetes determina el estado del clúster y sus objetos al ejecutar continuamente los siguientes pasos en un bucle:

1. Obtiene el estado declarado por el usuario (el *estado deseado*).
2. Observa el estado del clúster.
3. Compara los estados observados y deseados para hallar diferencias.
4. Toma medidas según el estado observado.

## etcd

Kubernetes usa etcd como almacén de datos. etcd es un almacén de clave-valor responsable de conservar todos los objetos de Kubernetes. Lo creó originalmente el equipo de CoreOS y ahora lo administra la CNCF. Por lo general, se pone en marcha en una configuración de alta disponibilidad y los nodos de etcd se hospedan en instancias independientes.

## Nodos de trabajo

Un clúster de Kubernetes contiene un conjunto de máquinas de trabajo denominadas nodos de trabajo que ejecutan las aplicaciones contenedorizadas. El plano de control administra los nodos de trabajo y los pods en el clúster. Algunos componentes se ejecutan en todos los nodos de trabajo de Kubernetes (los analizamos en las siguientes subsecciones).

### Kubelet

Kubelet es el agente demonio que se ejecuta en todos los nodos para asegurarse de que los contenedores siempre estén en estado de ejecución en un pod y en buen estado. Kubelet informa al servidor de API sobre los recursos actualmente disponibles (CPU, memoria, disco) en los nodos de trabajo para que el servidor de API pueda usar el administrador del controlador para observar el estado de los pods. Como kubelet es el agente que se ejecuta en los nodos de trabajo, estos manejan tareas básicas de limpieza, como reiniciar los contenedores si es necesario y realizar constantemente comprobaciones de estado.

### Kube-proxy

Kube-proxy es el componente de red que se ejecuta en cada nodo. Kube-proxy observa todos los servicios de Kubernetes<sup>3</sup> en el clúster y garantiza que cuando se realice una

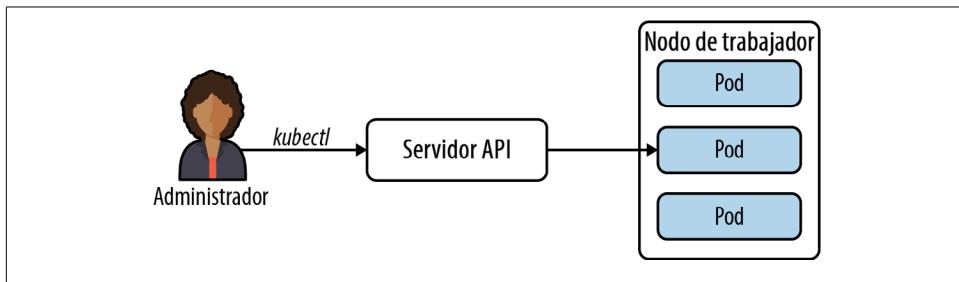
---

<sup>3</sup> En Kubernetes, los servicios son una forma de exponer los pods para que puedan detectarse dentro del clúster de Kubernetes.

solicitud a un servicio en particular, se enrute al punto de conexión IP virtual específico. Kube-proxy es responsable de implementar un tipo de IP virtual para los servicios Ahora que conoce los conceptos básicos de los componentes de Kubernetes, profundicemos un poco más y veamos más información sobre el servidor de API de Kubernetes.

## Objetos del servidor de API de Kubernetes

El servidor de API es responsable de toda la comunicación dentro y fuera de un clúster de Kubernetes, y expone una API de RESTful mediante HTTP. El servidor de API, en un nivel fundamental, le permite consultar y manipular los objetos de Kubernetes. En términos sencillos, los objetos de Kubernetes son entidades con estado que representan el estado general de su clúster ([Figura 4-2](#)). Para empezar a trabajar con estos objetos, necesitamos entender los fundamentos de cada uno.



*Figura 4-2. Interacción del servidor de API con objetos del clúster*

### Pods

En Kubernetes, los pods son la unidad atómica básica más pequeña. Un pod es un grupo de uno o más contenedores que se implementan en los nodos de trabajo. Kubernetes es responsable de administrar los contenedores que se ejecutan dentro de un pod. Los contenedores dentro de un pod siempre terminarán en el mismo nodo de trabajo y están acoplados estrechamente. Dado que los contenedores dentro de un pod están coubicados, se ejecutan en el mismo contexto (es decir, comparten la red y el almacenamiento). Este contexto compartido es característico de los espacios de nombres de Linux, cgroups y cualquier otro aspecto que mantenga el aislamiento (como explicamos en el [Capítulo 3](#)). Los pods también obtienen una dirección IP única.

En escenarios típicos, se ejecuta un solo contenedor dentro de un pod, pero en algunos casos, es necesario que varios contenedores trabajen juntos en un pod. Esta última configuración normalmente se conoce como contenedor *sidecar*. Uno de los ejemplos más comunes donde se ejecuta un contenedor sidecar es al ejecutar un contenedor de registro para la aplicación que enviará los registros a un almacenamiento externo, como un servidor ELK (Elasticsearch, Logstash y Kibana), en caso de que el pod de la aplicación falle o se elimine un pod. Los pods también son inteligentes con respecto a que si muere un proceso en un contenedor, Kubernetes lo reiniciará al instante en función de las comprobaciones de estado definidas en el nivel de la aplicación.

Otra característica de los pods es que permiten el escalado horizontal mediante la replicación implementada a través de ReplicaSets. Esto significa que si desea que su aplicación escale horizontalmente, debe crear más pods mediante ReplicaSets.

Los pods también son de naturaleza efímera, lo que significa que si un pod muere, se migrará y reiniciará en un host diferente. Esto también se logra mediante ReplicaSets.

## ReplicaSets

La confiabilidad es una característica principal de Kubernetes, y dado que nadie ejecutaría una sola instancia de un pod, la redundancia se vuelve importante. Un *ReplicaSet* es un objeto de Kubernetes que garantiza que se esté ejecutando un conjunto estable de pods replicados para mantener un clúster de recuperación automática. Todo esto se logra mediante el bucle de conciliación, que sigue ejecutándose en segundo plano para observar el estado general del clúster de Kubernetes. Los ReplicaSets utilizan el bucle de conciliación y aseguran que si alguno de los pods deja de funcionar o se reinicia, se iniciará un nuevo pod para mantener el estado deseado de la replicación. En general, no debe ocuparse directamente de ReplicaSets, sino más bien usar el objeto *Deployment*, que garantiza las actualizaciones sin tiempo de inactividad de la aplicación y tiene un enfoque declarativo para administrar y operar Kubernetes.

## Implementaciones

Kubernetes es principalmente un orquestador orientado a la sintaxis declarativa. Esto significa que con el fin de implementar nuevas características, debe indicarle a Kubernetes lo que debe hacer y depende de Kubernetes averiguar cómo realizar esa operación de forma segura. Uno de los objetos que Kubernetes ofrece para lograr que el lanzamiento de nuevas versiones de las aplicaciones sea más fluido es *Deployment*. Si sigue actualizando los pods manualmente, deberá reiniciarlos, lo que provocará tiempos de inactividad. Si bien un ReplicaSet sabe cómo mantener el número deseado de pods, no hará una actualización sin tiempo de inactividad. Aquí es donde entra en escena el objeto *Deployment*, ya que ayuda a implementar cambios en los pods sin tiempo de inactividad al mantener un número predefinido de pods activos todo el tiempo antes de que se implemente un nuevo pod actualizado.

## Servicios

Para exponer una aplicación que se ejecuta dentro de un pod, Kubernetes ofrece un objeto llamado *Service*. Dado que Kubernetes es un sistema muy dinámico, es necesario asegurarse de que las aplicaciones estén hablando con los back-ends correctos. Los pods son procesos de corta duración en el mundo de Kubernetes, ya que se crean o destruyen con frecuencia. Los pods se acoplan con una dirección IP única, lo que significa que si usted se basa solo en la dirección IP de los pods, lo más probable es que acabe con un servicio interrumpido cuando un pod muera porque el pod obtendrá una dirección IP diferente después del reinicio, aunque esté ejecutando ReplicaSets. El objeto *Service* ofrece abstracción al definir un conjunto lógico de pods y una política sobre cómo acceder a ellos.

Cada **Service** obtiene una dirección IP estable y un nombre DNS que se puede utilizar para acceder a los pods. Puede definir de forma declarativa los servicios que se encuentran frente a los pods y utilizar el **Label-Selector** para acceder al servicio.

## Espacios de nombre

Dado que varios equipos y proyectos se implementan en un entorno de producción, se vuelve necesario organizar los objetos de Kubernetes. En un sentido simple, los *espacios de nombres* son clústeres virtuales separados por particiones lógicas. Es decir, puede agrupar sus recursos, como implementaciones, pods, y así sucesivamente, en función de las particiones lógicas. A algunas personas les gusta considerar los espacios de nombres como directorios para separar nombres. Cada objeto del clúster tiene un nombre que es único para ese tipo específico de recurso y, de forma similar, cada objeto tiene un UID único en todo el clúster. Los espacios de nombres también le permiten dividir los recursos del clúster entre varios usuarios al establecer *cuotas de recursos*.

## Etiquetas y selectores

A medida que empieza a usar Kubernetes y crear objetos, se dará cuenta de la necesidad de identificar o marcar sus recursos de Kubernetes para agruparlos en entidades lógicas. Kubernetes ofrece *etiquetas* para identificar los metadatos de los objetos, lo que permite fácilmente agrupar y operar los recursos. Las etiquetas son pares de clave-valor que se pueden conectar directamente a los objetos, como pods, espacios de nombres, DaemonSets y así sucesivamente. Puede agregar etiquetas en cualquier momento y modificarlas a su gusto. Para encontrar o identificar los recursos de Kubernetes, puede consultar las etiquetas mediante *selectores de etiquetas*. Por ejemplo, una etiqueta para un tipo de nivel de aplicación puede ser:

```
"tier" : "frontend", "tier" : "backend", "tier" : "midtier"
```

## Anotaciones

Las *anotaciones* también son pares de clave-valor, pero a diferencia de las etiquetas, que se usan para identificar objetos, las anotaciones se usan para mantener información que no identifica el objeto en sí. Por ejemplo, se puede registrar en una anotación la compilación, la versión o la información de la imagen, como las marcas de tiempo, los identificadores de versión, la rama de Git, los números de PR, los hashes de imagen y la dirección del registro.

## Controlador de ingreso

Para que sus servicios reciban tráfico de Internet, debe exponer los puntos de conexión HTTP y HTTPS desde el exterior a los servicios de Kubernetes que se ejecutan en pods. Un *ingreso* le permite exponer los servicios que se ejecutan dentro del clúster al mundo exterior al ofrecer equilibrio de carga con terminaciones de Capa de sockets seguros/Seguridad de capa de transporte (SSL/TLS) utilizando el hospedaje virtual basado en el nombre. Para admitir un ingreso, primero debe elegir un *controlador de ingreso*, que es similar a un proxy inverso, para aceptar conexiones entrantes para HTTP y HTTPS.

## StatefulSets

Para administrar y escalar sus cargas de trabajo con estado en Kubernetes, debe asegurarse de que los pods sean estables (es decir, una red estable y un almacenamiento estable). Los *StatefulSets* aseguran los pedidos de los pods y mantienen su singularidad (a diferencia de *ReplicaSets*). Un *StatefulSet* es un controlador que le ayuda a implementar grupos de pods que siguen siendo resilientes a reinicios y reprogramaciones. Cada pod de un *StatefulSet* tiene convenciones de nombres únicas cuyo valor ordinal comienza en 0 y tiene un ID de red estable asociado (a diferencia de un *ReplicaSet*, en el que la convención de nomenclatura es de naturaleza aleatoria).

## DaemonSets

En entornos normales, ejecutamos una serie de servicios y agentes de demonio en el host, incluidos agentes de registro y agentes de supervisión. Kubernetes le permite instalar estos agentes mediante la ejecución de una copia de un pod en un conjunto de nodos en un clúster mediante *DaemonSets*. Al igual que los *ReplicaSets*, los *DaemonSets* son procesos de larga ejecución que aseguran que el estado deseado y el estado observado sigan siendo los mismos. Al eliminar un *DaemonSet*, también se eliminan los pods que creó anteriormente.

## Trabajos

Los trabajos en el mundo de Kubernetes son entidades de corta duración, que pueden ser pequeñas tareas, como por ejemplo, ejecutar un script independiente. Los trabajos finalmente crean pods. Los trabajos se ejecutan hasta que se terminan correctamente, que es la diferencia principal entre un pod que controla un trabajo y un pod normal que seguirá reiniciándose y reprogramándose si se termina. Si se produce un error en un pod de trabajo antes de completarse, el controlador creará un nuevo pod basado en la plantilla.



Este capítulo es un curso rápido de Kubernetes y solo muestra la superficie de la orquestación de contenedores. Hay otros recursos disponibles que pueden ayudarlo a obtener más información sobre Kubernetes. Estos son algunos de los que recomendamos:

- *Managing Kubernetes* de Brendan Burns y Craig Tracey (O'Reilly, 2018)
- *Kubernetes: Up and Running, 2nd Edition* de Brendan Burns, Joe Beda y Kelsey Hightower (O'Reilly 2019)
- "[Introducción a Kubernetes](#)", un curso gratuito de LinuxFoundationX
- *Cloud Native DevOps with Kubernetes, 2nd Edition* de John Arundel y Justin Domingus (O'Reilly, 2022)

Ahora que hemos visto algunos de los términos básicos del mundo de Kubernetes, veamos los detalles operativos para administrar el clúster.

## Observe, opere y administre clústeres de Kubernetes con kubectl

Una de las formas más comunes de interactuar con un orquestador de contenedores es mediante el uso de una herramienta de línea de comandos o una herramienta gráfica. En Kubernetes, puede interactuar con el clúster de ambas maneras, pero se prefiere la línea de comandos. Kubernetes ofrece *kubectl* como la CLI. *kubectl* se utiliza ampliamente para administrar el clúster. Puede considerar *kubectl* como una navaja suiza con varias funciones de Kubernetes que le permiten implementar y administrar aplicaciones. Si usted es administrador de su clúster de Kubernetes, usaría el comando *kubectl* ampliamente para administrar el clúster. *kubectl* ofrece una variedad de comandos, entre los que se incluyen:

- Comandos de configuración de recursos, que son de naturaleza declarativa
- Comandos de depuración para obtener información sobre las cargas de trabajo
- Comandos de depuración para manipular e interactuar con pods
- Comandos de administración general de clústeres

En esta sección, exploraremos en profundidad Kubernetes centrándonos en los comandos básicos del clúster para administrar clústeres de Kubernetes, pods y otros objetos con la ayuda de *kubectl*.

### Información general y comandos del clúster

El primer paso para interactuar con el clúster de Kubernetes es aprender a obtener información del clúster, la infraestructura y los componentes de Kubernetes con los que trabajará. Para ver los nodos de trabajo que ejecutan las cargas de trabajo en su clúster, emita el siguiente comando *kubectl*:

```
$ ~ kubectl get nodes
NAME      STATUS    ROLES     AGE      VERSION
worker0   Ready     <none>    11d     v1.17.3
worker1   Ready     <none>    11d     v1.17.3
worker2   Ready     <none>    11d     v1.17.3
```

Esto enumerará los recursos de su nodo y su estado, junto con información de la versión. Puede obtener un poco más de información mediante el uso de la marca *-o wide* en el comando *get nodes* de la siguiente manera:

```
$ ~ kubectl get nodes -o wide
NAME      STATUS    ROLES     AGE      VERSION    INTERNAL-IP    EXTERNAL-IP    OS-IMAGE
worker0   Ready     <none>    11d     v1.17.3    10.240.0.20    <none>
worker1   Ready     <none>    11d     v1.17.3    10.240.0.21    <none>
worker2   Ready     <none>    11d     v1.17.3    10.240.0.22    <none>
```

```

KERNEL-VERSION      CONTAINER-RUNTIME
Ubuntu 16.04.7 LTS  4.15.0-1092-azure  containerd://1.3.2
Ubuntu 16.04.7 LTS  4.15.0-1092-azure  containerd://1.3.2
Ubuntu 16.04.7 LTS  4.15.0-1092-azure  containerd://1.3.2

```

Para obtener (`get`) aún más detalles específicos de un recurso o trabajo, puede usar el comando `describe` de la siguiente manera:

```
$ ~ kubectl describe nodes worker0
```

El comando `kubectl describe` es un comando de depuración muy útil. Podría usarlo para obtener información detallada sobre los pods y otros recursos.

El comando `get` se puede utilizar para obtener más información sobre los pods, servicios, controladores de replicación y mucho más. Por ejemplo, para obtener información sobre todos los pods, puede ejecutar lo siguiente:

```

$ ~ kubectl get pods
NAME          READY   STATUS        RESTARTS   AGE
busybox-56d8458597-xpjcg  0/1     ContainerCreating   0          1m

```

Para obtener una lista de todos los espacios de nombres en su clúster, puede usar `get` de la siguiente manera:

```

$ ~ kubectl get namespace
NAME          STATUS   AGE
default       Active   12d
kube-node-lease  Active   12d
kube-public    Active   12d
kube-system    Active   12d
kubernetes-dashboard  Active   46h

```

De forma predeterminada, `kubectl` interactúa con el espacio de nombres `default`. Para usar un espacio de nombres diferente, puede pasar la marca `--namespace` para hacer referencia a objetos en un espacio de nombres:

```

$ ~ kubectl get pods --namespace=default
NAME          READY   STATUS        RESTARTS   AGE
busybox-56d8458597-xpjcg  1/1     Running   0          47h

$ ~ kubectl get pods --namespace=kube-public
No resources found in kube-public namespace.

$ ~ kubectl get pods --namespace=kubernetes-dashboard
NAME          READY   STATUS        RESTARTS   AGE
dashboard-metrics-scraper-779f5454cb-gq82q  1/1     Running   0          2m
kubernetes-dashboard-857bb4c778-qxsnj        1/1     Running   0          46h

```

Para ver los detalles generales del clúster, puede hacer uso de `cluster-info` de la siguiente manera:

```

$ ~ kubectl cluster-info
Kubernetes master is running at https://40.70.3.6:6443
CoreDNS is running at https://40.70.3.6:6443/api/v1/namespaces/kube-system/services/ \
  kube-dns:dns/proxy

```

El comando `cluster-info` le proporciona los detalles del equilibrador de carga de API donde está el plano de control, junto con otros componentes.

En Kubernetes, para mantener una conexión con un clúster específico, use un *contexto*. Un contexto ayuda a agrupar los parámetros de acceso en un solo nombre. Cada contexto contiene un clúster de Kubernetes, un usuario y un espacio de nombres. El contexto actual es el clúster que actualmente está predeterminado para kubectl, y todos los comandos que kubectl emite se ejecutan en este clúster. Puede ver su contexto actual de la siguiente manera:

```
$ ~ kubectl config current-context  
cloud-native-azure
```

Para cambiar el espacio de nombres predeterminado, puede usar un contexto que se registre en el archivo kubeconfig de kubectl de su entorno. El archivo kubeconfig es el archivo real que le dice a kubectl cómo descubrir su clúster de Kubernetes y luego autenticarlo en función de los secretos que configuraron. Por lo general, el archivo se almacena en el directorio de inicio en *.kube/*. Para crear y utilizar un nuevo contexto con un espacio de nombres diferente predeterminado, puede hacer lo siguiente:

```
$ ~ kubectl config set-context test --namespace=mystuff  
Context "test" created.  
$ ~ kubectl config use-context test  
Switched to context "test"
```

Asegúrese de que realmente tenga el contexto (el clúster de Kubernetes de *prueba*) o, de lo contrario, no podrá usarlo.

Las etiquetas, como mencionamos previamente, se utilizan para organizar sus objetos. Por ejemplo, si desea etiquetar el pod busybox con un valor llamado **production**, puede hacerlo de la siguiente manera, donde **environment** es el nombre de la etiqueta:

```
$ ~ kubectl label pods busybox-56d8458597-xpjcg environment=production  
pod/busybox-56d8458597-xpjcg labeled
```

A veces, puede que desee averiguar qué sucede con sus pods e intente depurar un problema. Para ver el registro de un pod, puede ejecutar el comando **logs** en el nombre de un pod:

```
$ ~ kubectl get pods  
NAME READY STATUS RESTARTS AGE  
busybox-56d8458597-xpjcg 0/1 ContainerCreating 0 2d  
$ ~ kubectl logs busybox-56d8458597-xpjcg  
Error from server (BadRequest): container "busybox" in pod "busybox-56d8458597-xpjcg" is  
waiting to start: ContainerCreating
```

A veces, puede tener varios contenedores que se ejecutan dentro de un pod. Para elegir contenedores dentro de este, puede pasar la marca **-c**.

## Administración de pods

Como ya vimos, los pods son los artefactos más pequeños que se pueden implementar en Kubernetes. Como usuario o administrador final, usted trata directamente con los pods y no con los contenedores. Kubernetes controla internamente los contenedores y esta lógica se abstrae. También es importante recordar que todos los contenedores dentro de un pod se

ubican en el mismo nodo. Los pods también tienen un ciclo de vida definido cuyos estados van de *Pending* (pendiente) a *Running* (activo) a *Succeeded* (correcto) o *Failed* (error).

Una de las formas de crear un pod es con el comando `kubectl run` de la siguiente manera:

```
$ kubectl run <name of pod> --image=<name of the image from registry>
```

El comando `kubectl run` extrae una imagen pública de un repositorio de contenedores y crea un pod. Por ejemplo, puede ejecutar una imagen de Hazelcast de la siguiente manera y exponer también el puerto del contenedor:

```
$ ~ kubectl run hazelcast --image=hazelcast/hazelcast --port=5701
deployment.apps/hazelcast created
```



En los entornos de producción, nunca debe ejecutar o crear pods porque Kubernetes no administra directamente estos pods y no se reiniciarán ni reprogramarán en caso de error. Debe utilizar las implementaciones como el método preferido para operar los pods.

El comando `kubectl run` contiene muchos conjuntos de características y puede controlar muchos comportamientos del pod. Por ejemplo, si desea ejecutar un pod en primer plano (es decir, con una terminal interactiva dentro del pod) y no desea reiniciarlo si se bloquea, puede hacer lo siguiente:

```
$ ~ kubectl run -i -t busybox --image=busybox --restart=Never
If you don't see a command prompt, try pressing enter.
/ #
```

Este comando lo registrará directamente dentro del contenedor. Y dado que ejecuta el pod en modo interactivo, puede comprobar que el estado de busybox cambie de *Running* (activo) a *Completed* (completo), como se ve a continuación:

```
$ ~ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
busybox        1/1     Running   0          52s

$ ~ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
busybox        0/1     Completed  0          61s
```

Otra forma de crear pods en Kubernetes es mediante el uso de la sintaxis declarativa en los manifiestos del pod. Los manifiestos del pod, que deben tratarse con la misma importancia que el código de la aplicación, se pueden escribir utilizando YAML o JSON. Puede crear un manifiesto de pod para ejecutar un pod de Nginx de la siguiente manera:

```
apiVersion: v1
kind: pod
metadata:
  name: nginx
spec:
  containers:
```

```

- image: nginx
  name: nginx
  ports:
    - containerPort: 80
      name: http

```

El pod manifiesta información como `kind`, `spec`, y otros datos que se envían al servidor de API de Kubernetes para actuar. Puede guardar el manifiesto del pod con una extensión YAML y usar `apply` de la siguiente manera:

```

$ kubectl apply -f nginx_pod.yaml
pod/nginx created
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
nginx         1/1     Running   0          7s

```

Cuando ejecuta `kubectl apply`, el manifiesto del pod se envía al servidor de API de Kubernetes, que programa el pod de forma instantánea para que se ejecute en un nodo en buen estado del clúster. El proceso del demonio de kubelet supervisa el pod y si se bloquea, se reprograma para que se ejecute en un nodo en buen estado.

Avancemos para ver cómo puede implementar comprobaciones de estado en sus servicios en Kubernetes.

## Comprobaciones de estado

Kubernetes ofrece tres tipos de comprobaciones de estado HTTP, llamadas *sondeos*, para garantizar que la aplicación esté realmente activa: sondeos de vitalidad, sondeos de preparación y sondeos de inicio.

**Sondeo de vitalidad.** Este sondeo es responsable de garantizar que la aplicación esté realmente en buen estado y que funcione correctamente. Después de la implementación, puede que la aplicación tarde unos segundos para estar lista, por lo que puede configurar este sondeo a fin de comprobar un punto de conexión determinado en la aplicación. Por ejemplo, en el siguiente manifiesto de pod, se usó un sondeo de vitalidad para realizar una operación `httpGet` contra la ruta / en el puerto 80. `initialDelaySeconds` se establece en 2, lo que significa que el punto de conexión / no se alcanzará hasta que haya transcurrido este período. Además, hemos establecido el tiempo de espera para que sea 1 segundo y el umbral de error sea 3 errores de sondeo consecutivos. El `periodSeconds` se define como la frecuencia con la que Kubernetes llamará a los pods. En este caso, es 15 segundos.

```

apiVersion: v1
kind: Pod
metadata:
  name: mytest-pod
spec:
  containers:
    - image: test_image
      imagePullPolicy: IfNotPresent
      name: mytest-container
      command: ['sh', '-c', 'echo Container 1 is Running ; sleep 3600']
      ports:
        - name: liveness-port

```

```

    containerPort: 80
    hostPort: 8080
  livenessProbe:
    httpGet:
      path: /
      port: 80
    initialDelaySeconds: 2
    timeoutSeconds: 1
    periodSeconds: 15
    failureThreshold: 3

```

**Sondeo de preparación.** La responsabilidad del sondeo de preparación es identificar cuándo un contenedor está listo para atender la solicitud del usuario. Un sondeo de preparación ayuda a Kubernetes al no agregar un punto de conexión de pod que no está listo a un equilibrador de carga demasiado pronto. El sondeo de preparación se puede configurar simultáneamente con un bloque de sondeo de vitalidad en el manifiesto del pod de la siguiente manera:

```

  containers:
  - name: test_image
    image: test_image
    command: ["/bin/sh"]
    args: ['sh', '-c', 'echo Container 1 is Running ; sleep 3600']
  readinessProbe:
    httpGet:
      path: /
      port: 80
    initialDelaySeconds: 5
    periodSeconds: 5

```

**Sondeo de inicio.** A veces, una aplicación necesita cierto tiempo de inicio adicional en su primera inicialización. En tales casos, puede configurar un sondeo de inicio con una comprobación de HTTP o TCP que tenga un **umbral de error \* periodSeconds**, que es lo suficientemente largo como para abarcar el peor momento de inicio posible:

```

  startupProbe:
    httpGet:
      path: /healthapi
      port: liveness-port
    failureThreshold: 30
    periodSeconds: 10

```

## Límites de recursos

Cuando trata con pods, puede especificar los recursos que necesitará su aplicación. Algunos de los requisitos de recursos más básicos para que un pod se ejecute son la CPU y la memoria. Aunque hay más tipos de recursos que Kubernetes puede manejar, no nos complicaremos analizando solo la CPU y la memoria.

Puede declarar dos parámetros en el manifiesto del pod: *request* (solicitud) y *limit* (límite). En el bloque de solicitud, le dice a Kubernetes el requerimiento mínimo de recursos para que su aplicación opere y, en el bloque de límite, le dice a Kubernetes el umbral máximo. Si su aplicación infringe el umbral en el bloque de límite, se terminará o reiniciará y el pod se expulsará casi por completo. Por ejemplo, en el siguiente manifiesto de pod, colocamos un

límite máximo de CPU de 500m<sup>4</sup> y un límite máximo de memoria de 206Mi<sup>5</sup>, lo que significa que si se superan estos valores, el pod se expulsará y se reprogramará en algún otro nodo:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
          name: http
      resources:
        requests:
          cpu: "100m"
          memory: "108Mi"
        limits:
          cpu: "500m"
          memory: "206Mi"
```

## Volúmenes

Algunas aplicaciones requieren que los datos se almacenen de forma permanente, pero como los pods son entidades de corta duración y con frecuencia se reinician o eliminan sobre la marcha, también se pueden destruir todos los datos asociados con un pod. Los *volúmenes* resuelven este problema con una capa de abstracción de discos de almacenamiento en Azure. Un volumen es una forma de almacenar, recuperar y conservar los datos en los pods durante el ciclo de vida de la aplicación. Si la aplicación tiene estado, debe usar un volumen para conservar los datos. Azure ofrece *Azure Disk* y *Azure Files* para crear los volúmenes de datos que proporcionan esta funcionalidad.

**Solicitud de volumen persistente (PVC).** La PVC sirve como una capa de abstracción entre el pod y el almacenamiento. En Kubernetes, el pod monta volúmenes con la ayuda de las PVC y las PVC hablan con los recursos subyacentes. Debe tener en cuenta que una PVC le permite consumir el recurso de almacenamiento subyacente abstracto, es decir, le permite solicitar una sección de almacenamiento aprovisionado previamente. La PVC define el tamaño y el tipo de disco y, a continuación, monta el almacenamiento real en el pod; Este proceso de enlace puede ser estático, como en un volumen persistente (PV) o dinámico, como en una clase de Storage. Por ejemplo, en el siguiente manifiesto estamos solicitando que un PersistentVolumeClaim tenga almacenamiento de 1Gi:

---

<sup>4</sup> Se permiten solicitudes fraccionarias. Por ejemplo, una CPU se puede dividir en dos 0.5s. La expresión 0.1 equivale a 100m.

<sup>5</sup> El límite y la solicitud se miden en bytes. La memoria se puede expresar como un entero simple o un número de punto fijo mediante los sufijos E, P, T, G, M, K o sus equivalentes de potencia de dos Ei, Pi, Ti, Gi, Mi, Ki.

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: persistent-volume-claim-app1
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: azurefilestorage

```

**Volumen persistente: estático.** El administrador de clústeres, por lo general, el equipo de SRE o DevOps, puede crear un número predefinido de volúmenes persistentes de forma manual, que los usuarios del clúster pueden utilizar como necesiten. El volumen persistente es el método de aprovisionamiento estático. Por ejemplo, en el siguiente manifiesto estamos creando un `PersistentVolume` con una capacidad de almacenamiento de 2Gi:

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: static-persistent-volume-app1
  labels:
    storage: azurefile
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteMany
  storageClassName: azurefilestorage
  azureFile:
    secretName: static-persistence-secret
    shareName: user-app1
    readOnly: false

```

**Clase de almacenamiento: dinámico.** Las clases de almacenamiento son volúmenes aprovisionados dinámicamente para la PVC (es decir, permiten que los volúmenes de almacenamiento se creen a petición). Básicamente, las clases de almacenamiento proporcionan a los administradores del clúster una forma de describir las *clases* de almacenamiento que se pueden ofrecer. Cada clase de almacenamiento tiene un aprovisionador que determina qué complemento de volumen se usa para aprovisionar los volúmenes persistentes.

En Azure, hay dos tipos de aprovisionadores que determinan el tipo de almacenamiento que se usará: `AzureFile` y `AzureDisk`.

`AzureFile` se puede utilizar con el modo de acceso `ReadWriteMany`:

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
parameters:
  skuName: Standard_LRS

```

```

location: eastus
storageAccount: azure_storage_account_name

```

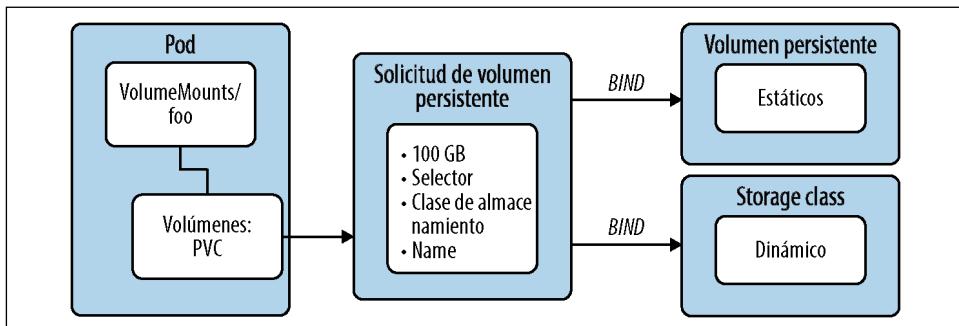
AzureDisk solo se puede utilizar con el modo de acceso ReadWriteOnce:

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/azure-disk
parameters:
  storageaccounttype: Standard_LRS
  kind: Shared

```

En la [Figura 4-3](#), se muestra la relación lógica entre los diversos objetos de almacenamiento que ofrece Kubernetes.



*Figura 4-3. Relación entre pods, PVC, PV y clase de Storage.*

Por último, puede eliminar un pod mediante `kubectl delete pod` de la siguiente manera:

```
$ ~ kubectl delete pod nginx
pod "nginx" deleted
```

Debe asegurarse de que al eliminar un pod, en realidad no se controle con una implementación, porque si es así, volverá a aparecer. Por ejemplo:

```

$ ~ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
hazelcast-84bb5bb976-vrk97  1/1     Running   2          2d7h
nginx-76df748b9-rdbqq      1/1     Running   2          2d7h
$ ~ kubectl delete pod hazelcast-84bb5bb976-vrk97
pod "hazelcast-84bb5bb976-vrk97" deleted
$ ~ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
hazelcast-84bb5bb976-rpcfh  1/1     Running   0          13s
nginx-76df748b9-rdbqq      1/1     Running   2          2d7h

```

La razón por la que se separó un nuevo pod es porque Kubernetes observó que un estado en el clúster se alteró porque los estados deseados y observados no coincidían y, por lo tanto, se inició el bucle de conciliación para equilibrar los pods. Esto podría ocurrir porque el pod se implementó en realidad a través de una implementación y tendría ReplicaSets asociados

con él. Por lo tanto, para eliminar un pod, tendría que eliminar la implementación, que a su vez eliminaría automáticamente el pod. Esto se muestra a continuación:

```
$ ~ kubectl get deployments --all-namespaces
NAMESPACE          NAME            READY   UP-TO-DATE   AVAILABLE   AGE
default            hazelcast       1/1     1           1           2d21h
default            nginx          1/1     1           1           2d21h
kube-system        coredns         2/2     2           2           3d7h
$ ~ kubectl delete -n default deployment hazelcast
deployment.apps "hazelcast" deleted
$ ~ kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
hazelcast-84bb5bb976-rpcfh  0/1    Terminating   0          21m
nginx-76df748b9-rdbqq    1/1    Running    2          2d7h
```

## Kubernetes en producción

Ahora que nos hemos referido a los aspectos básicos de los pods, los conceptos de Kubernetes y cómo funciona el clúster de Kubernetes, veamos cómo vincular los pods y prepararlos para la producción en un clúster de Kubernetes. En esta sección, veremos cómo los conceptos analizados en la sección anterior empoderan las cargas de trabajo de producción y cómo se implementan las aplicaciones en Kubernetes.

### ReplicaSets

Como analizamos, los ReplicaSets habilitan la capacidad de recuperación automática de Kubernetes en el nivel de infraestructura al mantener un número estable de pods. En caso de que se produzca un error en el nivel de la infraestructura (es decir, los nodos que poseen los pods), ReplicaSet volverá a programar los pods en un nodo diferente en buen estado. Un ReplicaSet incluye lo siguiente:

#### Selector

Los ReplicaSets usan etiquetas del pod para encontrar y enumerar los pods que se ejecutan en el clúster a fin de crear réplicas en caso de error.

#### Número de réplicas que se crearán

Especifica cuántos pods se deben crear.

#### Plantilla

Especifica los datos asociados de los pods nuevos que un ReplicaSet debe crear para cumplir con el número deseado de pods.

En los casos de uso de producción reales, necesitará un ReplicaSet para mantener un conjunto estable de pods en ejecución al introducir redundancia. Pero no tiene que ocuparse directamente de ReplicaSet, ya que es una capa de abstracción. En su lugar, debe usar implementaciones, que ofrecen una forma mucho mejor de implementar y administrar pods.

Un ReplicaSet se ve así:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-cnf-replicaset
  labels:
    app: cnfbook
spec:
  replicas: 3
  selector:
    matchLabels:
      app: cnfbook
  template:
    metadata:
      labels:
        app: cnfbook
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

Para crear ReplicaSet a partir de la configuración anterior, guarde el manifiesto como `nginx_Radicaset.yaml` y aplíquelo. Esto creará un ReplicaSet y tres pods, como se muestra a continuación:

```
$ kubectl apply -f nginx_Radicaset.yaml
replicaset.apps/myapp-cnf-replicaset created

$ kubectl get rs
NAME           DESIRED   CURRENT   READY   AGE
myapp-cnf-replicaset   3         3         3       6s

$ kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
myapp-cnf-replicaset-drwp9  1/1     Running   0          11s
myapp-cnf-replicaset-pgwj8  1/1     Running   0          11s
myapp-cnf-replicaset-wqkll  1/1     Running   0          11s
```

## Implementaciones

En los entornos de producción, algo clave que es constante es el cambio. Seguirá actualizando/cambiando las aplicaciones en su entorno de producción a un ritmo muy rápido, ya que la tarea más común que realizará en producción es implementar nuevas características de su aplicación. Kubernetes ofrece el objeto `Deployment` como un estándar para realizar actualizaciones continuas y brinda una experiencia fluida al administrador de clústeres, así como a los usuarios finales. Esto significa que puede insertar actualizaciones en sus aplicaciones sin tener que desactivarlas, ya que `Deployment` garantiza que solo un cierto número de pods estén desactivados mientras se actualizan. Se conocen como *implementaciones sin tiempo de inactividad*. De forma predeterminada, las implementaciones aseguran que al menos el 75 % del número deseado de pods esté activo. Las implementaciones son la forma confiable, segura y actual de implementar nuevas versiones de aplicaciones sin tiempo de inactividad en Kubernetes.

Puede crear un objeto Deployment de la siguiente manera:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

Para aplicar la configuración anterior, guárdela en un archivo llamado *nginx\_Deployment.yaml* y, luego, use kubectl apply de la siguiente manera:

```
$ kubectl apply -f nginx_Deployment.yaml
deployment.apps/nginx-deployment created
```

Curiosamente, la implementación también maneja ReplicaSets, ya que declaramos que necesitaríamos tres réplicas para el pod de implementación. Podemos comprobarlo de la siguiente manera:

```
$ kubectl get deployment
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3           3           18s
$ kubectl get rs
NAME           DESIRED   CURRENT   READY   AGE
nginx-deployment-d46f5678b   3         3         3       29s
$ kubectl get pods
NAME           READY   STATUS   RESTARTS   AGE
nginx-deployment-d46f5678b-dpc7p   1/1     Running   0       36s
nginx-deployment-d46f5678b-kdjxv   1/1     Running   0       36s
nginx-deployment-d46f5678b-kj8zz   1/1     Running   0       36s
```

Entonces, desde el archivo de manifiesto, creamos tres réplicas en el archivo *.spec.replicas*, y para que el objeto Deployment busque los pods que administra *nginx-deployment*, usamos el campo *spec.selector*. Los pods se etiquetan con el campo de plantilla mediante *metadata.labels*.

Ahora vamos a implementar una nueva versión de Nginx. Supongamos que queremos anclar la versión de Nginx a 1.14.2. Solo tenemos que editar el manifiesto de implementación editando el archivo, es decir, cambiar la versión y, a continuación, guardar el archivo de manifiesto, de la siguiente manera:

```
$ kubectl edit deployment.v1.apps/nginx-deployment
deployment.apps/nginx-deployment edited
```

Esto actualizará el objeto Deployment. Puede comprobarlo de la siguiente manera:

```
$ kubectl rollout status deployment.v1.apps/nginx-deployment
Waiting for deployment "nginx-deployment" rollout to finish: 1 out of 3 new replicas have
been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 1 out of 3 new replicas have
been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 3 new replicas have
been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 3 new replicas have
been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 3 new replicas have
been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas are pending
termination...
Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas are pending
termination...
Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas are pending
termination...
deployment "nginx-deployment" successfully rolled out
```

El objeto Deployment garantiza que un determinado número de pods estén siempre disponibles y ejecutándose mientras se actualizan los pods más antiguos. Como ya mencionamos, de forma predeterminada, no más del 25 % de los pods no están disponibles mientras se realiza una actualización. La implementación garantiza un porcentaje de aumento máximo del 25 %, lo que asegura que solo se cree un número determinado de pods del número deseado de pods. Por lo tanto, desde `rollout status`, puede ver claramente que al menos dos pods están disponibles en todo momento mientras se está implementando un cambio. Puede volver a obtener los detalles de la implementación con `kubectl describe deployment`.



Emitimos directamente un comando de implementación mediante `kubectl edit`, pero el enfoque preferido es siempre actualizar el archivo de manifiesto real y, a continuación, aplicar `kubectl`. Esto también le ayuda a mantener sus manifiestos de implementación en el control de versiones. También puede usar el comando `--record` o `set` para realizar la actualización.

### Escalador automático horizontal de pod

Kubernetes admite el escalado dinámico mediante el uso del Escalador automático horizontal de pod (HPA), donde los pods se escalan horizontalmente. Es decir, puede crear un número  $n$  de pods en función de las métricas observadas para su pod si, por ejemplo, desea aumentar o disminuir el número de pods de forma dinámica según la métrica de CPU que se observa. El HPA funciona a través de un bucle de control que, en un intervalo predeterminado de 15 segundos, comprueba el uso de recursos especificados en las métricas (consulte la Figura 4-4).

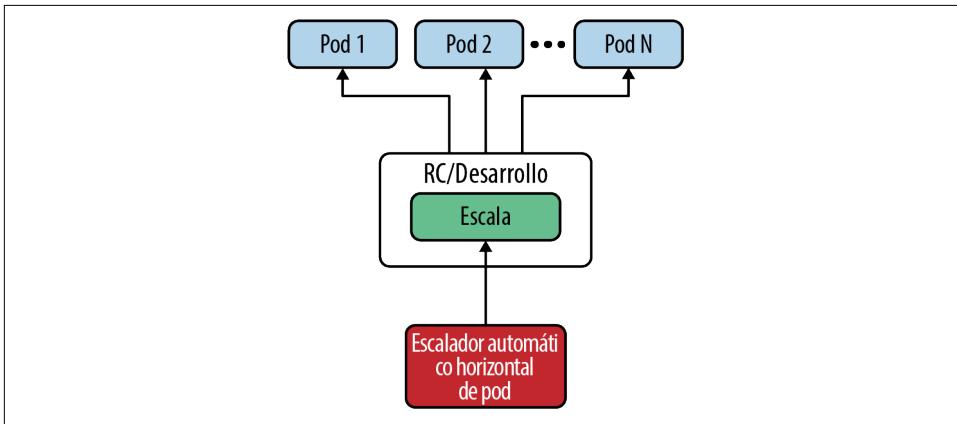


Figura 4-4. Escalador automático horizontal de pod en el trabajo

Es importante tener en cuenta que para usar el HPA, necesitamos el **servidor de métricas**, que recopila métricas de kubelets y las expone en el servidor de API de Kubernetes a través de la API de métricas para que las use el HPA. Primero creamos un escalador automático con `kubectl autoscale` para nuestra implementación, de la siguiente manera:

```
$ ~ kubectl autoscale deployment nginx-deployment --cpu-percent=50 --min=3 --max=10
horizontalpodautoscaler.autoscaling/nginx-deployment autoscaled
```

El comando `kubectl` anterior creará un HPA que garantizará que no se usen menos de tres y no más de 10 pods en nuestra `nginx-deployment`. El HPA aumentará o disminuirá el número de réplicas para mantener un uso de la CPU promedio en todos los pods de no más de un 50 %.

Puede ver su HPA de esta manera:

```
$ ~ kubectl get hpa
NAME          REFERENCE          TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
nginx-deployment Deployment/nginx-deployment 0%/50%     3          10         3          6m59s
```

Podemos agregar el factor de escalado en el que podemos crear el escalador automático con la ayuda de un archivo YAML de manifiesto de HPA que está vinculado a la implementación.

## Servicio

Como mencionamos anteriormente, el entorno de Kubernetes es un sistema muy dinámico donde los pods se crean, destruyen y mueven a un ritmo variable. Este entorno dinámico también abre una puerta a un problema bien conocido: encontrar los pods de réplicas donde reside una aplicación, ya que se ejecutan varios pods para una implementación. Los pods también necesitan una manera de encontrar a los otros pods para poder comunicarse. Kubernetes ofrece el objeto `Service` como una abstracción para un único punto de entrada al grupo de pods. El objeto `Service object` tiene una dirección IP, un nombre DNS y un puerto que nunca cambia mientras exista el objeto.

Esta característica, que se conocía como detección de servicios, básicamente ayuda a otros pods/servicios a llegar a otros servicios en Kubernetes sin lidiar con la complejidad subyacente. Analizaremos el enfoque de detección de servicios nativos de la nube con más detalle en el [Capítulo 6](#).

Para usar un servicio, puede utilizar el YAML de manifiesto de servicio. Supongamos que tiene una aplicación "hello world" simple que ya se ejecuta como una implementación. Una técnica por defecto para exponer este servicio es especificar el `type` de servicio como `ClusterIP`. Este servicio se expondrá en la IP interna del clúster y solo se podrá acceder desde dentro del clúster, de la siguiente manera:

```
---
apiVersion: v1
kind: Service
metadata:
  name: hello-world-service
spec:
  type: ClusterIP
  selector:
    app: hello-world
  ports:
    - port: 8080
      targetPort: 8080
```

El puerto representa el puerto donde estará disponible el servicio y `targetPort` es el puerto de contenedor real donde se reenviará el servicio. En este caso, hemos expuesto el puerto 8080 de la *aplicación hello-world* en el puerto de destino 8080 de la IP del pod:

```
$ kubectl get svc
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP  PORT(S)        AGE
hello-world-service  ClusterIP  10.32.0.40  <none>       8080/TCP      6s
```

La dirección IP que se muestra es la IP del clúster y no la dirección IP real de la máquina. Si usa SSH en un nodo de trabajo, puede comprobar si el servicio se expuso en el puerto 8080 simplemente haciendo un curl de la siguiente manera:

```
ubuntu@worker0:~$ curl http://10.32.0.40:8080
Hello World
```

Si intenta comunicarse con esta dirección IP desde fuera del clúster (es decir, cualquier otro nodo aparte de los de trabajo), no podrá conectarse con ella. Por lo tanto, puede usar `NodePort`, que expone un servicio en la IP de cada nodo en un puerto definido de la siguiente manera:

```
---
apiVersion: v1
kind: Service
metadata:
  name: hello-world-node-service
spec:
  type: NodePort
  selector:
    app: hello-world
  ports:
    - port: 8080
      targetPort: 8080
      nodePort: 30767
```

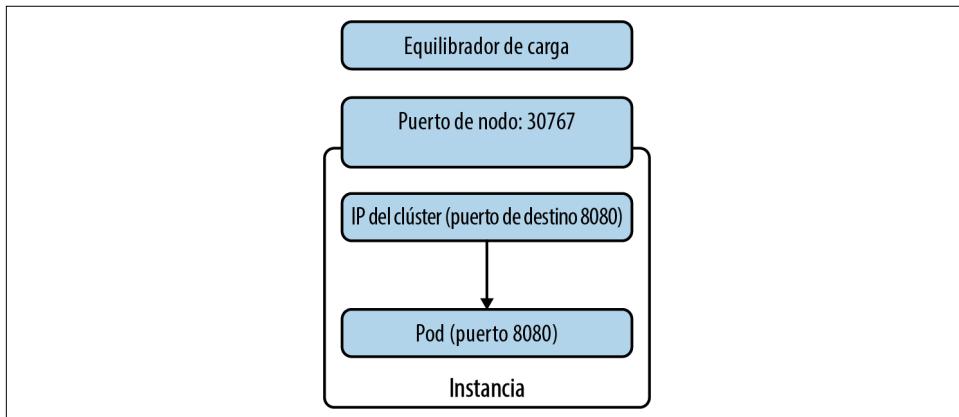
En el manifiesto de servicio, hemos asignado el puerto 8080 del pod a **NodePort** (es decir, puerto de instancia física) 30767. De esta manera, puede exponer la IP directamente o colocar el equilibrador de carga que prefiera. Si ahora aplica un `get svc`, puede ver la asignación de puertos de la siguiente manera:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
<b>hello-world-node-service</b>	<b>NodePort</b>	<b>10.32.0.208</b>	<none>	<b>8080:30767/TCP</b>	<b>8s</b>
<b>hello-world-service</b>	ClusterIP	10.32.0.40	<none>	8080/TCP	41m

Ahora podemos acceder al servicio en el nodo en el puerto 30767:

```
ubuntu@controller0:~$ curl http://10.240.0.21:30767
Hello World
```

La IP en el comando curl es la dirección IP física del nodo de trabajo (no la IP del clúster) y el puerto que se expone es 30767. Incluso puede llegar directamente la IP pública del nodo para el puerto 30767. En la [Figura 4-5](#), se muestra cómo la IP del clúster, el puerto del nodo y el equilibrador de carga se relacionan entre sí.



*Figura 4-5. IP del clúster, puerto de nodo y equilibrador de carga en un nodo de Kubernetes*

Otros tipos de servicios incluyen **LoadBalancer** y **ExternalName**. **LoadBalancer** expone el servicio de forma externa mediante el equilibrador de carga de un proveedor de nube. Los servicios de **NodePort** y **ClusterIP**, a los que se enruta el equilibrador de carga externo, se crean automáticamente, mientras que **ExternalName** asigna un servicio a un nombre DNS, como `my.redisdb.internal.com`. En la [Figura 4-6](#), puede ver cómo se relacionan entre sí los diferentes tipos de servicio.

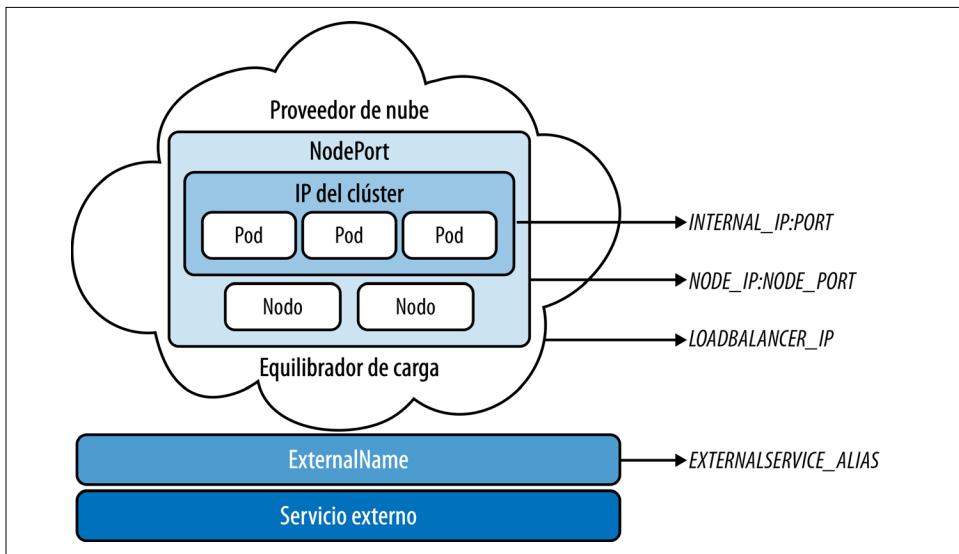


Figura 4-6. Servicio externo y LoadBalancer en un entorno de nube

## Ingreso

El objeto `Service` ayuda a exponer la aplicación dentro y fuera del clúster, pero en los sistemas de producción no podemos permitirnos seguir abriendo puertos nuevos y únicos para todos los servicios que implementamos con `NodePort` y tampoco podemos crear un nuevo equilibrador de carga cada vez que elegimos que el tipo de servicio sea `LoadBalancer`. A veces necesitamos implementar un servicio basado en HTTP y también realizar la descarga de SSL y el objeto `Service` realmente no nos ayuda en esas instancias. En Kubernetes, el objeto `Ingress` realiza el equilibrio de carga HTTP (o, formalmente, equilibrio de carga de Capa 7).

Para trabajar con `Ingress`, primero tenemos que configurar un controlador de ingreso.<sup>6</sup> Vamos a configurar un controlador de ingreso de Application Gateway de Azure Kubernetes Service (AKS) para comprenderlo mejor y ver cómo se comporta, pero en general, una de las formas más fáciles de entenderlo mejor es observar lo siguiente:

---

6 Para que el recurso `Ingress` funcione, el clúster debe tener un controlador de ingreso en ejecución. A diferencia de otros tipos de controladores que se ejecutan como parte del binario `kube-controller-manager`, los controladores de ingreso no se inician automáticamente con un clúster. Existen diferentes tipos de controladores de ingreso; por ejemplo, Azure ofrece el controlador de ingreso de Application Gateway de AKS para configurar Azure Application Gateway.

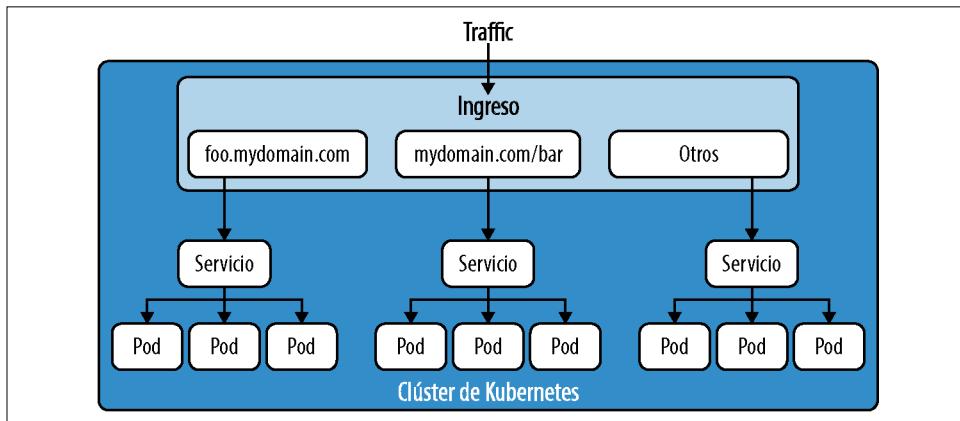
```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
    - host: "foo.bar.com"
      http:
        paths:
          - pathType: Prefix
            path: "/bar"
            backend:
              service:
                name: hello-world-node-service
                port:
                  number: 8080
    - host: "*.foo.com"
      http:
        paths:
          - pathType: Prefix
            path: "/foo"
            backend:
              service:
                name: service2
                port:
                  number: 80

```

En el manifiesto de ingreso, creamos dos reglas y asignamos *foo.bar.com* como un host a un subdominio */bar*, que enruta a nuestro servicio anterior *hello-world-node-service*.

Del mismo modo, podemos tener varias rutas definidas para un dominio y enrutarlas a otros servicios. Hay varias formas de configurar el ingreso según sus necesidades, que puede abarcar desde enrutar un solo dominio a varios servicios o enrutar varios dominios a varios servicios (consulte la [Figura 4-7](#)).



*Figura 4-7. Ingreso con varias rutas para un dominio*

Por último, puede especificar el soporte de TLS mediante la creación de un objeto **Secret** y luego usar ese secreto en su especificación de ingreso de la siguiente manera:

```
apiVersion: v1
kind: Secret
metadata:
  name: my_tls_secret
  namespace: default
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls
```

Puede proteger el ingreso especificando solo el certificado TLS codificado Base64 y la clave. Cuando haga referencia a este secreto en el manifiesto de ingreso, aparecerá de la siguiente manera:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tls-example-ingress
spec:
  tls:
    - hosts:
        - https.mywebsite.example.com
      secretName: my_tls_secret
  rules:
    - host: https.mywebsite.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: service1
                port:
                  number: 80
```

Los controladores de ingreso se ocupan de muchas características y complejidades, y el controlador de ingreso de Azure Gateway maneja muchos de ellos. Por ejemplo, aprovecha el equilibrador de carga del gateway de aplicación de la capa 7 nativa de Azure para exponer sus servicios a Internet. Analizaremos esto con más detalle cuando presentemos AKS en el [Capítulo 5](#).

## DaemonSet

Los DaemonSets, como ya explicamos, se usan normalmente para ejecutar un agente en una serie de nodos en el clúster de Kubernetes. El agente se ejecuta dentro de un contenedor que los pods abstraen. La mayoría de las veces, los ingenieros de confiabilidad del sitio y DevOps prefieren ejecutar un agente de registro o un agente de supervisión en cada nodo para obtener telemetría y eventos de aplicaciones. De forma predeterminada, un DaemonSet crea una copia de un pod en cada nodo, aunque esto también se puede restringir con un selector de nodos. Hay muchas similitudes entre ReplicaSets y DaemonSets, pero la distinción clave entre ellos es el requisito (con DaemonSets) de ejecutar una aplicación de un solo agente (es decir, pod) en todos sus nodos.

Una de las formas de ejecutar un contenedor de registros es mediante la implementación de un pod en cada nodo usando un DaemonSet. Fluentd es una solución de registro de open source que se usa ampliamente para recopilar registros de los sistemas. En este ejemplo, se muestra una de las formas de implementar Fluentd mediante un DaemonSet:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    central-log-k8s: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
        - key: node-role.kubernetes.io/master
          effect: NoSchedule
      containers:
        - name: fluentd-elasticsearch
          image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
          resources:
            limits:
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          volumeMounts:
            - name: varlog
              mountPath: /var/log
            - name: varlibdockercontainers
              mountPath: /var/lib/docker/containers
              readOnly: true
      terminationGracePeriodSeconds: 30
  volumes:
    - name: varlog
      hostPath:
        path: /var/log
    - name: varlibdockercontainers
      hostPath:
        path: /var/lib/docker/containers
```

En la configuración anterior de DaemonSet, creamos un DaemonSet que implementará el contenedor Fluentd en cada nodo. También creamos una *tolerancia*, que no programa el nodo de Fluentd en los nodos maestros (plan de control).



Kubernetes ofrece características de programación llamadas *contaminaciones* y *tolerancias*:

- Las contaminaciones en Kubernetes permiten que un nodo pueda repeler un conjunto de pods (es decir, si desea que ciertos nodos no programen algún tipo de pod).
- Las tolerancias se aplican a los pods y permiten (pero no requieren) que los pods se programen en los nodos con contaminaciones que coincidan.

Las contaminaciones y las tolerancias funcionan en conjunto para garantizar que los pods no se programen en nodos inadecuados. Se aplican una o más contaminaciones a un nodo. Esto indica que el nodo no debería aceptar ningún pod que no tolere las contaminaciones.

Puede comprobar los pods que se crearon automáticamente para cada nodo de trabajo de la siguiente manera:

```
$ kubectl apply -f fluentd.yaml
daemonset.apps/fluentd-elasticsearch created

$ kubectl get ds --all-namespaces
NAMESPACE      NAME            DESIRED   CURRENT   READY    UP-TO-DATE   AVAILABLE
kube-system    fluentd-elasticsearch   3         3         3        3           3
NODE SELECTOR   AGE
<none>        5m12s

$ kubectl get pods --namespace=kube-system -o wide
NAME                  READY   STATUS    RESTARTS   AGE     IP          NODE
fluentd-elasticsearch-5jxg7   1/1    Running   1          45m    10.200.1.53  worker1
fluentd-elasticsearch-c5s4c   1/1    Running   1          45m    10.200.2.32  worker2
fluentd-elasticsearch-r4pqz   1/1    Running   1          45m    10.200.0.43  worker0
NOMINATED NODE   READINESS GATES
<none>            <none>
<none>            <none>
<none>            <none>
```

## Trabajos

A veces necesitamos ejecutar un pequeño script hasta que pueda finalizar correctamente. Kubernetes le permite hacerlo con el objeto Job. El objeto Job crea y administra pods que se ejecutarán hasta que se complete correctamente y, a diferencia de los pods normales, una vez completada la tarea, estos pods que creó Job no se reinician. Puede utilizar un YAML de Job para describir un Job simple de la siguiente manera:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: examplejob
spec:
  template:
    metadata:
      name: examplejob
    spec:
```

```

    containers:
      - name: examplejob
    image: busybox
    command: ["echo", "Cloud Native with Azure"]
    restartPolicy: Never

```

Aquí creamos un Job para imprimir un comando de shell. Para crear un Job, puede guardar el YAML anterior y aplicarlo con `kubectl apply`. Una vez que aplique el manifiesto del trabajo, Kubernetes creará el trabajo y lo ejecutará de inmediato. Puede comprobar el estado del job usando `kubectl describe` de la siguiente manera:

```

$ kubectl apply -f job.yaml
job.batch/examplejob created
$ kubectl get jobs
NAME      COMPLETIONS DURATION   AGE
examplejob 1/1        3s         9s
$ kubectl describe job examplejob
Name:           examplejob
Namespace:      default
Selector:       controller-uid=f6887706-85ef-4752-8911-79cc7ab33886
Labels:         controller-uid=f6887706-85ef-4752-8911-79cc7ab33886
                job-name=examplejob
Annotations:   kubernetes.io/last-applied-configuration:
                  {"apiVersion":"batch/v1","kind":"Job","metadata":{}},"name":"examplejob","namespace":"default"}, "spec": {"template": {"metadata": ...
Parallelism:   1
Completions:   1
Start Time:    Mon, 07 Sep 2020 01:08:53 +0530
Completed At:  Mon, 07 Sep 2020 01:08:56 +0530
Duration:      3s
Pods Statuses: 0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels: controller-uid=f6887706-85ef-4752-8911-79cc7ab33886
          job-name=examplejob
  Containers:
    examplejob:
      Image:      busybox
      Port:       <none>
      Host Port: <none>
      Command:
        echo
        Cloud Native with Azure
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Events:
    Type      Reason     Age   From            Message
    ----      -----     --   --   -----
    Normal    SuccessfulCreate 61s   job-controller  Created pod: examplejob-mcqzc
    Normal    Completed      58s   job-controller  Job completed

```

## Resumen

Kubernetes es una plataforma eficaz que se creó a partir de una década de experiencia adquirida por las aplicaciones contenedoras a escala en Google. Básicamente, Kubernetes condujo a la creación de la Cloud Native Computing Foundation y fue el primer proyecto que se graduó en el marco de la fundación. Esto trajo como resultado una gran optimización del ecosistema de microservicios con respecto al soporte y la mayor adopción del entorno nativo de la nube. En este capítulo, analizamos los diversos componentes y conceptos que permiten a Kubernetes operar a escala. Este capítulo también prepara el camino hacia los próximos capítulos en los que utilizaremos la plataforma de Kubernetes para transmitir las aplicaciones nativas de la nube de nivel de producción.

Debido a la complejidad subyacente de la administración de un clúster de Kubernetes, en el [Capítulo 5](#) analizaremos cómo crear y usar un clúster de este tipo. También exploraremos Azure Kubernetes Service y mucho más.

## CAPÍTULO 5

# Creación de un clúster de Kubernetes en Azure

Ahora que entiende los aspectos básicos de cómo funciona un clúster de Kubernetes bajo la superficie, es momento de poner ese conocimiento a prueba para compilar y usar un clúster de Kubernetes. Existen varias herramientas disponibles para crear y ejecutar un clúster de Kubernetes en un entorno de producción con alta disponibilidad. Algunas de las herramientas más comunes son kops, kubeadm, Kubespray y Rancher. Estas herramientas tienen manuales escritos previamente sobre la creación de un clúster que se pueden ejecutar sin problemas en un entorno de producción con una combinación de tecnologías.

En este capítulo, no usaremos una herramienta preconfigurada. En cambio, tomaremos un enfoque más práctico para crear un clúster que pueda utilizar en un entorno de producción.

## Creación de un clúster de Kubernetes desde cero

En esta sección, compilaremos un clúster de Kubernetes desde cero con Ansible, Terraform y Packer. Le mostraremos cómo crear el clúster de forma manual para que pueda comprender mejor los diversos componentes del clúster. Si bien esta no es la forma preferida de crear o ejecutar un clúster de Kubernetes de nivel de producción, le proporcionará un buen conocimiento sobre los fundamentos.

Para empezar, deberá clonar el [GitHub para este libro](#) y abrir la carpeta Chapter5. En este capítulo, se supone que ya configuró su cuenta de Azure siguiendo los pasos del [Capítulo 2](#) y que experimentó con la creación de la infraestructura utilizando los manuales de Terraform, Packer y Ansible.

Organizamos el código en el repositorio de Git Chapter5 de la siguiente manera:

```
$ Chapter5 git:(master)
.
├── AKS           - Azure Kubernetes Service Cluster
├── Ansible-Playbooks - Ansible playbooks for creating k8s cluster
└── Deployments
```

```
└── K8S_Packer_Image - Machine images for worker and controller
    ├── Kubernetes_Cluster- Terraform initialization directory for k8s
    └── README.md
        └── terraform_modules - Terraform modules for k8s cluster
```

En esta sección, solo utilizaremos `K8S_Packer_Image`, `Ansible-Playbooks`, `Kubernetes_Cluster`, y `terraform_modules`.

## Creación de un grupo de recursos

El primer paso es crear un grupo de recursos para todos los componentes de la infraestructura de modo que podamos agruparlos juntos. Para ello, vaya al directorio `Chapter5/Kubernetes_Cluster/A-Resource_Group` y ejecute el siguiente comando:

```
$ terraform init
```

Después de la inicialización, ejecute lo siguiente para crear un grupo de recursos:

```
$terraform apply
```

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create
```

```
Terraform will perform the following actions:
```

```
# module.CNA-Terraform-Resource-Grp.azurerm_resource_group.generic-resource-gp
# will be created
+ resource "azurerm_resource_group" "generic-resource-gp" {
    + id      = (known after apply)
    + location = "eastus2"
    + name    = "K8Scluster"
    + tags    = {
        + "cluster"     = "k8s-experiments"
        + "environment" = "dev"
    }
}
```

```
Plan: 1 to add, 0 to change, 0 to destroy.
```

```
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
```

```
Enter a value: yes
```

Para continuar, ingrese `yes`. Esto creará un grupo de recursos llamado `K8Scluster` en la región `eastus2` de Azure.

## Creación de imágenes para máquinas de trabajo y controladores

El segundo paso es crear las imágenes de Packer de las imágenes de la instancia del controlador y trabajador. En el directorio `Chapter5/K8S_Packer_Image`, actualice `controller.json` y `worker.json` con las credenciales de Azure para `client_id`, `client_secret`, `tenant_id` y `subscription_id`. Ahora ejecute los comandos `packer` de la siguiente manera:

```

$ packer build worker.json
==> Builds finished. The artifacts of successful builds are:
--> azure-arm: Azure.ResourceManagement.VMImage:

OSType: Linux
ManagedImageResourceGroupName: K8Scluster
ManagedImageName: k8s_worker
ManagedImageId: /subscriptions/b5624140-9087-4311-a94a-3b16a2e84792/resourceGroups/ \
K8Scluster/providers/Microsoft.Compute/images/k8s_worker
ManagedImageLocation: eastus2

$ packer build controller.json
==> Builds finished. The artifacts of successful builds are:
--> azure-arm: Azure.ResourceManagement.VMImage:

OSType: Linux
ManagedImageResourceGroupName: K8Scluster
ManagedImageName: k8s_controller
ManagedImageId: /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \
K8Scluster/providers/Microsoft.Compute/images/k8s_controller
ManagedImageLocation: eastus2

```

Tenga en cuenta el `ManagedImageId`, que usaremos más adelante en el capítulo.

## Creación del back-end de la cuenta de almacenamiento

A continuación, crearemos la cuenta de almacenamiento para almacenar toda la asignación de infraestructura que utilizará Terraform. Una vez más, inicializaremos primero el repositorio con `terraform init` en `Kubernetes_Cluster/B-Storage_Account_backend` y luego aplicaremos el comando de la siguiente manera:

```

$ terraform apply
.

.

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

Blob-ID = https://cnabookprod.blob.core.windows.net/k8s-cluster-dev/test
Blob-URL = https://cnabookprod.blob.core.windows.net/k8s-cluster-dev/test
Primary-Access-Key = ymMDE1pUQgtuxh1AOJyUvlvfXnmjAeJEHl2XvMmQ38AZp108Z0Xk4Hrw4N/ \
d8yovb8FQ5VzqtREH94gzPCzAWCA==

```

Esto creará una cuenta de almacenamiento con el nombre `cnabookprod`. Actualice el `bash_profile` con un valor de clave de acceso principal con el nombre de la variable `ARM_ACCESS_KEY`:

```
export ARM_ACCESS_KEY=ymMDE1pUQgtuxh1AOJyUvlvfXnmjAeJEHl2XvMmQ38AZp108Z0Xk4Hrw4N/ \
d8yovb8FQ5VzqtREH94gzPCzAWCA==
```

## Creación de una red virtual de Azure

El siguiente paso es crear la red virtual en la que hospedaremos nuestro clúster de Kubernetes. Vaya al directorio *Chapter5/Kubernetes\_Cluster/C-Virtual\_Network* y vuelva a inicializar Terraform con `terraform init`. Una vez inicializado, aplique la siguiente configuración para crear una red virtual de Azure:

```
$ terraform apply
.
.
.
Apply complete! Resources: 6 added, 0 changed, 0 destroyed.
Releasing state lock. This may take a few moments...
Outputs:

subnet_id = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \
  K8Scluster/providers/Microsoft.Network/virtualNetworks/cna-k8s-vnet/subnets/cna-k8s-subnet
vnet_id = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \
  K8Scluster/providers/Microsoft.Network/virtualNetworks/cna-k8s-vnet
```

Esto creará una red virtual con un CIDR de 10.240.0.0/24 llamado `cnak-8s-vnet`.

## Creación de direcciones IP públicas para el equilibrador de cargas

Ahora crearemos direcciones IP públicas para el equilibrador de carga junto con todos los nodos de trabajo y plano de control. Una vez que hayamos creado las IP públicas para todos los recursos, crearemos el equilibrador de carga para que los nodos del plano de control expongan el servidor de API a los clientes remotos.

Para crear las direcciones IP públicas, vaya al directorio *Chapter5/Kubernetes\_Cluster/D-K8S\_PublicIP* e inicialice Terraform con `terraform init`. El código se organizará de la siguiente manera:

```
module "K8S-API-Server-Public-IP" {
  source = "../../terraform_modules/Azure_PublicIP"
  name_of_ip = "k8s_master_lb"
  resource-grp-name = "K8Scluster"
  azure-dc = "eastus2"
  env = "dev"
  type-of-cluster = "k8s-experiments"
}

module "Worker0" {
  source = "../../terraform_modules/Azure_PublicIP"
  name_of_ip = "Worker0"
  resource-grp-name = "K8Scluster"
  azure-dc = "eastus2"
  env = "dev"
  type-of-cluster = "k8s-experiments"
}
module "Worker1" {
  source = "../../terraform_modules/Azure_PublicIP"
  name_of_ip = "Worker1"
```

```

resource-grp-name = "K8Scluster"
azure-dc = "eastus2"
env = "dev"
type-of-cluster = "k8s-experiments"
}
module "Worker2" {
  source = "../../terraform_modules/Azure_PublicIP"
  name_of_ip = "Worker2"
  resource-grp-name = "K8Scluster"
  azure-dc = "eastus2"
  env = "dev"
  type-of-cluster = "k8s-experiments"
}

module "Controller0" {
  source = "../../terraform_modules/Azure_PublicIP"
  name_of_ip = "Controller0"
  resource-grp-name = "K8Scluster"
  azure-dc = "eastus2"
  env = "dev"
  type-of-cluster = "k8s-experiments"
}
module "Controller1" {
  source = "../../terraform_modules/Azure_PublicIP"
  name_of_ip = "Controller1"
  resource-grp-name = "K8Scluster"
  azure-dc = "eastus2"
  env = "dev"
  type-of-cluster = "k8s-experiments"
}
module "Controller2" {
  source = "../../terraform_modules/Azure_PublicIP"
  name_of_ip = "Controller2"
  resource-grp-name = "K8Scluster"
  azure-dc = "eastus2"
  env = "dev"
  type-of-cluster = "k8s-experiments"
}

```

Una vez que se inicialice Terraform, puede aplicar la configuración mediante **terraform apply** de la siguiente manera, que creará las direcciones IP públicas:

```

$ terraform apply
.
.
.
Apply complete! Resources: 7 added, 0 changed, 0 destroyed.
Releasing state lock. This may take a few moments...
Outputs:
Controller0_IP = 52.252.6.89
Controller0_IP_ID = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \
  K8Scluster/providers/Microsoft.Network/publicIPAddresses/Controller0
Controller1_IP = 52.254.50.7
Controller1_IP_ID = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \
  K8Scluster/providers/Microsoft.Network/publicIPAddresses/Controller1
Controller2_IP = 52.251.58.212
Controller2_IP_ID = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \
  K8Scluster/providers/Microsoft.Network/publicIPAddresses/Controller2
PublicIP_ID = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \

```

```

K8Scluster/providers/Microsoft.Network/publicIPAddresses/k8s_master_lb
Public_IP = 52.254.73.23
Worker0_IP = 52.251.59.169
Worker0_IP_ID = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \
    K8Scluster/providers/Microsoft.Network/publicIPAddresses/Worker0
Worker1_IP = 52.251.59.78
Worker1_IP_ID = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \
    K8Scluster/providers/Microsoft.Network/publicIPAddresses/Worker1
Worker2_IP = 52.254.50.15
Worker2_IP_ID = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \
    K8Scluster/providers/Microsoft.Network/publicIPAddresses/Worker2

```

El código anterior creará siete IP públicas: seis para las instancias de nodo (de trabajo y controlador) y una para el equilibrador de carga. Ahora podemos crear el equilibrador de carga y adjuntar su IP al ir al directorio *Chapter5/Kubernetes\_Cluster/EK8S-API-Public-loadbalancer* e inicializar Terraform. Luego, podemos usar de nuevo `terraform apply` de la siguiente manera:

```

$ terraform apply
.
.
.

Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
Releasing state lock. This may take a few moments...

Outputs:

lb_backend_pool = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \
    K8Scluster/providers/Microsoft.Network/loadBalancers/k8s_master_lb/backendAddressPools/ \
    k8s-control-plane
load_balancer_frontend_id = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/ \
    resourceGroups/K8Scluster/providers/Microsoft.Network/loadBalancers/k8s_master_lb/ \
    frontendIPConfigurations/K8S-frontend-config
load_balancer_id = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \
    K8Scluster/providers/Microsoft.Network/loadBalancers/k8s_master_lb
load_balancer_private_ip =
load_balancer_public_ip = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/ \
    resourceGroups/K8Scluster/providers/Microsoft.Network/publicIPAddresses/k8s_master_lb

```

Este paso creará un equilibrador de carga y aplicará las reglas del equilibrador de carga que predefinimos en el archivo de configuración de Terraform.

## Creación de instancias de controlador y trabajo

En este punto, estamos listos para crear los nodos de instancias que constituirán nuestro clúster de Kubernetes. Vamos a crear un clúster de seis nodos en el que tres serán nodos de trabajo que ejecutarán la carga de trabajo real (pods, etc.) y tres serán parte del plano de control.

Antes de esto, tenemos que actualizar el archivo *main.tf* en el directorio */Chapter5/Kubernetes\_Cluster/F-K8S-Nodes/main.tf* con el ID de imagen para el nodo de trabajo y el nodo de controlador. Además, tenemos que actualizar la ruta para la ubicación de la clave de Secure Shell (SSH) que se utilizará para ssh en las máquinas:

```

module "master" {
  source = "../../terraform_modules/Azure_VMs-Master"
  azure_dc = "eastus2"
  resource_grp_name = "K8Scluster"
  private_ip_addresses = ["10.240.0.10", "10.240.0.11", "10.240.0.12"]
  vm_prefix = "controller"
  username = "ubuntu"
  public_ip_address_id = [data.terraform_remote_state.k8s_public_ip.outputs. \
    Controller0_IP_ID,data.terraform_remote_state.k8s_public_ip.outputs. \
    Controller1_IP_ID,data.terraform_remote_state.k8s_public_ip.outputs.Controller2_IP_ID]
  vm_size = "Standard_D1_v2"
  env = "dev"
  type-of-cluster = "k8s-experiments"
  vm_count = 3
  subnet_id = data.terraform_remote_state.k8s_vnet.outputs.subnet_id
  image_id = "/subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/ \
    resourceGroups/K8Scluster/providers/Microsoft.Compute/images/k8s_controller"
  ssh_key = "${file("/Users/nissingh/TESTING/SSH_KEYS/id_rsa.pub")}"
  lb_backend_pool = data.terraform_remote_state.k8s_loadbalancer.outputs.lb_backend_pool
}

```

En la configuración anterior (*main.tf*), el ID de la imagen se actualizó para el nodo de controlador (maestro) junto con la ruta de acceso a la clave SSH. Ahora puede generar un par de claves SSH mediante el comando `ssh-keygen` para `ssh_key`. Del mismo modo, puede actualizar el archivo de configuración del trabajo y, a continuación, inicializar Terraform como antes. Una vez que se finaliza la inicialización, puede usar `terraform apply` para crear las instancias de trabajo y controlador:

```

$ terraform apply
.
.
.
.
Apply complete! Resources: 17 added, 0 changed, 0 destroyed.
Releasing state lock. This may take a few moments...

```

Outputs:

```

master_machine_id = [
  [
    "/subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/K8Scluster/ \
      providers/Microsoft.Compute/virtualMachines/controller0",
    "/subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/K8Scluster/ \
      providers/Microsoft.Compute/virtualMachines/controller1",
    "/subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/K8Scluster/ \
      providers/Microsoft.Compute/virtualMachines/controller2",
  ],
]
master_machine_name = [
  [
    "controller0",
    "controller1",
    "controller2",
  ],
]
master_machine_private_ips = [
  [
    "10.240.0.10",
    "10.240.0.11",
  ]
]

```

```

        "10.240.0.12",
    ],
}
worker_machine_id = [
[
    "/subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/K8Scluster/ \
    providers/Microsoft.Compute/virtualMachines/worker0",
    "/subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/K8Scluster/ \
    providers/Microsoft.Compute/virtualMachines/worker1",
    "/subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/K8Scluster/ \
    providers/Microsoft.Compute/virtualMachines/worker2",
],
]
worker_machine_name = [
[
    "worker0",
    "worker1",
    "worker2",
],
]
worker_machine_private_ips = [
[
    "10.240.0.20",
    "10.240.0.21",
    "10.240.0.22",
],
]

```

Una vez que se completa `terraform apply`, verá que los nodos se han aprovisionado para su clúster de Kubernetes (consulte la [Figura 5-1](#)).

Name	Type	Status	Resource group	Location	Source	Maintenance status	Subscription	Public IP address
controller0	Virtual machine	Running	K8Scluster	East US 2	Image	-	Visual Studio Enterprise	52.252.6.89
controller1	Virtual machine	Running	K8Scluster	East US 2	Image	-	Visual Studio Enterprise	52.254.50.7
controller2	Virtual machine	Running	K8Scluster	East US 2	Image	-	Visual Studio Enterprise	52.251.58.212
worker0	Virtual machine	Running	K8Scluster	East US 2	Image	-	Visual Studio Enterprise	52.251.59.169
worker1	Virtual machine	Running	K8Scluster	East US 2	Image	-	Visual Studio Enterprise	52.251.59.78
worker2	Virtual machine	Running	K8Scluster	East US 2	Image	-	Visual Studio Enterprise	52.254.50.15

*Figura 5-1. Nodos de trabajo y controlador en funcionamiento*

## Uso de Ansible para implementar y configurar los nodos del controlador de Kubernetes

Con la infraestructura en funcionamiento, es momento de configurar el clúster, ya que recién tenemos el esqueleto desnudo listo. Para implementar y configurar Kubernetes para trabajar en el nodo, utilizaremos manuales de Ansible.

En primer lugar, actualice el archivo *hosts* en *Chapter5/Ansible-Playbooks/hosts* con las direcciones IP de las instancias de trabajo y controlador, junto con el nombre de usuario SSH y la ruta de acceso de la clave privada SSH que se generó con *ssh-keygen*:

```
[controllers]
inventory_hostname=controller0 ansible_host=52.252.6.89
inventory_hostname=controller1 ansible_host=52.254.50.7
inventory_hostname=controller2 ansible_host=52.251.58.212

[workers]
inventory_hostname=worker0 ansible_host=52.251.59.169
inventory_hostname=worker1 ansible_host=52.251.59.78
inventory_hostname=worker2 ansible_host=52.254.50.15

[all:vars]
ansible_user = ubuntu
ansible_ssh_private_key_file = /Users/SSH_KEYS/id_rsa
```

Una vez que se actualiza el archivo *hosts*, tenemos que actualizar *groupvars* (que se encuentra en el archivo *all* en el directorio *Chapter5/Ansible-Playbooks/group\_vars*) con *loadbalancer\_public\_ip* y la clave de cifrado. Dado que Kubernetes almacena una variedad de datos, incluidos el estado del clúster, las configuraciones de aplicación y los secretos, tenemos que cifrar estos datos en reposo. Kubernetes admite la capacidad de cifrar los datos del clúster en reposo.

Para crear una clave de cifrado, simplemente puede emitir el siguiente comando en su terminal y pegar el secreto generado en el archivo *group\_vars/all* de la siguiente manera:

```
$ head -c 32 /dev/urandom | base64
Uvivn+4ONy9yqRf0ynRV0psEE7WsfyvYnM7VNakiNeA=
```

También tendrá que actualizar la IP del equilibrador de carga. El archivo final *group\_vars* debería verse así:

```
...
#Change loadbalancer_public_ip
loadbalancer_public_ip : 52.254.73.23
controller_private_ips_list : 10.240.0.10,10.240.0.11,10.240.0.12
# Keep k8s_internal_virtual_ip as it is
k8s_internal_virtual_ip: 10.32.0.1
k8s_cluster_cidr: "10.200.0.0/16"
k8s_cluster_name: "cloud-native-azure"
# Generate your own encryption_key : "head -c 32 /dev/urandom | base64"
encryption_key: Uvivn+4ONy9yqRf0ynRV0psEE7WsfyvYnM7VNakiNeA=
#ETC config below for systemd file, place private ip's. No need to change
controller_0_ip: 10.240.0.10
controller_1_ip: 10.240.0.11
controller_2_ip: 10.240.0.12
```

Una vez que hayamos finalizado la actualización de los archivos, debemos instalar las herramientas del lado del cliente en el sistema local, principalmente *kubectl*, *cfssl* y *cfssljson*. CFSSL es el kit de herramientas de infraestructura de clave pública/seguridad de la capa de transporte (PKI/TLS) de Cloudflare para firmar, verificar y agrupar los certificados TLS en general. Usaremos este kit de herramientas para generar el certificado TLS para los nodos de Kubernetes.

Puede instalar los programas `cfssl` y `cfssljson` en macOS de la siguiente manera:

```
$ brew install cfssl
```

En Linux, puede hacer lo siguiente:

```
$ wget -q --show-progress --https-only --timestamping \
  https://github.com/cloudflare/cfssl/releases/download/v1.4.1/cfssl_1.4.1_linux_amd64 \
  https://github.com/cloudflare/cfssl/releases/download/v1.4.1/cfssljson_1.4.1_linux_amd64
$ chmod +x cfssl_1.4.1_linux_amd64 cfssljson_1.4.1_linux_amd64
$ sudo mv cfssl_1.4.1_linux_amd64 /usr/local/bin/cfssl
$ sudo mv cfssljson_1.4.1_linux_amd64 /usr/local/bin/cfssljson
```

Ejecute el comando `cfssl version` para asegurarse de que se instaló correctamente:

```
$ ~ cfssl version
Version: dev
Runtime: go1.14
```

Para instalar el binario `kubectl` en OS X, podemos hacer lo siguiente:

```
$ curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https:// \
  storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/darwin/amd64/kubectl"
$ chmod +x ./kubectl
$ sudo mv ./kubectl /usr/local/bin/kubectl
```

Este es el comando para instalarlo en Linux:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s https:// \
  storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl
$ chmod +x ./kubectl
$ sudo mv ./kubectl /usr/local/bin/kubectl
```

Puede comprobar que el cliente `kubectl` funciona correctamente al ejecutar lo siguiente:

```
$ kubectl version --client
Client Version: version.Info{Major:"1", Minor:"17", GitVersion:"v1.17.3",
GitCommit:"06ad960bfd03b39c8310aaef92d1e7c12ce618213", GitTreeState:"clean",
BuildDate:"2020-02-11T18:14:22Z", GoVersion:"go1.13.6", Compiler:"gc",
Platform:"darwin/amd64"}
```

Ahora estamos listos para crear nuestros nodos de controlador con Ansible. Migré al directorio del manual de Ansible (*Chapter5/Ansible-Playbooks*) y ejecute el siguiente comando:

```
$ ansible-playbook -vi hosts controllers.yaml
```

Esto tardará en ejecutarse, pero al finalizar debería ver el siguiente resultado:

```
PLAY RECAP ****
inventory_hostname=controller0 : ok=43    changed=39    unreachable=0      failed=0      skipped=4
  rescued=0    ignored=0
inventory_hostname=controller1 : ok=22    changed=19    unreachable=0      failed=0      skipped=7
  rescued=0    ignored=0
inventory_hostname=controller2 : ok=22    changed=19    unreachable=0      failed=0      skipped=7
  rescued=0    ignored=0
```

Esto finaliza la configuración de los nodos de controlador. Esencialmente, con el manual *controllers.yaml*, hemos arrancado los nodos de controlador y etcds. El manual de Ansible contiene instrucciones paso a paso, que se explican por sí mismas, para arrancar los nodos de controlador para el clúster. Le recomendamos encarecidamente que siga los pasos del manual para la ruta de ejecución a fin de entender los detalles esenciales.

## Uso de Ansible para implementar y configurar los nodos de trabajo de Kubernetes

Ahora que los nodos del controlador se han implementado, podemos implementar los nodos de trabajo mediante el archivo *workers.yaml*:

```
$ ansible-playbook -vi hosts workers.yaml
.
.

PLAY RECAP ****
inventory_hostname=worker0 : ok=31    changed=29    unreachable=0      failed=0      skipped=20
  rescued=0    ignored=0
inventory_hostname=worker1 : ok=23    changed=21    unreachable=0      failed=0      skipped=23
  rescued=0    ignored=0
inventory_hostname=worker2 : ok=23    changed=21    unreachable=0      failed=0      skipped=23
  rescued=0    ignored=0
```

Esto implementará y configurará los nodos de trabajo con los binarios de Kubernetes necesarios.

## Configuración del enrutamiento y la red del pod

Todavía tenemos que configurar el enrutamiento y la red del pod para nuestro clúster. Para ello, primero tenemos que configurar las redes entre los pods al migrar al directorio *Chapter5/Kubernetes\_Cluster/G-K8S-PodsNetwork* y ejecutar `terraform init`:

```
$ terraform init
```

Una vez que hayamos inicializado Terraform en el directorio, podemos ejecutar `terraform apply`:

```
$ terraform apply
.
.

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

route_table_id = /subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/ \
  K8Scluster/providers/Microsoft.Network/routeTables/k8s-pod-router
```

Esto configura la tabla de rutas del pod en Azure. Ahora podemos finalmente crear las rutas migrando al directorio final, *Chapter5/Kubernetes\_Cluster/H-k8S\_Route-creation*, e inicializando Terraform como antes. Una vez inicializado, podemos ejecutar `terraform apply`:

```

$ terraform apply
.
.
.

Plan: 3 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

module.route0.azure_rm_route.generic_routes: Creating...
module.route1.azure_rm_route.generic_routes: Creating...
module.route0.azure_rm_route.generic_routes: Creation complete after 4s [id=/subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/K8Scluster/providers/Microsoft.Network/routeTables/k8s-pod-router/routes/k8s-pod-router0]
module.route2.azure_rm_route.generic_routes: Creating...
module.route1.azure_rm_route.generic_routes: Creation complete after 5s [id=/subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/K8Scluster/providers/Microsoft.Network/routeTables/k8s-pod-router/routes/k8s-pod-router1]
module.route2.azure_rm_route.generic_routes: Creation complete after 4s [id=/subscriptions/b5624140-9087-4311-a04b-3b16a2e84792/resourceGroups/K8Scluster/providers/Microsoft.Network/routeTables/k8s-pod-router/routes/k8s-pod-router2]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

```

Esto completa la configuración de nuestra red para el clúster de Kubernetes.

## Generar el archivo kubeconfig para el acceso remoto y la validación del clúster

Por último, tenemos que descargar el archivo kubeconfig para el clúster que acabamos de crear y probar si la configuración del clúster se realizó correctamente. Vuelva al directorio de Ansible, *Chapter5/Ansible-Playbooks*, y ejecute lo siguiente:

```

$ ansible-playbook -vi hosts remote_access_k8s.yaml
.

.

PLAY RECAP ****
inventory_hostname=controller0 : ok=6    changed=5    unreachable=0    failed=0    skipped=0
  rescued=0   ignored=0
inventory_hostname=controller1 : ok=1    changed=0    unreachable=0    failed=0    skipped=0
  rescued=0   ignored=0
inventory_hostname=controller2 : ok=1    changed=0    unreachable=0    failed=0    skipped=0
  rescued=0   ignored=0

```

Esto descargará el archivo kubeconfig y lo almacenará localmente para que pueda acceder e interactuar con el clúster. Compruebe si puede hacer esto al emitir `kubectl get nodes` desde su máquina local de la siguiente manera:

```

$ kubectl get nodes
NAME      STATUS    ROLES     AGE      VERSION
worker0   Ready     <none>   27m     v1.17.3

```

```
worker1 Ready <none> 27m v1.17.3
worker1 Ready <none> 27m v1.17.3
```

El resultado enumerará los nodos de trabajo de su clúster. Ahora puede usar este clúster para experimentar y explorar las características que abarcamos en este capítulo. Por supuesto, este no es un clúster listo para la producción, pero le permite comprender cómo crear, ejecutar y mantener un clúster de Kubernetes desde cero.

Ahora que sabe cómo lograrlo de la manera difícil (semiautomatizada), veamos la manera preferida y sin esfuerzo para ejecutar un clúster de Kubernetes de producción en Azure.

## Azure Kubernetes Service

Como ha visto, el proceso de administrar y mantener un clúster de Kubernetes es complejo. Desde una perspectiva operativa, no se trata solo de ejecutar un clúster. Se trata de la seguridad, el rendimiento, el registro, la actualización del clúster, la aplicación de parches y mucho más. Para reducir este gran esfuerzo, Azure ofrece una solución administrada llamada *Azure Kubernetes Service o AKS*.

AKS hace que el proceso de implementación y administración de un clúster de Kubernetes en Azure sea tremadamente sencillo. La solución de Kubernetes administrada por Azure elimina gran parte del proceso de administración de Kubernetes para que lo único que tenga que hacer sea administrar su aplicación. AKS ofrece varias ventajas, entre las que se incluyen las siguientes:

- Azure administra completamente la supervisión del estado de los clústeres.
- Las tareas de mantenimiento de los nodos subyacentes son responsabilidad de Azure.
- Azure administra y mantiene el plano de control, y usted como usuario solo tiene que administrar los nodos de agente.
- AKS le permite integrarse con Azure Active Directory y la API de RBAC de Kubernetes para el control de los accesos y la seguridad.
- Todos los registros de la aplicación y el clúster se almacenan en el área de trabajo de Azure Log Analytics.
- El escalador automático horizontal de pod (HPA) y el escalador automático del clúster son fáciles de usar con respecto a configurar un clúster de Kubernetes desde cero y ocuparse de los permisos de recursos subyacentes.
- AKS ofrece compatibilidad fluida con el volumen de almacenamiento.

Ahora que conoce las ventajas de AKS, vamos a crear un clúster de AKS. Hay dos formas sencillas de hacerlo: usar el portal de Azure (el enfoque manual) o usar Terraform (el enfoque automatizado).

Puede navegar por la consola de Azure y comenzar a implementar su clúster de AKS haciendo clic en el botón Add (Aregar) (Figura 5-2) y luego siguiendo las instrucciones que aparecen.

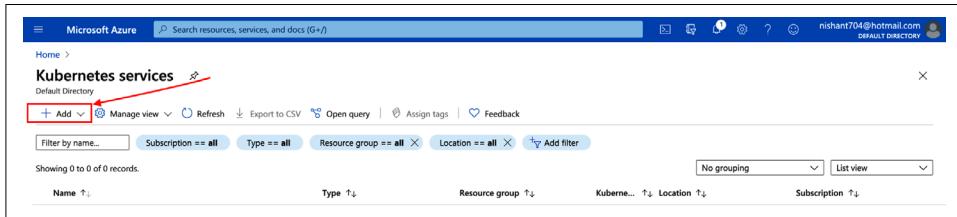


Figura 5-2. Haga clic en el botón Add para crear un clúster de AKS

O bien, en Terraform puede ir al directorio de AKS del repositorio */cloud\_native\_azure/Chapter5/AKS* y ejecutar el siguiente comando:

```
$ terraform apply
```

Haga clic en Apply (Aplicar) y, en unos minutos, debería tener un clúster de Kubernetes de tres nodos en funcionamiento (Figura 5-3).

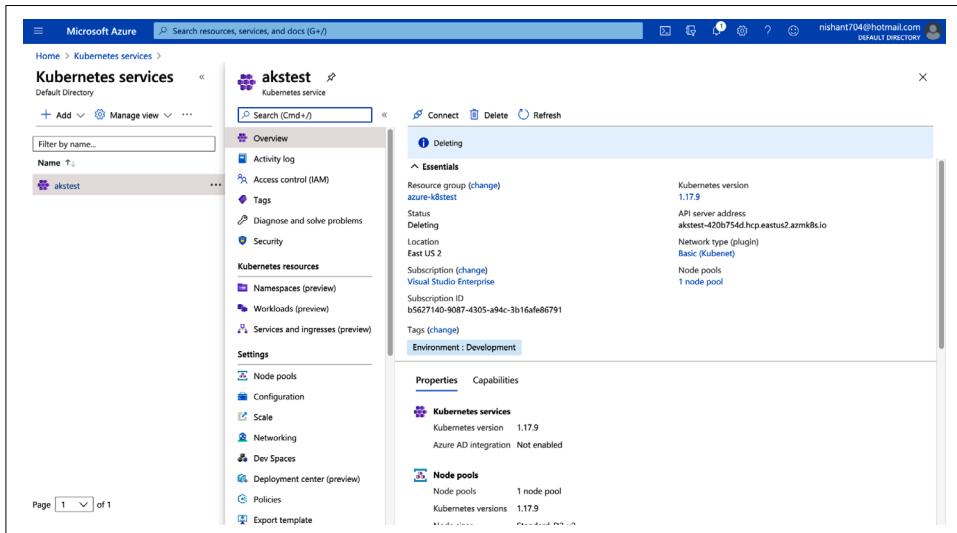


Figura 5-3. Clúster de AKS creado a través de Terraform

Para descargar el archivo kubeconfig, puede usar el comando de la CLI de Azure de su sistema local para obtener credenciales de acceso para el clúster de Kubernetes administrado de la siguiente manera:

```
az aks get-credentials --resource-group azure-k8stest --name akstest --file config
```

Su archivo `kubeconfig` se descargará con el nombre `config`. Muévalo a la carpeta. `kube`. Ahora puede usar este clúster tal como usaría el clúster que creó de la manera difícil:

```
$ ~ kubectl get nodes
NAME                      STATUS  ROLES   AGE    VERSION
aks-agentpool-42554519-vmss000000  Ready   agent   29m   v1.17.9
aks-agentpool-42554519-vmss000001  Ready   agent   29m   v1.17.9
aks-agentpool-42554519-vmss000002  Ready   agent   29m   v1.17.9
```

Si bien puede crear prácticamente cualquier recurso e implementar aplicaciones en este punto, por supuesto que este no es un clúster de nivel de producción. Hay muchos recursos en línea<sup>1</sup> que pueden guiarlo para crear clústeres de nivel de producción.

Con estos conocimientos, estamos listos para avanzar y aprender sobre Helm, que actúa principalmente como una herramienta para agilizar la instalación y administración de las aplicaciones de Kubernetes.

## Implementación de aplicaciones y servicios mediante Helm: un administrador de paquetes para Kubernetes

Probablemente, ya tiene bastante claro que cuando se trata de Kubernetes, debe encargarse de demasiados elementos. Aunque puede usar un servicio administrado, como Microsoft AKS, es difícil mantener las implementaciones y ReplicaSets a medida que se expanden sus requisitos.

Helm es un administrador de paquetes para Kubernetes que le permite recuperar, implementar y administrar aplicaciones. Básicamente, Helm se encarga del trabajo de mantener grandes archivos YAML que incluyen la información de pods, ReplicaSets, servicios, configuración de RBAC y mucho más. Helm también ofrece reveriones, información del historial de versiones y enganches de prueba integrados, lo que hace que sea muy fácil administrar el ciclo de vida completo de sus aplicaciones en un clúster de Kubernetes.

En este libro, estamos usando la versión estable de Helm, v3.3.1. Las versiones anteriores (es decir, v2) solían tener un componente del lado del servidor llamado *Tiller*, que debía implementarse en el clúster de Kubernetes. Tiller poseía un potencial riesgo de seguridad debido a los privilegios de usuario más amplios y la sobrecarga de administración, lo que inicialmente provocó menos tracción por Helm en la comunidad de usuarios. Con la introducción de la API de RBAC, ahora Helm contiene solo binarios del lado del cliente y Tiller se eliminó por completo. En la Figura 5-4 se muestra el funcionamiento básico del administrador de paquetes Helm.

---

<sup>1</sup> Consulte <https://kubernetes.io/docs/setup/production-environment/>, <https://kubernetes.io/docs/setup/production-environment/tools/>, y <https://github.com/kubernetes/kops>.

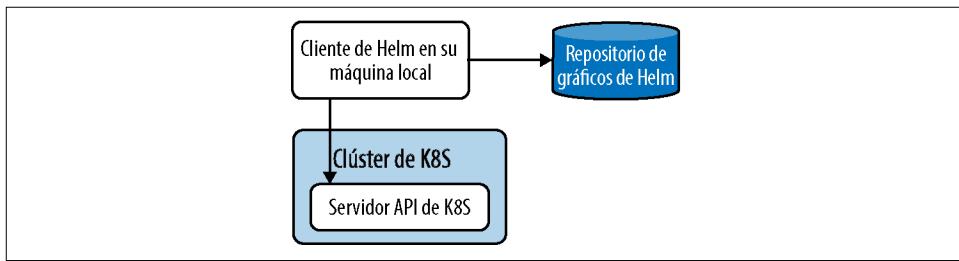


Figura 5-4. Repositorio de gráficos de Helm e interacción con el cliente

## Aspectos básicos de Helm

El cliente de Helm se instala como una utilidad de línea de comandos para el usuario final. La principal responsabilidad del cliente es ayudar en el desarrollo de gráficos y administrar repositorios y lanzamientos. Helm consta de los siguientes componentes:

### Gráfico

Los gráficos son los paquetes de Kubernetes que Helm administra. Los gráficos constan de toda la información necesaria para crear una aplicación en Kubernetes, incluidas las definiciones de recursos necesarias para ejecutar una aplicación o un servicio en un clúster de Kubernetes.

### Repositorio

Esta es la base de datos donde se almacenan todos los gráficos de Helm.

### Config

Esta es la configuración que se puede combinar en un gráfico empaquetado para crear un objeto que puede lanzarse.

### Lanzamiento

Los lanzamientos son una forma de realizar seguimiento de todas las aplicaciones que Helm instala en el clúster de Kubernetes. Se puede instalar un solo gráfico varias veces, y cada instalación nueva crea un nuevo lanzamiento.

## Instalación y administración de Helm

Puede descargar la versión más reciente del cliente Helm para su máquina desde sus repositorios oficiales en <https://github.com/helm/helm/releases>.

Después de instalar Helm, tenemos que agregar un repositorio de gráficos desde donde se puedan descargar los paquetes. Google ofrece el repositorio de gráficos de Kubernetes y se puede agregar de la siguiente manera:

```
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/
```

Desde una perspectiva bastante sencilla, Helm extrae gráficos o paquetes desde un repositorio central y luego los instala/lanza en el clúster de Kubernetes.

## Búsqueda de repositorios de Helm

Puede buscar repositorios de Helm con el comando `helm search` de dos maneras diferentes. La primera es buscar el repositorio que agregó localmente (como en el paso anterior, cuando agregamos un repositorio `stable`):

```
$ helm search repo stable
NAME          CHART VERSION   APP VERSION
DESCRIPTION
stable/acs-engine-autoscaler    2.2.2        2.1.1
  DEPRECATED Scales worker nodes within agent pools
stable/aerospike                0.3.3        v4.5.0.5
  A Helm chart for Aerospike in Kubernetes
stable/airflow                  7.6.0        1.10.10
  Airflow is a platform to programmatically autho...
stable/ambassador               5.3.2        0.86.1      DEPRECATED A Helm chart for...
stable/anchore-engine           1.6.9        0.7.2      Anchore container analysis...
stable/apm-server               2.1.5        7.0.0      The server receives data...
stable/ark                      4.2.2        0.10.2      DEPRECATED A Helm...
.
.
.
```

Esta búsqueda se realiza en los datos locales de su máquina.

La otra forma de buscar repositorios de Helm es mediante la búsqueda en Helm Hub. Helm Hub consta de varios repositorios públicos disponibles, por lo que si no encuentra un gráfico en su repositorio local, siempre puede buscarlo en el Hub. A continuación, buscamos todos los gráficos de "kafka":

```
$ ~ helm search hub kafka
URL          CHART VERSION   APP VERSION
DESCRIPTION
https://hub.helm.sh/charts/kafkaesque/imagepuller  1.0.0        1.0
  Pull container images to your nodes so that the...
https://hub.helm.sh/charts/kafkaesque/pulsar        1.0.26       1.0
  Apache Pulsar Helm chart for Kubernetes
https://hub.helm.sh/charts/kafkaesque/pulsar-mo...  0.1.5        1.0
  A Helm chart for the Pulsar Monitor application
https://hub.helm.sh/charts/kafkaesque/teleport     1.0.0        1.0
  Teleport Community
https://hub.helm.sh/charts/bitnami/kafka           11.8.4       2.6.0
  Apache Kafka is a distributed streaming platform.
https://hub.helm.sh/charts/stable/kafka-manager    2.3.1        1.3.3.22
  A tool for managing Apache Kafka.
https://hub.helm.sh/charts/touk/hermes            0.3.1        1.5.2
  A Helm chart for Kubernetes of Hermes, a reliab...
```

## Instalación de un gráfico de Helm en Kubernetes

Para instalar un gráfico, puede ejecutar el comando `helm install` de la siguiente manera:

```
$ ~ helm install my-first-server stable/tomcat
NAME: my-first-server
LAST DEPLOYED: Thu Sep 10 00:04:17 2020
NAMESPACE: default
STATUS: deployed
```

```

REVISION: 1
TEST SUITE: None
NOTES:
1. Get the application URL by running these commands:
  NOTE: It may take a few minutes for the LoadBalancer IP to be available.
  You can watch the status of by running 'kubectl get svc -w my-first-server-tomcat'
  export SERVICE_IP=$(kubectl get svc --namespace default my-first-server-tomcat -o \
    jsonpath='{.status.loadBalancer.ingress[0].hostname}')
  echo http://$SERVICE_IP:

```

Aquí intentamos implementar un servidor Tomcat, y el nombre del lanzamiento que elegimos es *my-first-server*. Con kubectl, puede comprobar los recursos que se crearon para el gráfico de Tomcat:

```

$ ~ kubectl get svc --namespace default my-first-server-tomcat
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
my-first-server-tomcat   LoadBalancer   10.32.0.3    <pending>     80:31495/TCP   56s

```

## Cambiar los valores predeterminados del gráfico

A veces, es posible que desee personalizar un gráfico antes de instalarlo. Para cambiar un valor de un gráfico predeterminado, primero debe ver el valor mediante `helm show values`:

```

$ helm show values stable/tomcat
# Default values for the chart.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.
replicaCount: 1

image:
  webarchive:
    repository: ananwaresystems/webarchive
    tag: "1.0"
  tomcat:
    .
    .
    .
resources: {}
# limits:
#   cpu: 100m
#   memory: 256Mi
# requests:
#   cpu: 100m
#   memory: 256Mi

nodeSelector: {}

tolerations: []

affinity: {}

```

Para cambiar el valor, puede utilizar el argumento `--values` o el argumento `--set`. La ventaja de usar el primero es que puede especificar un archivo YAML que puede pasar al instalar un gráfico. Esto reemplaza los valores del gráfico. Por ejemplo, en el gráfico anterior, si queremos cambiar los valores predeterminados, podemos crear un archivo YAML de la siguiente manera y guardarlo como *custom\_vals.yaml*:

```
limits:  
  cpu: 200m  
  memory: 256Mi  
requests:  
  cpu: 100m  
  memory: 256Mi
```

Para instalar los valores cambiados, puede hacer lo siguiente:

```
$ helm install --values custom_vals.yaml new-v2-install stable/tomcat
```

En la siguiente sección, analizaremos cómo administrar los lanzamientos en Helm.

## Administración de lanzamientos de Helm

En la terminología de Helm, un lanzamiento es una instancia de un gráfico que se ejecuta en un clúster de Kubernetes. Se puede instalar un gráfico muchas veces en el mismo clúster y cada vez que se instala, se crea un nuevo lanzamiento. En esta sección, le mostraremos cómo puede administrar los lanzamientos de Helm.

### Comprobación de un lanzamiento

Para encontrar el estado del lanzamiento, puede usar `helm status`, que le proporcionará detalles, como el número de revisión y cuándo se implementó por última vez:

```
$ helm status my-first-server  
NAME: my-first-server  
LAST DEPLOYED: Thu Sep 10 00:04:17 2020  
NAMESPACE: default  
STATUS: deployed  
REVISION: 1  
TEST SUITE: None  
NOTES:  
1. Get the application URL by running these commands:  
    NOTE: It may take a few minutes for the LoadBalancer IP to be available.  
    You can watch the status of by running 'kubectl get svc -w my-first-server-tomcat'  
    export SERVICE_IP=$(kubectl get svc --namespace default my-first-server-tomcat -o \  
        jsonpath='{.status.loadBalancer.ingress[0].hostname}')  
    echo http://$SERVICE_IP:  
$ ~ helm show values stable/tomcat  
# Default values for the chart.  
# This is a YAML-formatted file.  
# Declare variables to be passed into your templates.  
replicaCount: 1
```

### Actualizar un lanzamiento

Puede utilizar `helm upgrade` para actualizar los gráficos instalados cuando se lance una nueva versión o si ha realizado un cambio de configuración:

```
$ helm upgrade --values new_vals.yaml new-v2-install stable/tomcat
```

Esto actualizará el `new-v2-install` con el mismo gráfico (`stable/tomcat`), pero puede actualizar nuevos valores en el archivo `new_vals.yaml`.

## Revertir un lanzamiento

Si necesita revertir un gráfico implementado (lanzamiento), simplemente use la opción `helm rollback` como se indica a continuación:

```
$ ~ helm rollback my-first-server 1
Rollback was a success! Happy Helming!
$ ~
```

Aquí revertimos `my-first-server` a la versión 1.

## Desinstalar un lanzamiento

Para eliminar permanentemente un lanzamiento del clúster, puede usar `helm uninstall`:

```
$ ~ helm uninstall my-first-server
release "my-first-server" uninstalled
```

## Creación de gráficos para sus aplicaciones

Para crear un gráfico para su aplicación, siga el formato de empaquetado predeterminado que proporciona Helm. Puede usar los gráficos para implementar un pod simple o una pila de aplicaciones web completa que contenga servidores HTTP, bases de datos y mucho más. Los gráficos contienen una colección de archivos dentro de un directorio. La estructura del directorio es la siguiente, donde el nombre del directorio de nivel superior (en este caso, `nginx`) es el nombre del gráfico:

```
~ nginx/
  Chart.yaml
  LICENSE
  README.md
  values.yaml
  values.schema.json
  charts/
  crds/
  templates/
```

En el directorio, Helm espera que el archivo tenga la misma estructura y nombres, ya que los reserva. Cada elemento del directorio tiene un propósito específico:

- `Chart.yaml` contiene toda la información sobre el gráfico en formato YAML.
- `LICENSE` es un archivo opcional de texto sin formato que contiene la licencia para el gráfico.
- `README.md` es un archivo README legible para personas.
- `values.yaml` contiene todos los valores de configuración predeterminados para el gráfico.
- `values.schema.json` es un archivo de esquema JSON opcional que impone una estructura en `values.yaml`.
- `charts/` incluye todos los gráficos dependientes de los que depende el gráfico actual.
- `crds/` contiene definiciones de recursos personalizados.
- `templates/` es el directorio que contiene las plantillas, que se pueden combinar con los valores para generar archivos de manifiesto de Kubernetes.

El archivo *Chart.yaml* es necesario para crear el gráfico y contiene los siguientes campos:

```
apiVersion: The chart API version (required)
name: The name of the chart (required)
version: A SemVer 2 version (required)
kubeVersion: A SemVer range of compatible Kubernetes versions (optional)
description: A single-sentence description of this project (optional)
type: The type of the chart (optional)
keywords:
  - A list of keywords about this project (optional)
home: The URL of this projects home page (optional)
sources:
  - A list of URLs to source code for this project (optional)
dependencies: # A list of the chart requirements (optional)
  - name: The name of the chart (nginx)
    version: The version of the chart ("1.2.3")
repository: The repository URL ("https://example.com/charts") or alias ("@repo-name")
condition: (optional) A yaml path that resolves to a boolean, used for
  enabling/disabling charts (e.g. subchart1.enabled )
tags: # (optional)
  - Tags can be used to group charts for enabling/disabling together
enabled: (optional) Enabled bool determines if chart should be loaded
import-values: # (optional)
  - ImportValues holds the mapping of source values to the parent key to be imported.
    Each item can be a string or pair of child/parent sublist items.
alias: (optional) Alias to be used for the chart. Useful when you have to add the same
  chart multiple times
maintainers: # (optional)
  - name: The maintainers name (required for each maintainer)
    email: The maintainers email (optional for each maintainer)
    url: A URL for the maintainer (optional for each maintainer)
icon: A URL to an SVG or PNG image to be used as an icon (optional).
appVersion: The version of the app that this contains (optional). This needn't be SemVer.
deprecated: Whether this chart is deprecated (optional, boolean)
annotations:
  example: A list of annotations keyed by name (optional).
```

## Usar Helm para crear y administrar gráficos

Puede crear cada archivo en su directorio de gráficos manualmente o puede usar Helm para crear un proyecto de la siguiente manera:

```
$ helm create examplechart
Creating examplechart
```

A continuación, puede editar los archivos en el directorio de gráficos y empaquetarlos en un archivo de gráfico:

```
$ helm package examplechart
Archived examplechart.1--.tgz
```

Una vez que haya empaquetado su gráfico, puede usarlo con `helm install` para crear un lanzamiento.

## Resumen

En este capítulo, aprendió a crear un clúster de Kubernetes, tanto manualmente desde cero con Azure como mediante el uso de Azure AKS como servicio administrado. También exploramos Helm, que actúa como administrador de paquetes de Kubernetes y lo ayuda a crear y administrar sus aplicaciones con una sola plantilla.

Antes de continuar, queremos enfatizar que Kubernetes es difícil de administrar sin ayuda (especialmente en la producción), y definitivamente no es una fórmula mágica para aliviar todo el trabajo que conlleva administrar aplicaciones en entornos nativos de la nube. Debido a la complejidad subyacente que tiene administrar un clúster de Kubernetes, es evidente que un enfoque como AKS es una buena manera de administrar los servicios que se ejecutan en clústeres de Kubernetes. Con AKS, no tiene que preocuparse por la infraestructura subyacente ni las partes móviles del ecosistema, ya que se encarga de la mayoría de los puntos críticos, como la alta disponibilidad y la redundancia del clúster de Kubernetes.

Dicho esto, pasemos ahora al [Capítulo 6](#), que trata sobre el desarrollo de la observabilidad en sistemas distribuidos y cómo ayudar a que las aplicaciones nativas de la nube sean más confiables.

## CAPÍTULO 6

# Observabilidad: Seguir las huellas

La rápida evolución de los sistemas nativos de la nube ha presentado complejidades en torno al aprovisionamiento de la infraestructura, la implementación de infraestructura y la administración de software. Las aplicaciones se están diseñando para ser más resilientes, lo que requiere comprender con más profundidad la infraestructura subyacente del entorno de la nube, incluidos los modos de error y los cuellos de botella. Tener visibilidad de la pila de aplicaciones se vuelve aún más importante debido a la gran cantidad de nuevas piezas móviles y cambios en el diseño en las aplicaciones de hoy.

En este capítulo, presentaremos el concepto de observabilidad y explicaremos por qué es necesaria en el mundo nativo de la nube actual. Aprenderá sobre los tres pilares de la observabilidad (registros, métricas y seguimientos) y obtendrá información sobre cómo trabajan juntos para crear sistemas observables. También analizaremos varias herramientas nativas de la nube bien conocidas que se pueden usar para crear un sistema observable y demostrar cómo la observabilidad es un superconjunto de supervisión. Por último, mostraremos cómo puede aprovechar Azure como una plataforma en la nube para obtener observabilidad en las aplicaciones, la infraestructura subyacente y la pila de redes.

## Introducción a la observabilidad

Lo primero que debe entender sobre la observabilidad es la palabra en sí. El término *observabilidad* se originó en el mundo de la teoría del control en matemáticas, que se ocupa principalmente con la creación de una forma de controlar los *sistemas dinámicos* mediante un controlador con la ayuda de la retroalimentación del sistema. En la teoría del control, la observabilidad es la capacidad de medir el estado interno de un sistema a partir del conocimiento de los resultados externos. Con respecto a la ingeniería de software, la observabilidad significa conocer el estado *exacto* de una aplicación a partir de sus resultados. Al diseñar su aplicación e infraestructura para que sea observable, básicamente podrá descubrir *por qué* se produce un problema y *cómo* puede corregirlo. El objetivo subyacente

de hacer que un sistema sea observable es permitir que un ingeniero vea claramente un problema, desde su causa hasta su efecto.

La observabilidad es aún más relevante en la infraestructura nativa de la nube actual debido al entorno acelerado y distribuido en el que el software y los sistemas finalmente podrían fallar. El auge de los microservicios, los contenedores y la informática sin servidor ha dado lugar a una complicada malla de servicios interconectados que realizan enormes cantidades de llamadas entre sí. Cuando estas redes de servicios interconectados tienen un mal comportamiento, es casi imposible descubrir la causa raíz del problema y, al mismo tiempo, evitar la interrupción del negocio. Resulta fundamental tener una forma rápida de identificar un problema en el entorno subyacente de los servicios complejos a fin de que el negocio no se vea perjudicado. La observabilidad aborda esto al permitirle hacer preguntas sobre sus sistemas y obtener respuestas rápidamente.

Antes de que hablemos sobre cómo la observabilidad ayuda en los entornos distribuidos modernos, veamos lo que constituye un sistema observable y hablemos sobre el principio detrás del término y la filosofía detrás de la idea de desarrollar la observabilidad en los sistemas.

## Observabilidad: más que tres pilares

La idea fundamental detrás de la observabilidad es comprender mejor una aplicación a través de información significativa. A medida que las aplicaciones se vuelven más efímeras y distribuidas, se vuelve clave definir los principios rectores de un sistema observable. Los profesionales de la nube han definido *registros*, *métricas* y *seguimientos* como los tres verticales principales, o pilares, fundamentales para crear un sistema observable moderno. Vamos rápidamente estos pilares y por qué son importantes.

### Métricas

Las métricas representan datos numéricos medidos en un intervalo de tiempo. Por lo general, las métricas se recopilan mediante la ejecución de un agente en un host, que envía periódicamente los datos a un almacén de datos central y, más tarde, agrega los datos para representarlos en paneles. Luego, las métricas se usan para crear sistemas de alerta, ya que las métricas se almacenan en una base de datos de series temporales. Un ejemplo de una métrica son las consultas por segundo (qps).

### Registros

Los registros representan los eventos que se producen en un sistema (aplicación e infraestructura). Contienen detalles del estado interno y los eventos. En general, se escriben y almacenan en el sistema de aplicaciones en un formato estructurado que se puede analizar fácilmente. Una buena arquitectura de registros es la herramienta más fundamental para obtener la mayor parte de la información de un sistema y, a menudo, es la primera herramienta de depuración que usan los desarrolladores de software. A continuación, se muestra un ejemplo de registros simples de acceso del servidor de Apache:

```
10.185.248.71 - - [09/Jan/2015:19:12:06 +0000] 808840 "GET /inventoryService/inventory/ \
purchaseItem?userId=20253471&itemId=23434300 HTTP/1.1" 500 17 "-" \
"Apache-HttpClient/4.2.6 (java 1.5)"
```

## Seguimientos

Por último, pero no menos importante, los seguimientos son la solución moderna frente al problema de aprender cómo se comporta un solo evento en un sistema distribuido. Los seguimientos le ayudan a identificar la cantidad de procesamiento (es decir, la latencia) que se realiza en cada capa de la pila. En el mundo de los microservicios y los sistemas distribuidos, los seguimientos proporcionan información valiosa a los eventos mediante la creación de un objeto visual de las llamadas distribuidas correlacionadas entre los servicios.

La Figura 6-1 muestra seguimientos simples.



Figura 6-1. Ejemplo de aplicación que muestra seguimientos

Sin embargo, el hecho de que haya creado un sistema que incluya registros, métricas y seguimientos no significa necesariamente que el sistema sea observable. Los tres pilares tienen sus ventajas y desventajas, que analizaremos en el resto del capítulo. Además, de forma individual, son solo el punto de partida, ya que le permiten investigar un problema, pero no abordan directamente el caso de uso o las necesidades comerciales de su empresa. Para aprovechar los pilares, debe usarlos todos juntos con los factores que afectan a su negocio. Si recopila datos de métricas por separado de los registros y seguimientos, puede perder fácilmente el contexto y terminar observando solo un sistema subyacente de forma aislada. La observabilidad con respecto a un enfoque unificado significa recopilar y preservar el contexto subyacente, junto con toda su riqueza y dimensionalidad.

## Observabilidad: Un superconjunto de supervisión

La supervisión se centra principalmente en la recopilación pasiva de eventos, registros y métricas de un sistema. En general, supervisa un servicio crítico, que cree que podría tener problemas porque ya ha experimentado errores con el servicio o porque está al tanto de los modos de error del servicio. La supervisión adopta un enfoque proactivo al ayudarlo con las alertas y la planificación de la capacidad, mientras que la observabilidad adopta un enfoque más reactivo al permitirle examinar la pila de aplicaciones más de cerca para ver cuál es la falla y cuál es el motivo.

Veamos un ejemplo. Imagine que un desarrollador de su empresa cambia una ruta de código crítico mientras intenta agregar una nueva característica en una aplicación emblemática. Lamentablemente, de forma accidental introduce un error que luego

provoca una interrupción. En tal situación, sus sistemas de supervisión podrán mostrar claramente el aumento de la tasa de error de un servicio, pero no podrán profundizar más. Sin embargo, tener la observabilidad integrada en la pila puede ayudarlo a desglosar rápidamente el problema (por ejemplo, un problema con un clúster de Redis) y luego los seguimientos pueden conducirlo directamente a la causa raíz.

Es importante reiterar que no hay nada de malo con sus sistemas de supervisión en este escenario. Están haciendo lo que están diseñados para hacer: le dicen cuando algo está mal. La observabilidad es lo que le ayuda a descubrir qué salió mal.

También es importante tener en cuenta que la observabilidad no es una alternativa a la supervisión. Más bien, se trata de un superconjunto de supervisión, ya que proporciona una visión general de alto nivel de los servicios junto con información detallada sobre diversos eventos dentro de un sistema distribuido. Además, en un entorno de nube distribuido donde se ejecutan varios servicios e interactúan entre sí, la tarea de encontrar el error se vuelve extremadamente difícil si el sistema no se instrumenta para exponer el error correcto. La observabilidad maneja tales escenarios identificando la cadena de problemas a medida que la solicitud viaja a través de los servicios.

## Desarrollo impulsado por la observabilidad

Como hemos analizado, las aplicaciones nativas de la nube son, ante todo, distribuidas y se basan en microservicios, lo que las hace inherentemente complejas de mantener y administrar. Cualquier persona que ejecute una aplicación de nivel de producción también se dará cuenta de que cuando las aplicaciones empiezan a tener problemas, resulta casi imposible detectar qué parte móvil en particular es la causa principal. A veces, los errores se detectan con la ayuda de las configuraciones tradicionales de registros y supervisión, que pueden apuntar en la dirección correcta teniendo en cuenta la complejidad de la pila de aplicaciones. Sin embargo, a medida que sus servicios evolucionan, no puede saber con certeza todos los modos de error de su aplicación o la infraestructura subyacente.

El desarrollo impulsado por la observabilidad es un esfuerzo para solucionar el problema que representa encontrar las incógnitas en los entornos de producción al fomentar la incorporación de aspectos de observabilidad en las primeras etapas del ciclo de vida de desarrollo de software. Además, recomienda la instrumentación del código de la aplicación durante la fase de diseño del software, de modo que sea más fácil agregar instrumentación en un solo lugar y no en muchos lugares en una etapa posterior. También tiene como objetivo cerrar la brecha entre los desarrolladores y el personal operativo al permitir que los desarrolladores sean dueños del código y depuren fácilmente en los entornos de producción. Los equipos de ingeniería no solo pueden diagnosticar errores y fallas en los inicios de la producción, sino que también ayuda a estos equipos a medir y analizar el rendimiento operacional y empresarial.

En las próximas secciones, analizaremos varias formas en las que las aplicaciones y la infraestructura nativa de la nube moderna se pueden hacer observables. Profundizaremos en cada uno de los tres pilares y veremos cuál es la forma preferida de crear sus sistemas en entornos nativos de la nube, comenzando con las métricas.

# Supervisión de métricas con Prometheus en un mundo nativo de la nube

Como analizamos en el [Capítulo 4](#), Kubernetes ha cambiado enormemente la forma en que se crean e implementan las aplicaciones modernas. Los microservicios y las arquitecturas orientadas a servicios han transformado el panorama de los entornos nativos de la nube a medida que aumenta la velocidad de las incorporaciones e implementaciones de características. Con el cambio hacia estas arquitecturas modernas, el enfoque tradicional de supervisión no puede mantenerse al día con el ritmo actual de estas arquitecturas que giran en torno a la infraestructura efímera, varias regiones y zonas de disponibilidad, y a veces incluso abarcan varias nubes. Todas estas capas de complejidad añadidas también abren la puerta a modos de falla nuevos y desconocidos con los que el enfoque de supervisión tradicional realmente no puede mantenerse al día.

Para enfrentar el problema que presenta supervisar las aplicaciones e infraestructura nativas de la nube moderna, tenemos que volver a revisar uno de los principales pilares de la observabilidad: las métricas. Como examinamos, las métricas son una de las áreas clave que brindan una inmensa cantidad de información sobre el estado de la aplicación. Cuando se trata de comprender por completo las aplicaciones nativas de la nube, el conjunto tradicional de métricas a menudo no cumple con el propósito y se debe recopilar un conjunto personalizado de métricas para proporcionar una mejor medida del rendimiento y el estado general de la aplicación. Estas métricas suelen configurarse para que la aplicación las emita y, por lo tanto, proporcionan información más precisa. Aquí es donde se destaca Prometheus.

Prometheus es un sistema de supervisión y alerta de open source que creó originalmente SoundCloud y más tarde Cloud Native Computing Foundation (CNCF) lo incorporó como el segundo proyecto después de Kubernetes. Está diseñado principalmente para aplicaciones nativas de la nube moderna centradas específicamente en el espacio de métricas. Con la llegada de Kubernetes, Prometheus se ha convertido en una de las formas de facto para supervisar las nuevas aplicaciones nativas de la nube que exponen una amplia gama de métricas personalizadas. Prometheus también es más ligero que su predecesor en el espacio de supervisión, ya que funciona en modelos basados en la extracción de exportadores que, básicamente, extraen datos de las aplicaciones y la infraestructura.

## Componentes y arquitectura de Prometheus

Prometheus se creó principalmente con un enfoque en la supervisión de aplicaciones basadas en contenedores que se ejecutan a escala. Las aplicaciones nativas de la nube moderna revelan la variedad de datos que se deben recopilar además de la complejidad de las partes móviles subyacentes, y Prometheus se creó desde cero para abordar este entorno dinámico.

Para apoyar la nueva era de las aplicaciones nativas de la nube, el ecosistema de Prometheus consta de los siguientes cinco componentes principales:

#### *Servidor de Prometheus*

Responsable de extraer las métricas y almacenarlas en una base de datos de serie temporal.

#### *Bibliotecas de cliente*

Se utiliza para instrumentar el código de la aplicación y enviar los datos de métricas extraídos al servidor de Prometheus. Las bibliotecas cliente están disponibles en diferentes lenguajes, incluidos C#, Python, Java y Ruby.

#### *Pushgateway*

Se utiliza para admitir trabajos de corta duración.

#### *Exportadores*

Se ejecutan junto con aplicaciones que no se pueden instrumentar directamente, como HAP-roxy y statsd, para extraer métricas que se pueden enviar al servidor de Prometheus en el formato deseado.

#### *Alertmanager*

Recibe alertas del servidor de Prometheus y expone las notificaciones de forma eficiente. Esto lo hace de distintas maneras coherentes y efectivas:

- Agrupa alertas de un tipo similar, de modo que se envíen menos alertas durante una interrupción importante.
- Puede limitar las notificaciones enviadas al sistema receptor.
- Permite la supresión de notificaciones para las alertas seleccionadas si las alertas relacionadas ya están activadas.
- Puede enviar alertas a una serie de sistemas, como OpsGenie y PagerDuty, entre otros.

La [Figura 6-2](#) muestra cómo se ven estos componentes principales en la arquitectura de Prometheus.

Prometheus almacena todos los datos como una *serie temporal*; es decir, flujos de valores con marcas de tiempo que pertenecen a la misma métrica y el mismo conjunto de dimensiones etiquetadas. Cada serie temporal se identifica de forma única por el *nombre de la métrica* y un par de clave-valor opcional llamado *labels*. Además, las bibliotecas de cliente ofrecen tipos básicos de métricas quad-core: Contador, Medidor, Histograma y Resumen. Hablaremos sobre estos cuatro tipos más adelante en el capítulo y también brindaremos ejemplos de instrumentación.

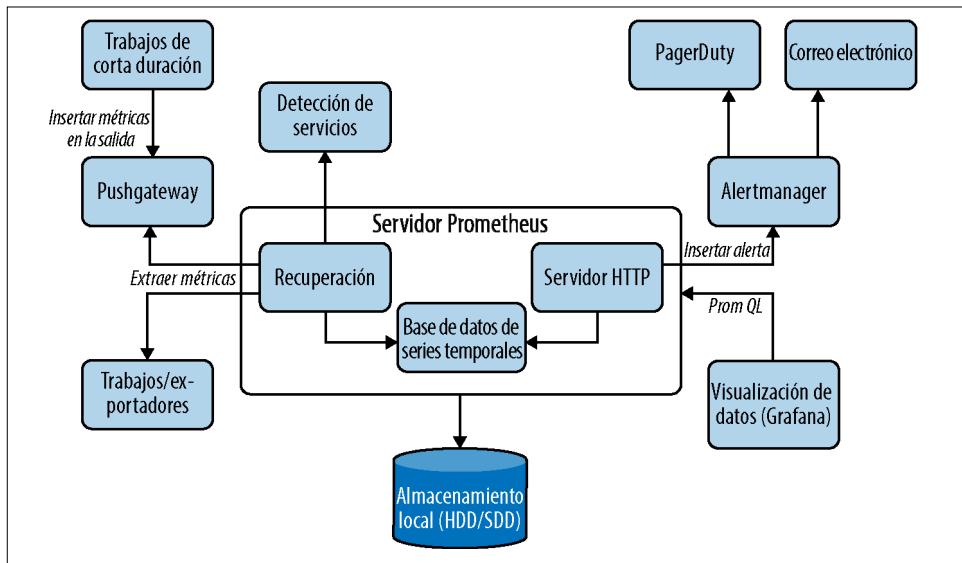


Figura 6-2. Arquitectura de Prometheus



### Trabajos e instancias

En términos de Prometheus, un punto de conexión que se puede extraer se denomina *instancia* y, por lo general, corresponde a un proceso único. Una colección de instancias con el mismo propósito (por ejemplo, un proceso replicado para escalabilidad o confiabilidad) se denomina *trabajo*.

Este es un ejemplo de un trabajo de servidor API con cuatro instancias replicadas:

```

job: api-server
instance 1: 1.2.3.4:5670
instance 2: 1.2.3.4:5671
instance 3: 5.6.7.8:5670
instance 4: 5.6.7.8:5671

```

Vamos a instalar y configurar Prometheus para comprender algunos conceptos clave adicionales en torno a su ecosistema.

## Instalación y configuración de Prometheus

Para instalar y ejecutar Prometheus localmente, primero debe [descargar la versión correcta para su plataforma](#). La versión más reciente de Prometheus en el momento de escribir este capítulo es 2.23. Una vez que haya descargado el archivo comprimido, vaya al directorio y extráigalo con el siguiente comando:

```

$ tar xvfz prometheus-2.23.0.darwin-amd64.tar.gz
$ cd prometheus-2.23.0.darwin-amd64

```

Una vez que se encuentre en el directorio de Prometheus, puede configurarlo para extraer su propio punto de conexión HTTP con la ayuda del archivo *prometheus.yaml*, que se encuentra en el directorio de Prometheus. El archivo predeterminado ya contiene el puerto TCP 9090 en el localhost donde se ejecuta el servidor de Prometheus. Puede ejecutar Prometheus ejecutando el binario con el siguiente comando:

```
$ ./prometheus
level=info ts=2020-11-29T06:51:43.519Z caller=head.go:659 component=tsdb msg="On-disk
    memory mappable chunks replay completed" duration=7.944µs
level=info ts=2020-11-29T06:51:43.519Z caller=head.go:665 component=tsdb msg="Replaying WAL,
    this may take a while"
level=info ts=2020-11-29T06:51:43.520Z caller=head.go:717 component=tsdb msg="WAL segment
    loaded" segment=0 maxSegment=0
level=info ts=2020-11-29T06:51:43.520Z caller=head.go:722 component=tsdb msg="WAL replay
    completed" checkpoint_replay_duration=93.857µs wal_replay_duration=695.77µs
    total_replay_duration=812.641µs
level=info ts=2020-11-29T06:51:43.521Z caller=main.go:742 fs_type=19
level=info ts=2020-11-29T06:51:43.521Z caller=main.go:745 msg="TSDB started"
level=info ts=2020-11-29T06:51:43.521Z caller=main.go:871 msg="Loading configuration file"
    filename=prometheus.yml
level=info ts=2020-11-29T06:51:43.712Z caller=main.go:902 msg="Completed loading of
    configuration file" filename=prometheus.yml totalDuration=190.710981ms
    remote_storage=6.638µs web_handler=591ns query_engine=807ns scrape=189.731297ms
    scrape_sd=42.312µs notify=322.934µs notify_sd=20.022µs rules=3.538µs
level=info ts=2020-11-29T06:51:43.712Z caller=main.go:694 msg="Server is ready to receive
    web requests."
```

Para acceder a la interfaz de usuario de Prometheus en su navegador, vaya a <http://localhost:9090>, y revise las diversas métricas que el servidor de Prometheus está transmitiendo sobre sí mismo en <http://localhost:9090/metrics>.

Ahora puede usar el *Navegador de expresiones* para ejecutar básicamente consultas o escribir **expresiones PromQL**.

PromQL es el lenguaje de consulta funcional de Prometheus que le permite seleccionar y agregar datos de series temporales en tiempo real. Los datos se pueden visualizar en la interfaz de usuario de Prometheus en formato de tabla o gráfico.

Para utilizar el Navegador de expresiones, use cualquier métrica que se exponga desde [localhost:9090/metrics](http://localhost:9090/metrics), como `prometheus_http_requests_total` (que es un contador para todas las solicitudes HTTP al servidor de Prometheus). A continuación, haga clic en la pestaña Graph (Gráfico) e ingrese la métrica en la consola de expresiones, como se muestra en la [Figura 6-3](#).

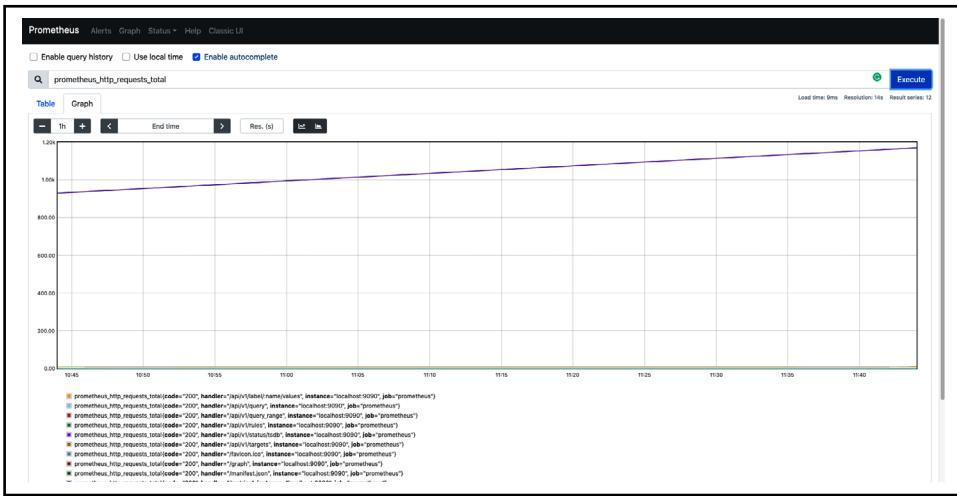


Figura 6-3. Resultado del navegador de expresiones para `prometheus_http_requests_total`

El lenguaje de consulta es muy extenso y admite una gran cantidad de funciones y operadores en las métricas. Por ejemplo, si desea contar el número de series temporales devueltas para la métrica `prometheus_target_interval_length_seconds`, puede emitir el siguiente comando en el Navegador de expresiones:

```
count(prometheus_target_interval_length_seconds)
```

Del mismo modo, si desea saber qué tan rápido aumenta una métrica por segundo, puede usar `rate` en el Navegador de expresiones:

```
rate(prometheus_tsdb_head_samples_appended_total[1m])
```

## node\_exporter

El `node_exporter` expone una variedad de métricas de nivel de host, como CPU, memoria, espacio en disco, I/O y ancho de banda de red, junto con una variedad de métricas de nivel de kernel. Puede descargar el `node_exporter` desde <https://prometheus.io/download>. Una vez que haya descargado y extraído el archivo comprimido, puede ejecutar el binario directamente sin realizar ningún cambio:

```
$ cd node_exporter-1.0.1.darwin-amd64
$ ./node_exporter
level=info ts=2020-12-01T13:20:29.510Z caller=node_exporter.go:177 msg="Starting
node_exporter" version="(version=1.0.1, branch=HEAD,
revision=3715be6ae899f2a9b9dbfd9c39f3e09a7bd4559f)"
level=info ts=2020-12-01T13:20:29.510Z caller=node_exporter.go:178 msg="Build context"
build_context="(go=go1.14.4, user=root@04c8e5c628328, date=20200616-12:52:07)"
level=info ts=2020-12-01T13:20:29.511Z caller=node_exporter.go:105 msg="Enabled collectors"
level=info ts=2020-12-01T13:20:29.511Z caller=node_exporter.go:112 collector=boottime
level=info ts=2020-12-01T13:20:29.511Z caller=node_exporter.go:112 collector=cpu
level=info ts=2020-12-01T13:20:29.511Z caller=node_exporter.go:112 collector=diskstats
level=info ts=2020-12-01T13:20:29.511Z caller=node_exporter.go:112 collector=filesystem
```

```
level=info ts=2020-12-01T13:20:29.511Z caller=node_exporter.go:112 collector=loadavg
level=info ts=2020-12-01T13:20:29.511Z caller=node_exporter.go:112 collector=meminfo
level=info ts=2020-12-01T13:20:29.511Z caller=node_exporter.go:112 collector=netdev
level=info ts=2020-12-01T13:20:29.511Z caller=node_exporter.go:112 collector=textfile
level=info ts=2020-12-01T13:20:29.511Z caller=node_exporter.go:112 collector=time
level=info ts=2020-12-01T13:20:29.511Z caller=node_exporter.go:112 collector=uname
level=info ts=2020-12-01T13:20:29.511Z caller=node_exporter.go:191 msg="Listening on"
    address=:9100
level=info ts=2020-12-01T13:20:29.511Z caller=tls_config.go:170 msg="TLS is disabled and it
cannot be enabled on the fly." http2=false
```

El `node_exporter` se ejecuta en <http://localhost:9100> de forma predeterminada. El archivo `scrape_configs` en `prometheus.yml` necesita actualizarse con un poco más de información para que pueda supervisar el `node_exporter` extrayendo datos de él:

```
scrape_configs:
  - job_name: node
    static_configs:
      - targets:
          - localhost:9100
```

Una vez que reinicie Prometheus para incorporar el nuevo cambio, Prometheus tendrá dos objetivos de punto de conexión desde los cuales extraer los datos de métricas. También puede comprobar el punto de conexión de <http://localhost:9090/targets>. Ahora, supongamos que necesita averiguar cuántos segundos dedica cada CPU a realizar diferentes tipos de trabajo (es decir, usuario, sistema, iowait, etc.). Para ello, puede ejecutar la siguiente consulta en el Navegador de expresiones, utilizando la función `irate`:

```
irate(node_cpu_seconds_total{job="node"}[5m])
```

`job="node"` es la etiqueta que se alineó y que filtra las métricas. Ahora veamos cómo la instrumentación de las aplicaciones puede trabajar con Prometheus.

## Instrumentación de aplicaciones

Para supervisar los servicios personalizados, tenemos que agregar instrumentación, lo que implica agregar una biblioteca de seguimiento al código de la aplicación para permitirnos implementar los tipos de métricas de Prometheus. En esta sección, veremos cómo lograrlo con la ayuda de las bibliotecas de cliente, que puede usar para instrumentar el código de su aplicación. La biblioteca de cliente principalmente le permite definir y exponer las métricas internas utilizando puntos de conexión HTTP en su aplicación. Cuando Prometheus extrae la instancia de la aplicación, la biblioteca del cliente le envía el estado actual de las métricas. Usaremos Python como la biblioteca de instrumentación preferida, aunque hay más implementaciones de bibliotecas de clientes disponibles en <https://prometheus.io/docs/instrumenting/clientlibs>.

Lo primero que hay que hacer es instalar la biblioteca de cliente en su host utilizando `pip` de la siguiente manera:

```
$ pip install prometheus_client
```

En función de los cuatro tipos de métricas que mencionamos antes, hay principalmente cuatro formas diferentes en las que puede instrumentar el código. Analizaremos estos métodos en las siguientes secciones.

## Contadores

Como su nombre indica, los contadores se utilizan principalmente para contar los valores que aumentan y se pueden restablecer a cero en el reinicio cuando se reinicia la aplicación. Por ejemplo, cuando se envía una solicitud GET al servidor HTTP de Python, el siguiente código aumenta el valor del contador en 1:

```
import http.server
from prometheus_client import Counter, start_http_server

http_requests = Counter('my_app_http_request','Description: Num of HTTP request')

class SampleServer(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        http_requests.inc()
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Simple Counter Example")

if __name__ == "__main__":
    start_http_server(5555)
    server = http.server.HTTPServer(('localhost', 5551), SampleServer)
    server.serve_forever()
```

Puede ejecutar este bloque de código como 'python counter.py' en su terminal y se iniciará un servidor HTTP simple. Estamos usando `start_http_server` como un punto de conexión desde donde se extraerán las métricas. Puede acceder a las métricas <http://localhost:5555>. Para ingerir estas métricas en Prometheus, es necesario configurar de nuevo el archivo `scrape_configs` en el archivo `prometheus.yml` de la siguiente manera:

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
  - job_name: node
    static_configs:
      - targets:
          - localhost:9100
  - job_name: my_application
    static_configs:
      - targets:
          - localhost:5555
```

Podrá acceder a la métrica a través del Navegador de expresiones mediante la expresión PromQL `rate(my_app_http_request_total[1m])`, una vez que reinicie Prometheus. El contador aumentará cada vez que se llegue la URL de la aplicación en [http://localhost: 5551](http://localhost:5551). El gráfico de la Figura 6-4 muestra una mayor tasa de solicitudes de HTTP entrantes.

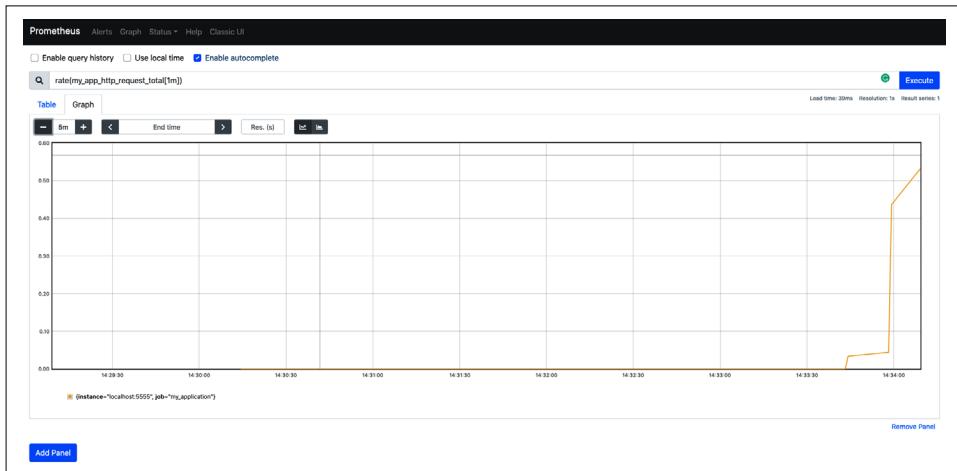


Figura 6-4. Contador de métricas para una aplicación instrumentada

Como puede ver, es bastante fácil instrumentar código con contadores. También puede usar contadores para contar las excepciones y los errores en el código, e incluso tener un valor personalizado para el contador en lugar de aumentarlo en uno.

## Medidores

Los medidores indican principalmente el estado actual de una entidad y pueden aumentar o disminuir. Los medidores se pueden usar en un par de escenarios diferentes. Por ejemplo, supongamos que necesita saber el número de subprocessos activos o el número de elementos en una cola, o desea conocer el número de elementos en una memoria caché.

Los medidores utilizan tres métodos principales: `inc`, `dec` y `set`. En el siguiente ejemplo, se muestra cómo puede usar los tres métodos de forma independiente para realizar seguimiento de una métrica que le interesa:

```
from prometheus_client import Gauge
sample_gauge_1 = Gauge('my_increment_example_requests', 'Description of increment gauge')
sample_gauge_2 = Gauge('my_decrement_example_requests', 'Description of decrement gauge')
sample_gauge_3 = Gauge('my_set_example_requests', 'Description of set gauge')

sample_gauge_1.inc()      # This will increment by 1
sample_gauge_2.dec(10)    # This will decrement by given value of 10
sample_gauge_3.set(48)    # This will set to the given value of 48
```

En el código anterior, el método `inc()` incrementará el valor de '`my_increment_example_requests`' por 1, disminuirá el valor de '`my_decrement_example_requests`' en 10 y establecerá el valor de '`my_set_example_requests`' en 48.

## Resumen

Un resumen suele hacer un seguimiento de la duración de la solicitud o el tamaño de respuesta. Normalmente, se usa para encontrar el tamaño y el número de eventos. Utiliza un método `observe` que acepta el tamaño del evento. Por ejemplo, en el siguiente código, estamos tratando de buscar el tiempo que tardó una solicitud en completarse:

```
import http.server
import time
from prometheus_client import Summary, start_http_server

LATENCY = Summary('latency_in_seconds','Time for a request')

class SampleServer(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        start_time = time.time()
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"My application with a Summary metric")
        LATENCY.observe(time.time() - start_time)

    if __name__ == "__main__":
        start_http_server(5555)
        server = http.server.HTTPServer(('localhost', 5551), SampleServer)
        server.serve_forever()
```

Para ejecutar el código anterior, puede seguir el mismo enfoque que tomamos en "Contadores" en la página 131. El punto de conexión de la métrica en <http://localhost:5555/metrics> tendrá dos series temporales: `latency_in_seconds_count` y `latency_in_seconds_sum`. La primera representa el número de llamadas de `observe` que se realizaron y la segunda es la suma de los valores pasados a `observe`. Si divide la tasa de estas dos métricas, tasa(`latency_in_seconds_count[1m]`)/tasa(`latency_in_seconds_sum[1m]`), obtendrá la latencia promedio del último minuto.

## Histogramas

Los histogramas le ayudan a realizar el seguimiento del tamaño y el número de eventos en cubos, a la vez que le permite agregar cálculos de cuantiles. Los histogramas se pueden usar para medir las duraciones de solicitudes para una llamada de solicitud HTTP específica.

La instrumentación de código para los histogramas también usa el método `observe`, y puede combinarlo con el tiempo para realizar seguimiento de la latencia, como en nuestro ejemplo anterior. El siguiente código agrega 10 segundos como una latencia de la solicitud:

```
from prometheus_client import Histogram
req_latency = Histogram('request_latency_seconds', 'Description of histogram')
req_latency.observe(10) # Observe 10 (seconds in this case)
```

Por lo general, la instrumentación se aplica en el nivel de servicio o en las bibliotecas de cliente que se usan en los servicios. Es importante tener en cuenta que hay límites en el

nivel de instrumentación que puede o debe agregar, a pesar de que Prometheus es eficiente para manejar varias métricas.<sup>1</sup>

Ahora que entiende cómo funciona la instrumentación del código y cómo se agregan los hosts a Prometheus y extraen, veamos cómo agregar hosts a Prometheus para que se extraigan en un entorno de producción.

## Encontrar hosts

Prometheus utiliza el archivo *static\_configs* en *scrape\_configs* en el archivo *prometheus.yml* para encontrar los destinos de host que se extraerán. Los entornos de nube son de naturaleza muy dinámica y el cambio ocurren con mucha frecuencia, en especial en entornos contenedorizados. En tales entornos, agregar hosts de forma manual al archivo *prometheus.yml* no escalará muy bien. Prometheus ofrece una amplia variedad de formas de manejar tales situaciones al aceptar muchos orígenes, como Kubernetes, Azure y plataformas de detección de servicios personalizados. La detección de servicios es una forma sencilla de descubrir todos los servicios que se ejecutan en un entorno de nube distribuida. Hablaremos en detalle sobre la detección de servicios en el Capítulo 7. Aquí veremos un par de maneras en que podemos descubrir los hosts en producción para que Prometheus los extraiga.

### Uso de Ansible

Si los hosts están ejecutando exportadores de nodo, una de las formas más sencillas de encontrarlos es actualizar estáticamente el archivo *prometheus.yml* con los hosts del inventario. Básicamente, cada vez que agregue una máquina, también debe tener una provisión para agregarla al archivo de configuración *prometheus.yml*. A continuación, se muestra un fragmento para actualizar *prometheus.yml* con Ansible:

```
scrape_configs:  
  - job_name: {{ hostname }}  
    static_configs:  
      - targets:  
        - {{ hostname }}:9100
```

También puede realizar un bucle sobre los hosts de cualquier grupo en un inventario de Ansible y agregarlos directamente.

### Uso de archivos

También puede proporcionar una lista de objetivos para leer directamente desde un archivo YAML o JSON sin ninguna sobrecarga de red. Este es otro método estático de detección de servicios en el que Prometheus puede leer los hosts directamente desde un archivo estático, por ejemplo, si tiene un archivo *inventory.json*:

---

<sup>1</sup> El equipo de Prometheus compiló los procedimientos recomendados sobre el uso de la instrumentación en <https://prometheus.io/docs/practices/instrumentation/#counter-vs-gauge-summary-vs-histogram>.

```
[
  [
    {
      "targets": [
        "192.168.101.11:9100"
      ],
      "labels": {
        "job": "node",
        "Team": "dev"
      }
    },
    [
      {
        "targets": [
          "192.168.101.155:9200"
        ],
        "labels": {
          "job": "node",
          "Team": "sre"
        }
      }
    ]
]
```

Ahora puede actualizar *prometheus.yml* usando `file_sd_configs` en `scrape_configs` de la siguiente manera:

```
scrape_configs:
  - job_name: file
    file_sd_configs:
      - files:
          - 'inventory.json'
```

Puede encontrar los objetivos que se están extrayendo en <http://localhost:9090/service-discovery>.

## Uso de `azure_sd_config`

Prometheus permite el soporte técnico inmediato para extraer las máquinas virtuales de Azure. Las configuraciones de detección de servicios (SD) de Azure le permiten recuperar objetivos de extracción de las máquinas virtuales de Azure. Puede usar la detección de servicios de Azure estableciendo las credenciales de Azure en el archivo *prometheus.yml* de la siguiente manera:

```
- job_name: 'azure-nodes'
  azure_sd_configs:
    - subscription_id: '$SUBSCRIPTION_ID'
      tenant_id: '$TENANT_ID'
      client_id: '$CLIENT_ID'
      client_secret: '$CLIENT_SECRET'
      port: 9100
    relabel_configs:
      - source_labels: [__meta_azure_machine_tag_ccloudnative]
        regex: true.*
        action: keep
      - source_labels: [__meta_azure_machine_name]
        target_label: web_instance
      - source_labels: [__meta_azure_machine_tag_public_ip]
        regex: (.+)
        replacement: ${1}:9100
        target_label: __address__
```

En esta configuración, también estamos usando el *reetiquetado*, que vuelve a escribir dinámicamente el conjunto de etiquetas de un objetivo antes de que se extraiga. El reetiquetado ayuda a manipular las métricas para mantener su almacenamiento limpio y no contaminarlo con datos innecesarios. Se aplican varios reetiquetados al conjunto de etiquetas de cada objetivo en el orden en que se produzcan en el archivo de configuración.

Echemos un vistazo más de cerca a las partes clave de *relabel\_configs* anterior:

#### `_meta_azure_machine_tag_cloudnative`

Prometheus examina todas las instancias de la máquina virtual de Azure y, si una instancia tiene una etiqueta llamada `cloudnative`, mantiene el servicio. Básicamente, esto le dice a Prometheus: "Si alguna máquina virtual se encuentra en la cuenta de Azure cuya etiqueta es `cloudnative`: `true`, extráela; de lo contrario, ignórala".

#### `_meta_azure_machine_name`

Aquí, Prometheus busca `machine_name` y luego la coloca en una etiqueta `web_instance`.

#### `_meta_azure_machine_tag_public_ip`

Aquí, Prometheus anexa `:9100` a la IP pública. También reemplaza el contenido de `_address_` label.

Una vez que tenga algunas máquinas virtuales de Azure en ejecución, puede ver el objetivo detectado en <http://localhost:9090/service-discovery> junto con los metadatos extraídos de la máquina virtual.

Ahora que hemos visto cómo funciona la detección de servicios tradicional en Prometheus, veamos cómo trabaja Prometheus con Kubernetes y ayuda a cerrar la brecha de supervisión en las aplicaciones nativas de la nube.

## Prometheus en Kubernetes

El soporte de detección de servicios de Kubernetes está integrado directamente en Prometheus desde el inicio, lo que permite recuperar los objetivos de extracción de la API de REST de Kubernetes. Para usar esta capacidad, Prometheus ya debe estar ejecutándose dentro del clúster de Kubernetes. Puede usar los gráficos de Helm para configurar e implementar Prometheus fácilmente en un clúster de AKS.

Se pueden configurar cinco tipos de roles para detectar objetivos en los clústeres de Kubernetes: nodo, servicio, pod, puntos de conexión e ingreso. Revisemos en detalle cada uno de estos roles de la detección de servicios.

### El rol de nodo

El rol de *nodo* detecta todos los nodos en el clúster de Kubernetes. Dado que kubelet se ejecuta en cada nodo de un clúster de Kubernetes, el rol de nodo detecta los nodos objetivo en el

clúster con la dirección predeterminada para el puerto HTTP de kubelet. Puede extraer los nodos de kubelet con el siguiente comando `scrape_configs` de Prometheus:

```
scrape_configs:
  - job_name: 'kubelet'
    kubernetes_sd_configs:
      - role: node
    scheme: https
    tls_config:
      ca_file: ca.crt
```

En la configuración anterior de `prometheus.yml`, la etiqueta se proporciona como `kubelet` y el rol se proporciona como `nodo`. Usamos el esquema HTTPS, ya que las métricas de kubelet solo están disponibles a través de HTTPS, que también requiere un certificado de Seguridad de la capa de transporte (TLS).

### El rol de servicio

El rol de *servicio* lo ayuda principalmente a supervisar los sistemas opacos para comprobar si todos los servicios responden. El rol de servicio encuentra un objetivo para cada puerto de servicio para todos los servicios. Puede extraer un objeto de servicio de la siguiente manera:

```
scrape_configs:
  - job_name: prometheus-pushgateway
    kubernetes_sd_configs:
      - role: service
```

### El rol de pod

El rol de *pod* es responsable de detectar todos los pods que se ejecutan en el clúster y exponer los contenedores que se ejecutan en ellos como objetivos. También devuelve todos los metadatos relacionados con los pods. Puede extraer los pods de la siguiente manera:

```
scrape_configs:
  - job_name: 'pods'
    kubernetes_sd_configs:
      - role: pod
```

### El rol de puntos de conexión

El rol de *puntos de conexión* encuentra los objetivos de los puntos de conexión de servicio en un clúster de Kubernetes. Para cada dirección de punto de conexión, se detecta un objetivo por puerto. Puede extraer los puntos de conexión de la siguiente manera:

```
scrape_configs:
  - job_name: 'k8apiserver'
    kubernetes_sd_configs:
      - role: endpoints
    scheme: https
    tls_config:
      ca_file: cert.crt
```

## El rol de ingreso

Por último, el rol de *ingreso* detecta un objetivo para cada ruta de cada ingreso. Este rol también es una forma de realizar una supervisión opaca del ingreso. Puede extraer un ingreso de la siguiente manera:

```
scrape_configs:  
  - job_name: 'gateway'  
    kubernetes_sd_configs:  
      - role: ingress  
    scheme: https  
    tls_config:  
      ca_file: cert.crt
```

Ahora que ya examinamos las métricas en detalle, veamos cómo se implementa el registro en las aplicaciones nativas de la nube.

## Crear registros en el mundo nativo de la nube

Los registros son una pieza fundamental de información en cualquier entorno. Los ingenieros los usan principalmente al resolver un problema u obtener información sobre un sistema. Sin embargo, crear registros en general es complejo. La complejidad surge debido a las diversas formas de consumir registros, las diferencias en los formatos de los datos y la necesidad de garantizar que la aplicación subyacente que se encarga de almacenar los registros también se escala de forma correcta a medida que aumente el número de registros. En entornos distribuidos nativos de la nube, los registros se vuelven aún más importantes. Estos entornos no solo tienen muchas partes móviles, sino que también son más complejos, ya que ahora los sistemas están distribuidos y son efímeros, como los pods de Kubernetes. Esto significa que los registros deben enviarse y almacenarse de forma centralizada para poder verlos más adelante.

Para resolver los problemas de los registros a escala en un entorno nativo de la nube, necesitamos una solución que se pueda conectar a un entorno fácilmente e integrarse sin problemas. Aquí es donde entra en escena *Fluentd*.

## Crear registros con Fluentd

Fluentd es un proyecto de open source de la CNCF que puede recopilar, analizar, redistribuir y examinar registros. En esencia, Fluentd actúa como una capa de registro unificada mediante la estructuración de datos como JSON (consulte la [Figura 6-5](#)). Esto, a su vez, ayuda a recopilar, filtrar, almacenar en búfer y generar los registros en varios orígenes y destinos.

Fluentd está escrito en CRuby, que es una implementación C del lenguaje de programación Ruby que viene con un espacio de memoria pequeño de 30 a 40 MB y procesa aproximadamente 13 000 eventos por segundo por núcleo. Veamos algunos de los conceptos básicos de Fluentd y cómo se puede usar para habilitar los registros en entornos nativos de la nube.

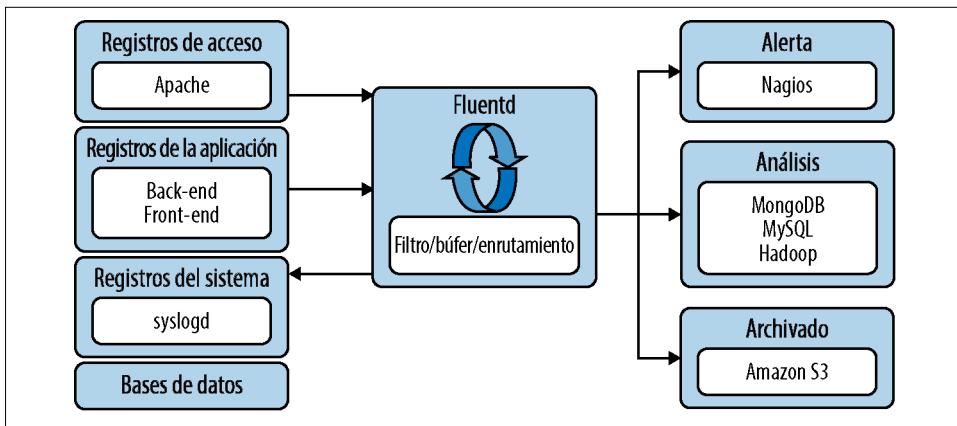


Figura 6-5. Fluentd actuando como una capa de registro

Para comenzar, instale el agente, en función de la distribución del sistema operativo, desde el [sitio web oficial de Fluentd](#).



El paquete oficial de distribución estable de Fluentd se denomina *td-agent*. Garantiza la estabilidad y viene preconfigurado con la configuración recomendada del autor original, Treasure Data Inc.

Una vez que haya descargado e instalado *td-agent* para su distribución, asegúrese de que esté en funcionamiento. Por ejemplo, en macOS puede hacerlo a través de `launchctl`:

```
$ ~ sudo launchctl load /Library/LaunchDaemons/td-agent.plist
Password:
$
```

Puede confirmar que el agente se esté ejecutando a través de los registros disponibles en la siguiente ubicación.

```
$ ~ tail -f /var/log/td-agent/td-agent.log
2020-12-17 22:57:46 +0530 [info]: adding match pattern="td.*.*" type="tdlog"
2020-12-17 22:57:46 +0530 [warn]: #0 [output_td] secondary type should be same with primary
one primary="Fluent::Plugin::TreasureDataLogOutput" secondary="Fluent::Plugin::FileOutput"
2020-12-17 22:57:46 +0530 [info]: adding match pattern="debug.**" type="stdout"
2020-12-17 22:57:46 +0530 [info]: adding source type="forward"
2020-12-17 22:57:46 +0530 [info]: adding source type="http"
2020-12-17 22:57:46 +0530 [info]: adding source type="debug_agent"
2020-12-17 22:57:46 +0530 [info]: #0 starting fluentd worker pid=96320 ppid=95841 worker=0
2020-12-17 22:57:47 +0530 [info]: #0 [input_debug_agent] listening dRuby
uri="druby://127.0.0.1:24230" object="Fluent::Engine"
2020-12-17 22:57:47 +0530 [info]: #0 [input_forward] listening port port=24224
bind="0.0.0.0"
2020-12-17 22:57:47 +0530 [info]: #0 fluentd worker is now running worker=0
```

Fluentd se entiende mejor con la ayuda de su archivo de configuración. Puede buscar el archivo de configuración predeterminado para Fluentd en `/etc/td-agent/td-agent.conf`. El archivo de configuración contiene una serie de directivas, como se explica en las siguientes subsecciones.

### Directiva source (donde se ingieren todos los datos)

La directiva `source` determina el origen de entrada de los datos que debe habilitarse (es decir, de dónde deben ingerirse los datos). La directiva `source` envía un evento al sistema de enrutamiento de Fluentd. De forma predeterminada, Fluentd ofrece dos complementos de entrada estándares. El complemento `http` ofrece un punto de conexión HTTP para escuchar los mensajes HTTP entrantes. El complemento `forward` proporciona un punto de conexión TCP para aceptar paquetes TCP:

```
<source>
@type forward
port 24224
</source>

<source>
@type http
port 9999
bind 0.0.0.0
</source>
```

La definición anterior especifica dos orígenes: en el primer origen estamos recibiendo eventos del puerto TCP 24224 y en el segundo tenemos un servidor HTTP escuchando en el puerto TCP 9999.

El evento que envió `source` al motor de enrutamiento de Fluentd contiene tres entidades: etiqueta, hora y registro. Por ejemplo, en el siguiente resultado del registro:

```
2020-12-16 15:38:27 +0900 test.root: {"action": "logout", "user": 3}
```

2020-12-16 15:38:27 +0900 es la hora, que se especifica mediante complementos de entrada y tiene que estar en formato de tiempo Unix. `test.root` es la etiqueta (según Fluentd, se recomienda que la etiqueta esté en minúsculas y sea una combinación de dígitos y guiones bajos). Y `{"action": "logout", "user": 3}` es un objeto JSON.

Los complementos de entrada son responsables de recuperar los registros de eventos de orígenes externos. Por lo general, un complemento de entrada crea un subproceso, un socket y un socket de escucha. Los complementos de entrada no solo se limitan a `http` y `forward`. A continuación, puede ver algunos complementos adicionales:

#### in\_syslog

Para recuperar eventos de registros a través del protocolo Syslog en UDP o TCP. Fluentd crea un socket en el puerto 5140. Entonces, lo único que necesita hacer es configurar su demonio de syslog para enviar eventos de registro en este socket:

```
<source>
@type syslog
port 5140
bind 0.0.0.0
```

```
    tag system  
</source>
```

## in\_tail

Permite que Fluentd lea los eventos del final de los archivos de registro, al igual que el comando `tail -f` de Unix:

```
<source>  
  @type tail  
  path /var/log/httpd-access.log  
  pos_file /var/log/td-agent/httpd-access.log.pos  
  tag apache.access  
  <parse>  
    @type apache2  
  </parse>  
</source>
```

Fluentd ofrece otros complementos de entrada, como `in_unix` para recuperar registros del socket de dominio Unix y `in_exec` para ejecutar programas externos y extraer registros de eventos. Incluso puede escribir sus propios complementos de entrada personalizados.

## Directiva match (qué hacer con los datos de entrada)

La directiva `match` determina el destino de salida al buscar los eventos con etiquetas coincidentes. Los complementos de salida estándares para Fluentd incluyen `file` y `forward`. La directiva `match` debe incluir un patrón de coincidencia y un parámetro `@type` que defina el complemento de salida que se usará. Una vez que un evento con una etiqueta coincide con el patrón, se envía al destino de salida. Este es un ejemplo de uso de la directiva `match`:

```
<match mywebapp.access>  
  @type file  
  path /var/log/mywebapp/access  
</match>
```

En la configuración anterior, estamos haciendo coincidir los eventos etiquetados con `mywebapp.access` y almacenándolos en `file` en `/var/log/mywebapp/access`.

Además de `file` y `forward`, puede tener otros tipos de complementos de salida que básicamente determinan dónde desea escribir los datos. Por ejemplo, `out_elasticsearch` es un complemento de salida que escribe los registros en un punto de conexión de clúster de Elasticsearch. Primero debe instalar el complemento con lo siguiente:

```
fluent-gem install fluent-plugin-elasticsearch
```

A continuación, puede usar la siguiente configuración para enviar los datos de salida a un punto de conexión de Elasticsearch:

```
<match my.logs>  
  @type elasticsearch  
  host 192.168.23.13  
  port 9200  
  logstash_format true  
</match>
```

Del mismo modo, puede usar `out_copy` para copiar eventos a varias salidas, `out_kafka2` para enviar registros a un clúster de Kafka, `out_mongo` para escribir registros en una base de datos de MongoDB y `stdout` para simplemente escribir eventos en la salida estándar.

## Directiva filter (la canalización de procesamiento de eventos)

La directiva `filter` actúa como una regla para aprobar o rechazar un evento en función de una condición. `filter` tiene la misma sintaxis que `match`, pero `filter` se puede encadenar a una canalización de procesamiento. Puede agregar una directiva `filter` de la siguiente manera:

```
<source>
  @type http
  port 9999
  bind 0.0.0.0
</source>

<filter test.session>
  @type grep
  <exclude>
    key action
    pattern ^logout_session$
  </exclude>
</filter>

<match test.session>
  @type stdout
</match>
```

En la configuración anterior, una vez que se obtienen los datos, pasan a través de la sección `filter` y luego la sección `match`. La directiva `filter` aceptará o rechazará el evento en función de su tipo y regla. Aquí descartamos la acción `logout_session` y usamos el tipo `grep` dentro de `filter` para excluir cualquier mensaje en el que la clave `action` tenga la cadena `logout`.

A continuación, puede ver algunos complementos `filter` adicionales:

### filter\_record\_transformer

Se utiliza para cambiar un flujo de eventos entrante. Puede usar el siguiente ejemplo de configuración para `record_transformer`:

```
<filter myweb.access>
  @type record_transformer
  <record>
    host_param "#{Socket.gethostname}"
  </record>
</filter>
```

En la configuración anterior, `record_transformer` agregará el campo `host_param` al evento de registro.

### **filter\_geoip**

Agrega información de ubicación geográfica a los registros mediante la base de datos de MaxMind GeoIP.

### **filter\_stdout**

Imprime eventos en la salida estándar y se utiliza principalmente para fines de depuración.

## **Directiva system (establece la configuración de todo el sistema)**

Puede establecer diversas configuraciones para todo el sistema mediante la directiva **system**. Algunas de las opciones de configuración disponibles en la directiva **system** son las siguientes:

- **log\_level**
- **suppress\_repeated\_stacktrace**
- **emit\_error\_log\_interval**
- **suppress\_config\_dump**
- **without\_source**
- **process\_name**

Puede utilizar la configuración de **system** de la siguiente manera:

```
<system>
  process_name my_app
  log_level error
  without_source
</system>
```

Aquí primero establecemos los nombres de los procesos de supervisor y de trabajo para **my\_app** y luego establecemos el nivel de registro predeterminado en **error**. También le estamos ordenando a Fluentd que empiece sin ningún complemento de entrada.

## **Directiva label (para la agrupación y el enrutamiento de la salida)**

Las etiquetas resuelven principalmente el problema de mantener y administrar el archivo de configuración al permitir secciones de enrutamiento que no siguen el orden de análisis de arriba hacia abajo en el archivo de configuración. También puede considerar las etiquetas como instrucciones GoTo. Este es un ejemplo de configuración simple para el uso de etiquetas:

```
<source>
  @type http
  bind 0.0.0.0
  port 9999
  @label @MYCLUE
</source>

<filter test.session>
  @type grep
  <exclude>
```

```

    key action
    pattern ^login$
  </exclude>
</filter>

<label @MYCLUE>
  <filter test.session>
    @type grep
    <exclude>
      key action
      pattern ^logout$
    </exclude>
  </filter>

  <match test.session>
    @type stdout
  </match>
</label>

```

En la configuración anterior, el parámetro `@label` en `source` es la redirección hacia la sección de la etiqueta `@MYCLUE`. El control omitirá la definición de `filter` para `login` y se moverá directamente a la etiqueta `@MYCLUE`.

### **Directiva @include (para importar archivos de configuración)**

Con la directiva `@include`, puede importar archivos de configuración desde un directorio independiente. La directiva `@include` admite la ruta de archivo normal, el patrón glob y las convenciones de URL HTTP de la siguiente manera:

```

# absolute path
@include /path/to/config.conf

# if using a relative path, the directive will use
# the dirname of this config file to expand the path
@include extra.conf

# glob match pattern
@include config.d/*.conf

# http
@include http://example.com/fluent.conf

```

Antes de continuar, veamos rápidamente cómo ejecutar el agente para ver sus registros. Para ejecutar una configuración sencilla como la siguiente:

```

<source>
  @type http
  port 9999
  bind 0.0.0.0
</source>

<filter myapp.test>
  @type grep
  <exclude>
    key action
    pattern ^logout$
  </exclude>

```

```

</filter>

<match myapp.test>
  @type stdout
</match>

$ sudo launchctl load /Library/LaunchDaemons/td-agent.plist
#check logs
$ tail -f /var/log/td-agent/td-agent.log
2020-12-21 12:21:57 +0530 [info]: parsing config file is succeeded \
  path="/etc/td-agent/td-agent.conf"
2020-12-21 12:21:58 +0530 [info]: using configuration file: <ROOT>
<source>
  @type http
  port 9999
  bind "0.0.0.0"
</source>
<filter myapp.test>
  @type grep
  <exclude>
    key "action"
    pattern ^logouts
  </exclude>
</filter>
<match myapp.test>
  @type stdout
</match>
</ROOT>
2020-12-21 12:21:58 +0530 [info]: starting fluentd-1.0.2 pid=43215 ruby="2.4.2"
2020-12-21 12:21:58 +0530 [info]: spawn command to main: \
  cmdline=["/opt/td-agent/embedded/bin/ruby", "-Eascii-8bit:ascii-8bit", \
  "/opt/td-agent/usr/sbin/td-agent", "--log", "/var/log/td-agent/td-agent.log", \
  "--use-v1-config", "--under-supervisor"]
2020-12-21 12:21:58 +0530 [info]: gem 'fluent-plugin-elasticsearch' version '2.4.0'
2020-12-21 12:21:58 +0530 [info]: gem 'fluent-plugin-kafka' version '0.6.5'
2020-12-21 12:21:58 +0530 [info]: gem 'fluent-plugin-rewrite-tag-filter' version '2.0.1'
2020-12-21 12:21:58 +0530 [info]: gem 'fluent-plugin-s3' version '1.1.0'
2020-12-21 12:21:58 +0530 [info]: gem 'fluent-plugin-td' version '1.0.0'
2020-12-21 12:21:58 +0530 [info]: gem 'fluent-plugin-td-monitoring' version '0.2.3'
2020-12-21 12:21:58 +0530 [info]: gem 'fluent-plugin-webhdfs' version '1.2.2'
2020-12-21 12:21:58 +0530 [info]: gem 'fluentd' version '1.0.2'
2020-12-21 12:21:58 +0530 [info]: adding filter pattern="myapp.test" type="grep"
2020-12-21 12:21:58 +0530 [info]: adding match pattern="myapp.test" type="stdout"
2020-12-21 12:21:58 +0530 [info]: adding source type="http"
2020-12-21 12:21:58 +0530 [info]: #0 starting fluentd worker pid=43221 ppid=43215 worker=0
2020-12-21 12:21:58 +0530 [info]: #0 fluentd worker is now running worker=0

```

Ahora puede ejecutar los siguientes comandos `curl` para simular diferentes acciones:

```

$ curl -i -X POST -d 'json=[{"action":"login","user":2}' http://localhost:9999/myapp.test

HTTP/1.1 200 OK
Content-Type: text/plain
Connection: Keep-Alive
Content-Length: 0

```

Dado que actualmente estamos aprobando la acción como `login`, podemos comprobar los registros en la ubicación para verificar que se hayan impreso:

```
2020-12-21 12:31:31.529967000 +0530 myapp.test: {"action": "login", "user": 2}
```

Si intenta publicar una acción como `logout` en el siguiente comando `curl`, se descartará en función de la regla de filtro que hemos especificado y no podrá ver los registros imprimiéndose:

```
$ curl -i -X POST -d 'json={"action": "logout", "user": 2}' http://localhost:9999/myapp.test
HTTP/1.1 200 OK
Content-Type: text/plain
Connection: Keep-Alive
Content-Length: 0
```

Ahora que entiende cómo funciona Fluentd, inspeccionemos cómo integrarlo en entornos nativos de la nube e implementarlo en orquestadores de contenedores como Kubernetes.

## Fluentd en Kubernetes

De forma predeterminada, en Kubernetes, los registros de los contenedores se escriben en la salida estándar y en el flujo de errores estándar. Este es el registro básico de I/O que Kubernetes ofrece desde el primer momento. El motor de contenedores redirige los flujos a un controlador de registros configurado en Kubernetes para escribir los registros en un archivo en formato JSON. Como usuario, si desea ver los registros, puede usar el comando `kubectl logs`. Sin embargo, el problema con este enfoque es muy evidente: Si un contenedor falla o un pod se desaloja, o si el propio nodo muere, ya no podrá acceder a los registros de la aplicación. Por lo tanto, los registros deben tener un almacenamiento independiente de los contenedores, los pods y los nodos. Este concepto se conoce formalmente como *registro en el nivel del clúster* y requiere un back-end independiente para almacenar, analizar y consultar los registros. Aunque Kubernetes no ofrece de forma nativa una solución para el registro en el nivel del clúster, podemos combinar Fluentd con una solución de almacenamiento, como Elasticsearch, para crear una solución escalable.

Antes de pasar a la forma preferida de implementar Fluentd en Kubernetes, veamos los diferentes enfoques mediante los que se puede implementar el registro en el nivel del clúster, junto con sus ventajas y desventajas. Básicamente, hay dos formas de crear una solución de registro en el nivel del clúster:

- Usar un contenedor sidecar en cada pod de la aplicación.
- Usar un agente de registro de nivel de nodo que se ejecute en cada nodo.

Revisemos en detalle cada una de estas soluciones.

## Enfoque de contenedor sidecar

Por lo general, en Kubernetes ejecuta las aplicaciones en un pod, como se describe en el [Capítulo 5](#). En el enfoque de sidecar, puede crear uno o más contenedores sidecar dentro del pod de la aplicación. El enfoque del contenedor sidecar se puede utilizar de dos maneras diferentes:

- Puede usar el contenedor sidecar para transmitir los registros de la aplicación al stdout del contenedor.
- Puede ejecutar un agente de registro dentro de un contenedor sidecar, que básicamente toma los registros de los contenedores de las aplicaciones y los envía a un back-end de registro.

El enfoque de contenedores sidecar presenta algunas ventajas:

- Puede enviar registros de aplicaciones que se producen en diferentes formatos de registro (estructurados o no estructurados) al tener dos contenedores sidecar para transmitir un archivo de registro determinado a diferentes flujos de registros.
- Los contenedores sidecar pueden leer registros de archivos, sockets o journald, y cada contenedor sidecar imprimirá el registro en los flujos de stdout y stderr.
- Los contenedores sidecar también se pueden usar para rotar los archivos de registro que la propia aplicación no puede rotar.

También hay algunas desventajas de ejecutar un contenedor sidecar:

- La I/O del disco puede aumentar significativamente en un escenario en el que esté escribiendo los registros en un archivo y luego transmitiendo a stdout.
- Dado que hay más de un pod que ejecuta la aplicación, también tendría que implementar varios contenedores sidecar en cada pod.

## Enfoque de agente de registro de nivel de nodo

Otra forma de hacer frente a los registros en Kubernetes es implementar un agente de registro en cada nodo del clúster de Kubernetes. El agente se puede ejecutar como un contenedor en un demonio que puede tener acceso a todos los registros que producen los pods de aplicaciones. De hecho, esta también es la forma preferida de ejecutar agentes de Fluentd.

Se usa un DaemonSet de Kubernetes para implementar una estrategia de este tipo donde cada nodo puede ejecutar una copia de un pod de agente de registro de Fluentd. Fluentd proporciona oficialmente un [DaemonSet](#) con las reglas correctas para implementar en los clústeres de Kubernetes. Puede tomar una copia del DaemonSet mediante una clonación desde el repositorio oficial de Fluentd:

```
$ git clone https://github.com/fluent/fluentd-kubernetes-daemonset
```

Antes de comenzar a usar el DaemonSet que mencionamos, necesita un clúster de Kubernetes. Si ya está ejecutando un clúster de Azure Kubernetes (como Azure Kubernetes

Service [AKS]), puede comenzar implementando un back-end de registro para almacenar los archivos de registros. Elasticsearch es uno de los back-ends de registro que se puede utilizar para almacenar los registros generados. Puede implementar Elasticsearch con Helm en Kubernetes de la siguiente manera:

```
$ helm repo add elastic https://helm.elastic.co
$ helm install elasticsearch elastic/elasticsearch
```

Puede usar el reenvío de puertos para comprobar que el clúster de Elasticsearch se haya implementado correctamente:

```
$ kubectl port-forward svc/elasticsearch-master 9200
Forwarding from 127.0.0.1:9200 -> 9200
Forwarding from [::1]:9200 -> 9200
Handling connection for 9200
Handling connection for 9200
```

Elasticsearch debería estar disponible en <http://localhost:9200>. Para implementar el DaemonSet de Fluentd, primero debe modificar las siguientes variables del entorno en el archivo *fluentd-daemonset-elasticsearch-rbac.yaml*:

- FLUENT\_ELASTICSEARCH\_HOST (el punto de conexión de Elasticsearch)
- FLUENT\_ELASTICSEARCH\_PORT (el puerto de Elasticsearch, habitualmente 9200)

Una vez que haya actualizado el archivo YAML con los detalles de Elasticsearch, puede implementar el DaemonSet de Fluentd en AKS migrando al repositorio de GitHub y aplicando el DaemonSet de la siguiente manera:

```
$ kubectl apply -f fluentd-daemonset-elasticsearch-rbac.yaml
```

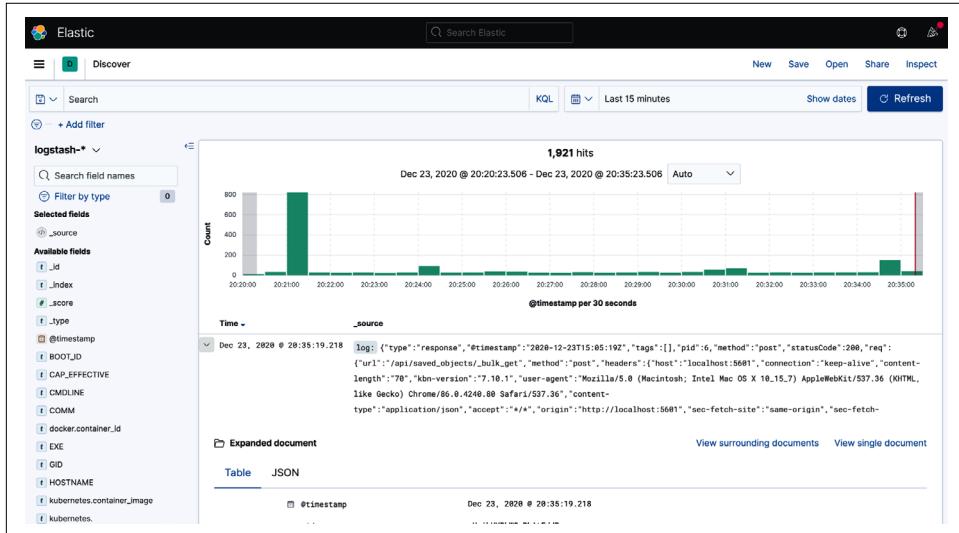
Esto implementará el DaemonSet de Fluentd en el clúster de Kubernetes, y ahora todos los nodos tendrán al menos un pod de Fluentd en ejecución que reenviará sus registros a Elasticsearch:

```
$ ~ kubectl get pods --namespace=kube-system -o wide
NAME                               READY   STATUS    RESTARTS   AGE     IP
  NODE                           NOMINATED NODE   READINESS GATES
fluentd-6pttr
  aks-agentpool-25245360-vmss000000  1/1     Running   2          3h37m   10.240.0.27
fluentd-9fj59
  aks-agentpool-25245360-vmss000001  1/1     Running   1          3h37m   10.240.0.147
fluentd-dh7pn
  aks-agentpool-25245360-vmss000003  1/1     Running   1          139m    10.240.1.84
fluentd-g5cnv
  aks-agentpool-25245360-vmss000002  1/1     Running   2          3h37m   10.240.1.25
fluentd-mzw2j
  aks-agentpool-25245360-vmss000004  1/1     Running   2          139m    10.240.2.0
fluentd-z52r2
  aks-agentpool-25245360-vmss000005  1/1     Running   2          139m    10.240.2.143
kube-proxy-56bv2
  aks-agentpool-25245360-vmss000005  1/1     Running   0          139m    10.240.2.47
```

También necesita ver estos registros, y puede hacerlo mediante Kibana.<sup>2</sup> Puede implementar Kibana de forma similar con la ayuda de Helm y el reenvío de puertos localmente de la siguiente manera:

```
$ helm repo add elastic https://helm.elastic.co
$ helm install kibana elastic/kibana
$ kubectl port-forward deployment/kibana-kibana 5601
Forwarding from 127.0.0.1:5601 -> 5601
Forwarding from [::1]:5601 -> 5601
```

Kibana (**Figura 6-6**) ahora debería estar listo en su <http://localhost:5601>.



*Figura 6-6. Presentación de registros con Kibana*

Otra herramienta nueva para la agregación de registros es [Loki de Grafana Labs](#). Loki registra de forma eficaz etiquetas de registros específicas en lugar de un flujo de registros completo y puede integrarse fácilmente con Fluentd. No profundizaremos en este tema, ya que va más allá del alcance de este libro.

Ahora que entiende cómo funcionan los registros en los entornos nativos de la nube, vamos a cerrar el capítulo con un análisis sobre el seguimiento distribuido.

2 Kibana es una interfaz de usuario de open source que generalmente se usa para visualizar datos (registros). Encontrará más información disponible en <https://elastic.co/kibana>.

# Seguimiento distribuido en el mundo nativo de la nube

El seguimiento distribuido es uno de los pilares críticos para obtener y desarrollar la observabilidad en las aplicaciones nativas de la nube moderna. Le permite realizar un seguimiento del flujo de solicitudes en un sistema distribuido a medida que pasan por varios servicios y emiten información de metadatos, como el tiempo dedicado a cada servicio y los detalles de llamadas subyacentes a otros puntos de conexión del servicio. Estos metadatos se pueden reagrupar más adelante para proporcionar una imagen más completa del comportamiento de la aplicación en el tiempo de ejecución, lo que puede ayudarlo a resolver problemas, como identificar problemas de latencia de la aplicación y localizar cuellos de botella.

El seguimiento distribuido da respuesta a preguntas importantes, como:

- ¿Cuáles son los cuellos de botella en un servicio?
- ¿Cuánto tiempo se dedica a cada etapa/servicio en un sistema distribuido?
- ¿Cuáles son los servicios ascendentes y descendentes y qué puntos de conexión alcanza una solicitud?

Para comprender mejor los fundamentos del seguimiento distribuido, observe la aplicación web simple que se muestra en la [Figura 6-7](#).

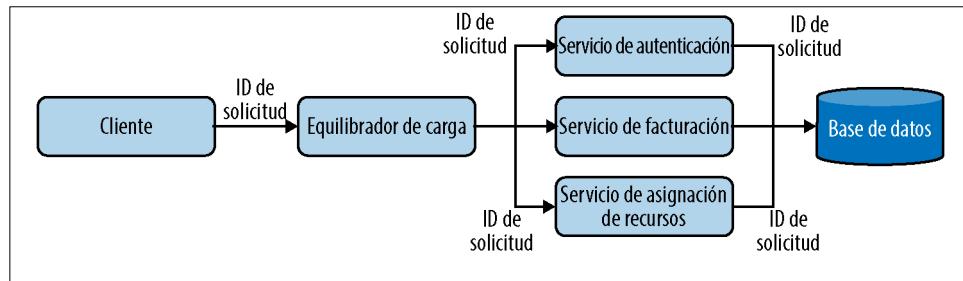
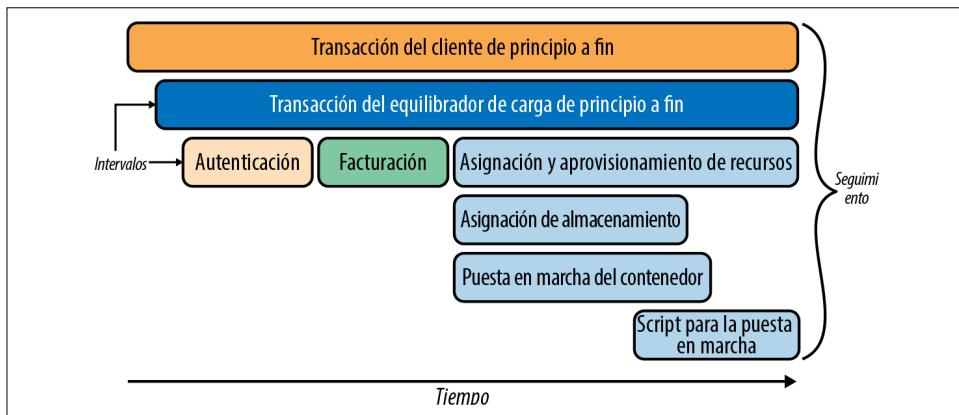


Figura 6-7. Aplicación web simple que muestra la propagación de solicitudes en varias capas de servicios

Considere la [Figura 6-7](#) y suponga que tiene un servicio de ejemplo que aprovisiona algunos recursos en una nube en base al plan del usuario. El flujo de ejecución normal comienza con el cliente que realiza una solicitud al equilibrador de carga, que se comunica con el servicio de autenticación y el servicio de facturación, y asigna recursos al cliente al realizar una llamada final al servicio de asignación de recursos. En la [Figura 6-8](#), puede ver que los identificadores de solicitud únicos se propagan en el flujo, lo que más adelante ayuda a unir los seguimientos.

Este flujo de transacciones se puede visualizar con la ayuda de un diagrama de Gantt (consulte la [Figura 6-8](#)).



*Figura 6-8. Intervalos y seguimientos en una aplicación web sencilla*

## Seguimiento: Conceptos clave

El diagrama de Gantt de la [Figura 6-8](#) resume los siguientes importantes conceptos clave de un sistema de seguimiento distribuido.

### Intervalos

Un intervalo representa una sola llamada operativa y es el bloque de creación principal de un seguimiento distribuido, que representa una unidad de trabajo individual que se realiza en un sistema distribuido. Los intervalos contienen *referencias* a otros intervalos, lo que permite que varios intervalos se ensamblen en un seguimiento completo. Por ejemplo, en la [Figura 6-8](#), el intervalo "Transacción del equilibrador de carga de principio a fin" incluye otros intervalos, como el equilibrador de carga que interactúa con el servicio de autenticación o el equilibrador de carga que interactúa con el servicio de facturación.

Los intervalos se crean en forma de una relación principal-secundario en la que cada intervalo tiene un elemento secundario, excepto para la raíz y el servicio de inicio ([Figura 6-9](#)).

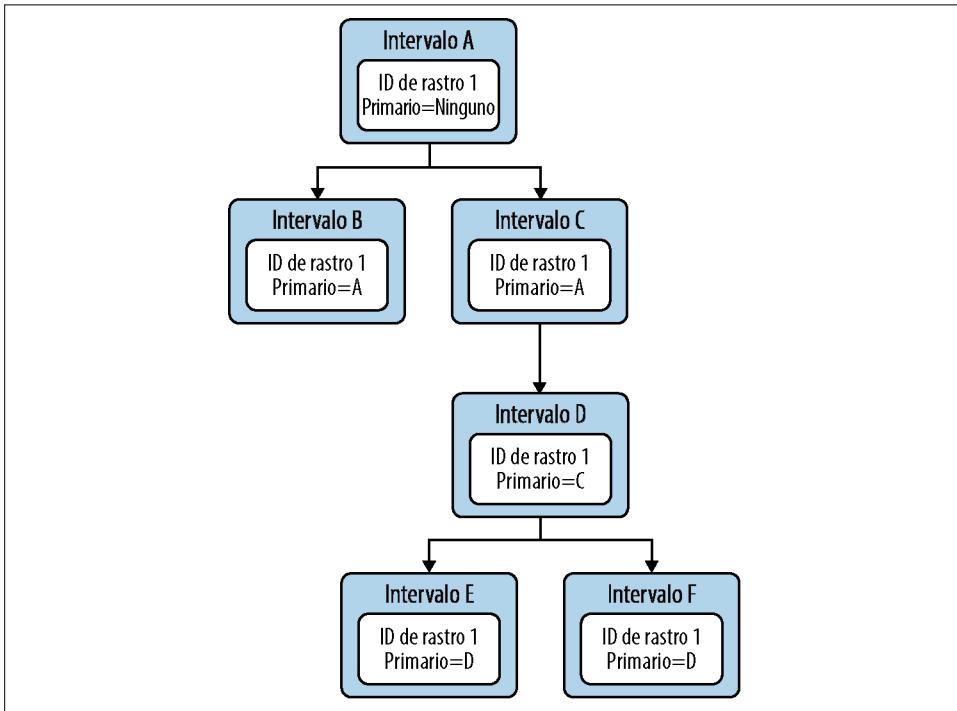


Figura 6-9. Anatomía de un intervalo

## Seguimientos

Un seguimiento representa todo el viaje de una solicitud en todos los servicios que alcanza durante su recorrido. Un único seguimiento completo normalmente consta de varios intervalos de la solicitud. Por ejemplo, la transacción completa del cliente de principio a fin consta de varios intervalos.

## Propagación del contexto

Con cada solicitud de un servicio a un servicio descendente, se pasa un identificador global de ejecución de solicitud como metadatos, que es único para cada solicitud. Este identificador global de ejecución de solicitud se usa más adelante para reconstruir la ejecución completa de la solicitud mediante la agrupación de los registros en función del identificador de la solicitud. El proceso de pasar el identificador de ejecución como metadatos de la solicitud se denomina *propagación de contexto distribuido*.

Los metadatos que se propagan a través de los intervalos también incluyen el ID de seguimiento y el ID del intervalo, junto con dos detalles adicionales: etiquetas y registros. Las *etiquetas* son pares de clave-valor que se pueden agregar a los intervalos para proporcionar información adicional, como códigos de error y detalles del host. Los *registros* también son pares de clave-valor que captan mensajes de registro específicos del intervalo y ayudan en la depuración.

## Muestreo

Recopilar todos los datos en la memoria y luego enviarlos a un back-end de seguimiento puede tener un impacto grave en la red, la latencia de la aplicación y el costo. Para evitar este problema, el sistema de seguimiento implementa el muestreo. El muestreo es el proceso de recopilar y almacenar solo un subconjunto de datos de seguimiento para evitar los costos de almacenamiento y la sobrecarga de rendimiento. Existen varios tipos de métodos de muestreo, que principalmente se clasifican como muestreo basado en la cabeza o en la cola. Por ejemplo, si desea recopilar  $x$  número de seguimientos por minuto, puede usar un tipo de muestreo conocido como *limitación de velocidad*. Del mismo modo, si desea obtener muestras de seguimientos cuando se destaca cierto tipo de error, puede usar el *muestreo sensible al contexto*.

## Arquitectura del sistema de seguimiento general y ensamblado de seguimiento

Para implementar un sistema de seguimiento en un entorno nativo de la nube, puede utilizar las bibliotecas existentes de open source (analizaremos esto con más profundidad en la siguiente sección), lo que lo ayudará a instrumentar su aplicación sin reinventar la rueda. En general, la mayoría de los sistemas de seguimiento tienen una arquitectura sencilla, como se muestra en la Figura 6-10.

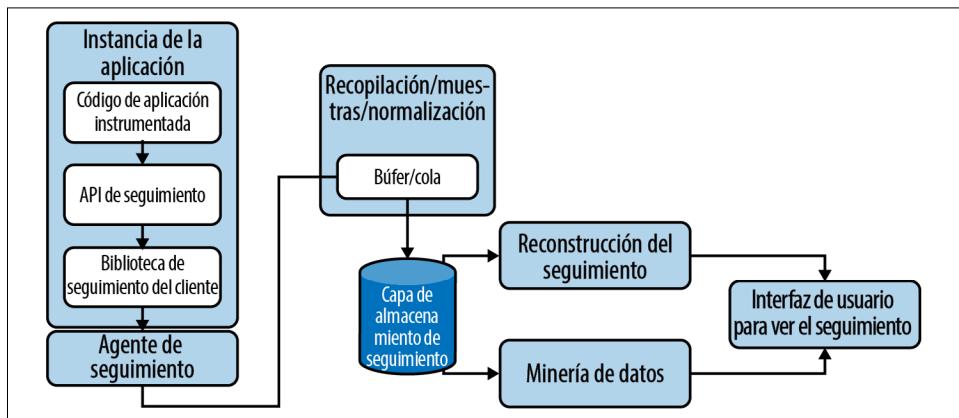


Figura 6-10. Arquitectura del sistema de seguimiento distribuido general

Para usar el seguimiento distribuido, primero necesita instrumentar el código de la aplicación mediante una biblioteca cliente. Las bibliotecas cliente están disponibles en muchos lenguajes, incluidos Java, Go y Python. La biblioteca envía los datos a un agente, que normalmente está disponible en la misma instancia o host. El agente es responsable de redirigir los seguimientos al back-end (por ejemplo, a la capa de almacenamiento a través de una capa de búfer asíncrona). Más tarde, los seguimientos se reconstruyen y muestran en la interfaz de usuario. Los seguimientos también se pueden usar para realizar más análisis e incorporación con técnicas de minería de datos.

Se debe revisar lo siguiente en una solicitud para ensamblar los seguimientos:

#### *Intervalo de solicitud entrante*

Cuando una solicitud entra en un servicio, el sistema de seguimiento comprueba si la solicitud tiene un encabezado de seguimiento adjunto. Si no se adjunta ninguno, se crea un intervalo raíz. De lo contrario, es necesario crear un intervalo secundario.

#### *Intervalo de solicitud saliente*

Si una solicitud sale de un servicio a un servicio diferente, primero se crea un intervalo y el servicio receptor continúa el seguimiento, como se describe en el intervalo de solicitud entrante.

Puede implementar el seguimiento distribuido en entornos de nube de varias maneras. En la siguiente sección, revisaremos algunas de ellas.

## **Normas de seguimiento, herramientas e instrumentación de código**

Con los años, el seguimiento distribuido ha evolucionado y se ha convertido en un método estándar para crear perfiles y supervisar aplicaciones en el mundo nativo de la nube. En 2016, *OpenTracing* se transformó en un proyecto CNCF cuyo único objetivo era proporcionar especificaciones independientes del proveedor para el seguimiento distribuido. Durante este período, Google trabajaba en un proyecto de la comunidad de open source llamado *OpenCensus*, que más tarde dirigió Microsoft.

OpenTracing y OpenCensus eran dos marcos que competían entre sí. Sus arquitecturas eran diferentes, pero básicamente resolvían el mismo problema. La presencia de dos estándares llevó a cierta incertidumbre con respecto al apoyo y las contribuciones, lo que tuvo como resultado que no se adoptaran en toda la industria. En mayo de 2019, la CNCF anunció la fusión de OpenTracing y OpenCensus en un único estándar conocido como *OpenTelemetry*. OpenTelemetry combina los beneficios de los estándares anteriores y cuenta con el respaldo de la comunidad. En esta sección, analizaremos cómo se puede instrumentar el código de aplicación básico con OpenTracing y OpenTelemetry.

Hoy en día, las dos herramientas de seguimiento más populares y prácticas son *Zipkin* y *Jaeger*. Zipkin fue el primer sistema de seguimiento desarrollado por la *infraestructura de seguimiento de Dapper* de Google. Jaeger se desarrolló en Uber y cuenta con el respaldo de la CNCF. Jaeger implementa la especificación OpenTracing y su método de implementación preferido es Kubernetes. Aquí nos limitaremos a analizar Jaeger.

Jaeger sigue estrechamente la arquitectura general de un sistema de seguimiento, puesto que consta de tres componentes principales: recopiladores, un servicio de consultas y una interfaz de usuario. Implementa un agente en cada host, que básicamente agrega los datos de seguimiento y los envía al recopilador (servicio de almacenamiento en búfer). Luego, los datos de seguimiento pueden almacenarse en una base de datos, de preferencia Elasticsearch o Cassandra.

Ahora revisemos algunas formas sencillas en las que podemos instrumentar código con diferentes estándares de seguimiento y Jaeger.

Por ejemplo, podemos usar un cliente Jaeger de Python:

```
$ pip install jaeger-client
```

A partir de ahí, podemos ejecutar todos los componentes de Jaeger en una sola imagen de Docker:

```
$ docker run -d -p6831:6831/udp -p16686:16686 jaegertracing/all-in-one:latest
```

Y podemos comprobar el contenedor Docker con docker ps:

```
$ ~ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
874b2a03ebaa      jaegertracing/all-in-one:latest   "/go/bin/all-in-one..."   About a minute ago
NAMES
jolly_mclean
```

Puede seleccionar <http://localhost:16686> localmente para acceder a la interfaz de usuario de Jaeger.

Ahora analizaremos cómo instrumentar código de aplicación sencillo con ayuda de OpenTracing. En el siguiente ejemplo, tenemos un programa de reserva de películas con un par de métodos que se encargan de comprobar si una película está disponible. Si es así, reserva las entradas:

```
import sys
import time
import logging
import random
from jaeger_client import Config
from opentracing_instrumentation.request_context import get_current_span, span_in_context

def initialize_tracer(service):
    logging.getLogger('').handlers = []
    logging.basicConfig(format='%(message)s', level=logging.DEBUG)
    config = Config(
        config={
            'sampler': {
                'type': 'const',
                'param': 1,
            },
            'logging': True,
        },
        service_name=service,
    )
    return config.initialize_tracer()

def booking_manager(movie):
    with tracer.start_span('booking') as span:
        span.set_tag('Movie', movie)
        with span_in_context(span):
            get_cinema_details = check_movie(movie)
            get_showtime_details = check_movie_showtime(get_cinema_details)
            book_movie_now(get_showtime_details)

def check_movie(movie):
    with tracer.start_span('CheckCinema', child_of=get_current_span()) as span:
        with span_in_context(span):
```

```

        num = random.randint(1,30)
        time.sleep(num)
        cinema_details = "Cinema Details"
        flags = ['false', 'true', 'false']
        random_flag = random.choice(flags)
        span.set_tag('error', random_flag)
        span.log_kv({'event': 'CheckCinema' , 'value': cinema_details })
        return cinema_details

    def check_movie_showtime( cinema_details ):
        with tracer.start_span('CheckShowtime', child_of=get_current_span()) as span:
            with span_in_context(span):
                num = random.randint(1,30)
                time.sleep(num)
                showtime_details = "Showtime Details"
                flags = ['false', 'true', 'false']
                random_flag = random.choice(flags)
                span.set_tag('error', random_flag)
                span.log_kv({'event': 'CheckCinema' , 'value': showtime_details })
                return showtime_details

    def book_movie_now(showtime_details):
        with tracer.start_span('BookShow', child_of=get_current_span()) as span:
            with span_in_context(span):
                num = random.randint(1,30)
                time.sleep(num)
                Ticket_details = "Ticket Details"
                flags = ['false', 'true', 'false']
                random_flag = random.choice(flags)
                span.set_tag('error', random_flag)
                span.log_kv({'event': 'CheckCinema' , 'value': showtime_details })

    assert len(sys.argv) == 2
    tracer = initialize_tracer('movie_booking')
    movie = sys.argv[1]
    booking_manager(movie)
    time.sleep(2)
    tracer.close()

```

En el código anterior, primero inicializamos el seguimiento con el método `initialize_tracer`, que establece la configuración para el registro y el muestreo. Lo siguiente que debemos tener en cuenta en el código es cómo usamos la instancia de seguimiento para comenzar un intervalo nuevo con `start_span` y usar `child_of` para iniciar nuevos intervalos secundarios hacia el intervalo raíz. Por ejemplo, el intervalo raíz se genera en el método `booking_manager` y los métodos restantes (p. ej., `check_movie_showtime` y `check_movie`) son principalmente intervalos secundarios del intervalo raíz.

También configuramos etiquetas y registros en los métodos. Por ejemplo:

```

-> span.set_tag('error', random_flag)

-> span.log_kv({'event': 'CheckCinema' , 'value': showtime_details })

```

Para simplemente ejecutar este programa en su consola de Python, pase un título ficticio de película como el argumento de la siguiente forma:

```
$ python jaeger_opentracing.py godfather
Initializing Jaeger Tracer with UDP reporter
Using sampler ConstSampler(True)
opentracing.tracer initialized to <jaeger_client.Tracer object at 0x10eeae410> \
[app_name=movie_booking]
Reporting span f99b147babd58321:5ca566beb40e89b0:931be7dc309045fd:1 movie_booking. \
CheckCinema
Reporting span f99b147babd58321:1ad5d00d2acbd02:931be7dc309045fd:1 movie_booking. \
CheckShowtime
Reporting span f99b147babd58321:f36b9959f34f61dc:931be7dc309045fd:1 movie_booking.BookShow
Reporting span f99b147babd58321:931be7dc309045fd:0:1 movie_booking.booking
```

Ahora puede acceder a la interfaz de usuario de Jaeger y buscar `movie_booking` en el menú desplegable del servicio. Puede hacer clic en el botón Find Traces (Buscar seguimientos) y ver los seguimientos que aparecen, como se muestra en la [Figura 6-11](#).



*Figura 6-11. Seguimientos del servicio de reserva de películas*

Puede ver los cuatro intervalos en la [Figura 6-11](#), en que `movie_booking` es el intervalo raíz y los intervalos restantes son secundarios. Como este es un programa de ejemplo, para fines de demostración, generamos aleatoriamente algunos errores en el código resultante. En entornos de producción, estos errores indicarían problemas que tienen lugar en el entorno (por ejemplo, se agota el tiempo de espera de Redis debido a un problema de I/O).

También podemos instrumentar nuestro código mediante OpenTelemetry, puesto que el uso de OpenTracing se está eliminando gradualmente. En primer lugar, tendríamos que instalar la API y el SDK de OpenTelemetry:

```
pip install opentelemetry-api
pip install opentelemetry-sdk
```

Para ver cómo funciona OpenTelemetry, mire el siguiente código de Python:

```
from opentelemetry import trace
from opentelemetry.exporter import jaeger
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchExportSpanProcessor

trace.set_tracer_provider(TracerProvider())

jaeger_exporter = jaeger.JaegerSpanExporter(
    service_name="my-helloworld-service",
    agent_host_name="localhost",
    agent_port=6831,
)

trace.get_tracer_provider().add_span_processor(
    BatchExportSpanProcessor(jaeger_exporter)
```

```

)
tracer = trace.get_tracer(__name__)

with tracer.start_as_current_span("foo"):
    with tracer.start_as_current_span("bar"):
        with tracer.start_as_current_span("tango"):
            print("Hello world from OpenTelemetry! Happy Tracing")

```

El código anterior es un script sencillo de Python tipo “hello world” que usa OpenTelemetry. Observe cómo utilizamos el exportador Jaeger para conocer la ubicación del agente de Jaeger.

Necesita Python 3 para ejecutar el script anterior:

```
$ python3 opentelemetry_simple.py
Hello world from OpenTelemetry! Happy Tracing
```

Puede ver los intervalos que se rellenan nuevamente en la interfaz de usuario en el mismo punto de conexión de Jaeger (<http://localhost:16686>) debajo del servicio `my-helloworld-service`.

De forma similar, puede instrumentar cualquier código en función del lenguaje de su aplicación. Por ejemplo, puede instrumentar una aplicación Python Flask de la siguiente manera:

```

import flask
import requests

from opentelemetry import trace
from opentelemetry.exporter import jaeger
from opentelemetry.instrumentation.flask import FlaskInstrumentor
from opentelemetry.instrumentation.requests import RequestsInstrumentor
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import (
    ConsoleSpanExporter,
    SimpleExportSpanProcessor,
)

trace.set_tracer_provider(TracerProvider())

jaeger_exporter = jaeger.JaegerSpanExporter(
    service_name="flask_app_example",
    agent_host_name="localhost",
    agent_port=6831,
)

trace.get_tracer_provider().add_span_processor(
    SimpleExportSpanProcessor(jaeger_exporter)
)

app = flask.Flask(__name__)
FlaskInstrumentor().instrument_app(app)
RequestsInstrumentor().instrument()

@app.route("/")
def hello():
    tracer = trace.get_tracer(__name__)
    with tracer.start_as_current_span("example-request"):
        requests.get("http://www.example.com")

```

```
    return "hello"

app.run(debug=True, port=5000)
```

Puede ejecutar el script anterior de la siguiente manera y presionar <http://localhost:5000> para generar seguimientos en la interfaz de usuario de Jaeger:

```
$ python3 opentelemetry_flask.py
* Serving Flask app "opentelemetry_flask" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 156-661-048
127.0.0.1 - - [30/Dec/2020 20:54:37] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [30/Dec/2020 20:54:37] "GET /favicon.ico HTTP/1.1" 404 -
```

Estas son algunas maneras en las que puede usar el seguimiento distribuido en sus entornos nativos de la nube para hacerlos observables. Aparte de los enfoques mencionados aquí, las mallas de servicios, como Istio, también proporcionan una arquitectura de implementación sidecar para realizar el seguimiento distribuido. Presentaremos las mallas de servicios en el [Capítulo 7](#). Para obtener más información sobre el seguimiento distribuido, consulte *Mastering Distributed Tracing* de Yuki Shkuro (Packt, 2019).

Ahora analicemos lo que Azure ofrece en el área de la supervisión.

## Azure Monitor

Además de las soluciones personalizadas antes mencionadas para la creación de sistemas observables, Microsoft Azure también ofrece *Azure Monitor*, una solución lista para usar, que puede ayudarlo a supervisar una aplicación y una infraestructura que se ejecutan en la nube y en entornos locales. Azure Monitor utiliza principalmente métricas y registros para descubrir cómo funciona su aplicación e identifica de forma proactiva los posibles problemas que la afectan, además de cualquier dependencia de recursos.

Azure Monitor tiene varias características que pueden ayudarlo a detectar diversos problemas en su aplicación e infraestructura, como se muestra aquí y se ilustra en la [Figura 6-12](#):

### *Application Insights*

Puede detectar y diagnosticar problemas en las aplicaciones y dependencias mediante la supervisión de la disponibilidad, el rendimiento y el uso de las aplicaciones web hospedadas en la nube o localmente.

### *Azure Monitor para contenedores y máquinas virtuales*

Puede supervisar el rendimiento de los contenedores que se ejecutan en AKS, además de la máquina virtual que los hospeda.

### Análisis de registros y consultas de registros

Puede usar los registros de Azure Monitor en función de Azure Data Explorer para consultar registros.

### Libros y paneles

Puede crear visualizaciones flexibles con libros y paneles para el análisis de datos, que le permitirán aprovechar varios recursos de datos en Azure.

### Métricas de Azure Monitor

Puede usar métricas de Azure Monitor para recopilar datos numéricos de su infraestructura y aplicación.

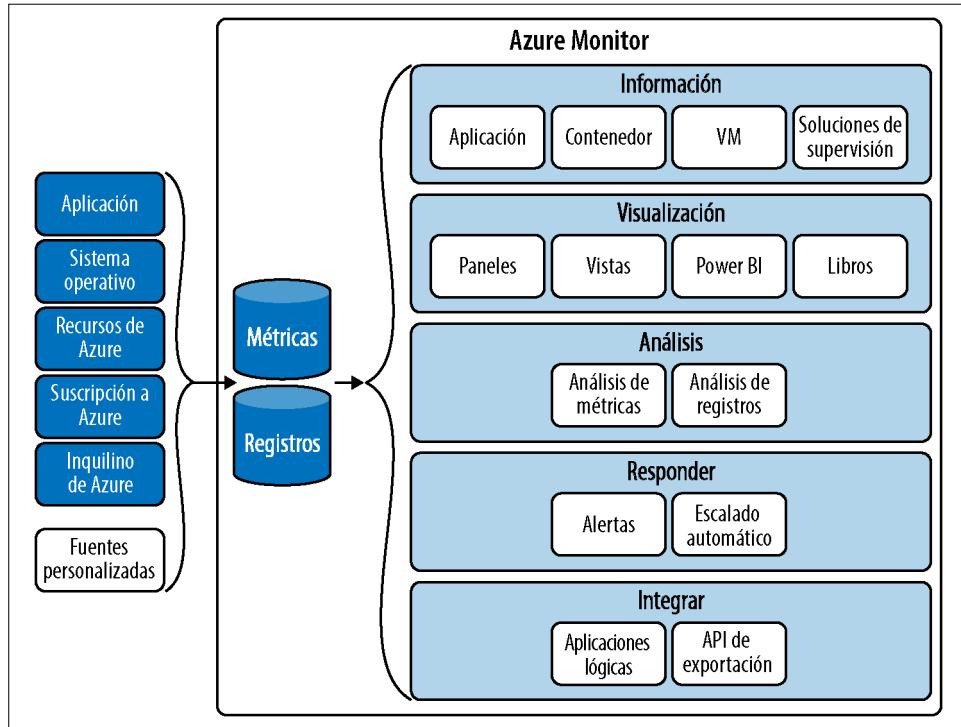


Figura 6-12. Azure Monitor

Azure Monitor también le permite consumir datos de seguimiento distribuidos. Hay dos formas principales de hacerlo. La primera es mediante el uso de la *vista de diagnóstico de transacciones*, que es similar a una pila de llamadas con una dimensión de tiempo agregada. La vista de diagnóstico de transacciones proporciona visibilidad en una sola transacción/solicitud, y es útil para encontrar la causa raíz de los problemas de confiabilidad y los cuellos de botella de rendimiento por solicitud. La segunda forma es mediante el uso de la *vista de mapa de aplicación*, que agrega muchas transacciones para mostrar una vista topológica de la forma en que interactúan los sistemas y cuáles son las tasas de rendimiento y errores promedio.

Además, puede integrar Azure Monitor con otros sistemas y crear una plataforma de observabilidad personalizada que use los datos de supervisión.

## Resumen

En este capítulo, presentamos la observabilidad como una necesidad cada vez mayor en el entorno nativo de la nube. Realizamos una introducción a la observabilidad y la forma en que complementa la supervisión. Además de mostrar cómo el desarrollo impulsado por la observabilidad es el nuevo norte verdadero para obtener información sobre su aplicación y la infraestructura en la nube. Analizamos los tres pilares de la observabilidad y presentamos las formas preferidas de registrar, supervisar y hacer seguimiento de los sistemas distribuidos modernos en Azure. Por último, analizamos brevemente la forma en que Azure ofrece sus soluciones integradas para realizar algunas tareas similares con Azure Monitor. Con este conocimiento bajo la manga, ahora puede crear con confianza sistemas observables en entornos nativos de la nube e integrar la observabilidad en sus aplicaciones.

En el siguiente capítulo, analizaremos cómo la malla de servicios y la detección de servicios ayudan en entornos nativos de la nube.



## CAPÍTULO 7

# Detección de servicios y malla de servicios: Descubrir nuevos territorios y cruzar fronteras

A medida que las aplicaciones modernas migraron a una arquitectura de microservicios, se convirtieron en escalables y eficientes en la nube. Los microservicios favorecen el desglose de una aplicación en componentes independientes y desacoplados. Normalmente, el desacoplamiento se logra separando las bases de código en aplicaciones independientes que se conectan mediante la red y las API o interfaces bien definidas que los microservicios utilizan para comunicarse. Una vez que traslada las aplicaciones a una arquitectura de microservicios, se abren muchas oportunidades: puede escalar cada servicio individual, desacoplar el ciclo de lanzamiento de cada servicio, elegir un lenguaje diferente para escribir los componentes individuales de su aplicación e, incluso, organizar la estructura de su equipo en función de estos servicios independientes.

Sin embargo, ahora que dividió su aplicación en aplicaciones más pequeñas, estos archivos binarios tienen su propio ciclo de vida. Este ciclo de vida se programa de forma independiente y finaliza de forma independiente según sea necesario, a la vez que mantiene la comunicación a través de la red. En este escenario, imagine que tiene muchos microservicios en ejecución y que se comunican entre sí. Probablemente, comenzaría a hacer varias preguntas, incluidas las siguientes:

- ¿Cómo se comunican los servicios entre sí cuando son de naturaleza dinámica? (Es decir, dado que los servicios se comunican a través de una red, la dirección IP no proporciona suficiente información para llegar a los servicios, puesto que se pueden terminar de forma independiente y reprogramar en un host diferente).
- ¿Cómo puedo controlar y administrar el tráfico o enrutarlo según sea necesario, incluidos escenarios avanzados como versiones de valor controlado, pruebas A/B e interrupción de circuito?

- ¿Cómo puedo garantizar la seguridad entre los servicios y cifrar el tráfico entre ellos?

A fin de resolver estos problemas, podría comenzar a escribir y compilar lógica adicional específica para la infraestructura dentro de su aplicación. Sin embargo, esto aumentaría el alcance de la aplicación más allá de las necesidades de su empresa, y tendría que implementar esta lógica con cada pila de microservicios que utilice.

Afortunadamente, estos problemas se han resuelto en el espacio de entorno nativo de la nube con la ayuda de las tecnologías de detección de servicios y malla de servicios, que ofrecen una forma mucho más flexible de administrar, proteger y proporcionar un control mucho mayor sobre los servicios.

*La detección de servicios* es un mecanismo mediante el cual los servicios se detectan entre sí para poder comunicarse. Se centra principalmente en encontrar la ubicación de red de un servicio en la nube. En los entornos tradicionales, donde tiene que ocuparse de servidores físicos, conoce las direcciones IP estáticas de sus servidores y puede usar un archivo de configuración sencillo para almacenar las IP, que pueden usar sus servicios a fin de ponerse en contacto con otros servicios. Esta configuración funciona bien en un entorno local o físico. Sin embargo, para las aplicaciones altamente escalables en la nube, este enfoque falla, porque ahora su aplicación es de naturaleza dinámica, lo que significa que se puede reiniciar, reprogramar o terminar según sea necesario. En estos escenarios, no se puede codificar la IP de forma rígida, porque puede cambiar. La detección de servicios ayuda mediante el almacenamiento de una asignación para cada nombre de servicio a la dirección IP en la nube.

Una *malla de servicios* se centra en solucionar la mayor complejidad que conlleva la comunicación entre servicios. Es una infraestructura dedicada que controla toda la comunicación de red de servicio a servicio dentro del entorno de nube. Proporciona alta visibilidad, resiliencia, administración de tráfico y control de seguridad con poco o ningún cambio en el código de la aplicación existente.

En este capítulo, analizaremos CoreDNS, una destacada plataforma de detección de servicios en el mundo nativo de la nube, que utiliza Azure Kubernetes Service (AKS) para la administración de DNS del clúster y la resolución de nombres. También abordaremos Istio, una malla de servicios destacada que se usa ampliamente en entornos de nube como Azure.

## Detección de servicios

La detección de servicios es un componente fundamental de un entorno moderno de nube nativa. Es responsable de garantizar que los servicios puedan buscarse y comunicarse entre sí a través de una red. Por lo general, la detección de servicios implica un *registro de servicios*, donde se deben registrar todos los servicios y aplicaciones del entorno de nube. El registro de servicios es el repositorio central que contiene la ubicación de los servicios. En un escenario típico, un nuevo servicio se registra primero con el registro de servicios cuando entra en el entorno. El registro de servicios actualiza la última ubicación de red del servicio y,

cuando un servicio de consumidor solicita conectarse al servicio, el registro de servicios entrega al consumidor una ubicación para el servicio.

En esta sección, exploraremos CoreDNS y veremos de qué manera se puede utilizar como una herramienta de detección de servicios en entornos de nube.

## Introducción a CoreDNS

CoreDNS es un servidor DNS extensible, que admite protocolos estándar DNS, DNS sobre TLS y DNS sobre gRPC. Fue creado por Miek Gieben en 2016 mediante la utilización de un marco de servidor, desarrollado como parte del [servidor web Caddy](#). Tiene una arquitectura de complementos muy configurable, flexible y extensible. CoreDNS experimentó una adopción más amplia después de recibir el soporte de la Cloud Native Computing Foundation (CNCF). En 2019, CoreDNS recibió el nivel de madurez “graduado” de la CNCF.<sup>1</sup>

CoreDNS es ahora el servidor DNS predeterminado oficial de Kubernetes 1.13 y las versiones posteriores, y reemplaza a kube-dns. Resuelve los problemas de confiabilidad y seguridad en la implementación original de kube-dns a través de varias diferencias clave:

- CoreDNS ejecuta un solo contenedor, mientras que kube-dns ejecuta tres: kubedns, dnsMasq y sidecar.
- CoreDNS está diseñado para su uso como un servidor DNS de uso general y es compatible con versiones anteriores de Kubernetes.
- CoreDNS es principalmente un proceso de Go que mejora la funcionalidad de kube-dns.

La arquitectura del complemento de CoreDNS garantiza que la funcionalidad del servidor DNS permanezca estable y permite agregar más funcionalidades mediante complementos. El usuario final puede definir varios servidores mediante la configuración de zonas para que sirvan en un puerto determinado. Cada servidor pasa las solicitudes a través de una cadena de complementos, que determina qué tipo de respuesta enviar:

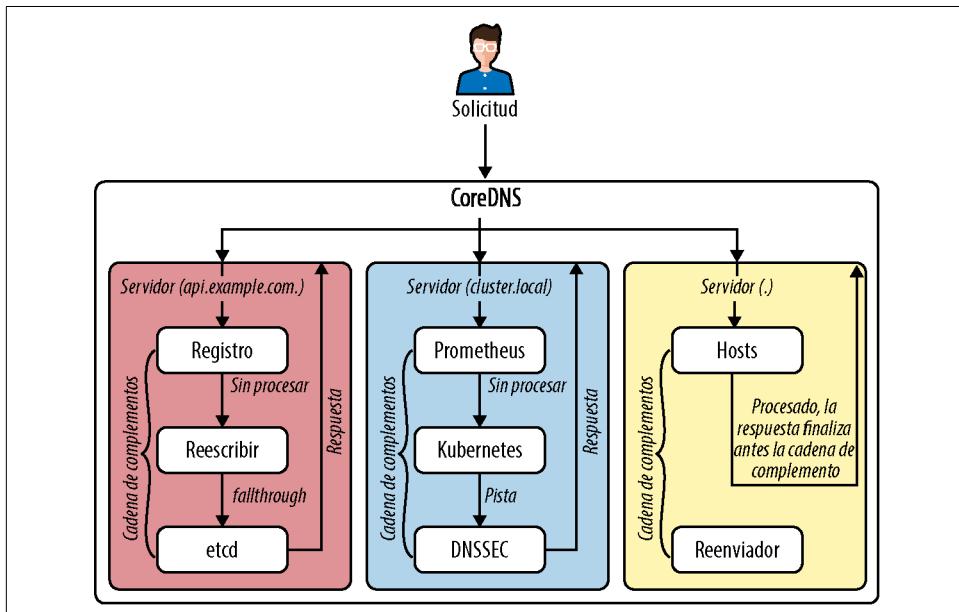
- Si varios servidores están configurados para escuchar en el puerto consultado, comprobará cuál tiene la zona más específica para la consulta mediante la coincidencia de sufijo más largo. Por ejemplo, si un usuario realiza una consulta para [www.health.abc.com](#) y hay dos servidores configurados como [abc.com](#) y [health.abc.com](#), la consulta se enrutaría al último servidor.
- Una vez que se encuentre un servidor adecuado, se enrutará a través de la cadena de complementos configurada para el servidor.

---

<sup>1</sup> La CNCF asigna diferentes niveles de madurez a los proyectos: sandbox, incubado y graduado, que corresponden a los niveles Innovadores, Primeros usuarios y Mayoría inicial del diagrama de Cruzando el abismo. El nivel de madurez es una señal que la CNCF asigna a un proyecto para identificar qué tipos de empresas deben adoptar el proyecto.

- Cada complemento inspeccionará la consulta y, en función de ella, puede ocurrir lo siguiente:
    - Si el complemento procesa la consulta, se devolverá una respuesta adecuada al cliente. El procesamiento de consultas se detiene justo aquí y no se llama a más complementos.
    - Si el complemento no procesa la consulta, se llamará al siguiente complemento en la cadena. Si el último complemento de la cadena tampoco puede procesar la consulta, CoreDNS devolverá un SERVFAIL al cliente que realiza la llamada.
    - Si la consulta se procesa con un método “fall-through” (de paso explícito), en el que también se llama a los otros complementos que están más abajo en la cadena para que revisen la consulta, la palabra clave **fallthrough** se utiliza para habilitarla.
    - Si la consulta se procesa mediante una sugerencia, siempre se llamará al siguiente complemento de la cadena. No obstante, proporcionará una sugerencia que le permita ver la respuesta que se devolverá al cliente.

La **Figura 7-1** muestra los pasos que se producen cuando CoreDNS procesa una consulta.



*Figura 7-1. Proceso de consulta de CoreDNS*

Ahora veremos cómo instalar CoreDNS y utilizar el archivo de configuración, también conocido como *Corefile*, para configurarlo.

## Instalación y configuración de CoreDNS

Puede instalar CoreDNS mediante el archivo binario, que está disponible en la [ubicación de GitHub](#) de este libro. Una vez que haya tomado el archivo binario para su SO, puede moverlo a su ubicación local. Por ejemplo, si descargó el archivo binario para macOS, `coredns_1.8.1_darwin_amd64.tgz`, puede descomprimir el archivo y moverlo a `/usr/local/bin`. Compruebe que CoreDNS funcione bien con el siguiente comando:

```
$ coredns -version
CoreDNS-1.8.1
darwin/amd64, go1.15.7, 95622f4
```

CoreDNS ofrece un conjunto de opciones de configuración que se controlan mediante un Corefile. Cuando se inicia CoreDNS, busca un archivo con el nombre *Corefile* en el directorio actual si no se pasa la marca `-conf`. El archivo define principalmente lo siguiente:

- Qué servidor escucha y en qué puerto
- Para qué zona está autorizado cada servidor
- Qué complementos se cargan en un servidor

Normalmente, un Corefile se ve así:

```
ZONE:[PORT] {
    [PLUGIN]...
}
```

donde ZONE representa la zona DNS, PORT es dónde se ejecuta el servidor de zona (por lo general, el puerto 53) y PLUGIN define los complementos que el usuario debe cargar.

Entre las entradas más comunes de un Corefile se encuentran los diversos *bloques de servidores* que definen un servidor dentro de CoreDNS (es decir, de qué forma se procesan las consultas para un nombre de dominio específico). Este es un ejemplo:

```
example.com {
}
```

El bloque de servidores anterior manejará todas las consultas que terminan en *example.com*, a menos que esté presente un bloque de servidor más específico, como *foo.example.com*. Para obtener un ejemplo más concreto, supongamos que tiene lo siguiente en su Corefile:

```
cloudnativeazure.io:53 {
    log
    errors
}

.:53 {
    forward . 8.8.8.8
    log
    errors
```

```
        cache  
    }
```

En este ejemplo, tenemos dos bloques de servidores para `cloudnativeazure.io` y bloques de servidores raíz (.). Ambos están escuchando en el puerto 53. Además, se definen varios complementos: `log`, para registrar cada consulta que CoreDNS recibe; `errors`, para registrar errores; y `cache`, para habilitar la caché de front-end (es decir, todos los registros, salvo las transferencias de zona y los metadatos se almacenarán en caché hasta los 3600). Puede ejecutar el Corefile anterior con solo tipear `coredns` en el mismo directorio en que se encuentra Corefile, o bien `coredns -conf Corefile`:

```
$ coredns -conf Corefile  
.53  
cloudnativeazure.io.:53  
CoreDNS-1.8.1  
darwin/amd64, go1.15.7, 95622f4
```

Si ejecuta `dig` para `cloudnativeazure.io` en otro terminal, verá lo siguiente:

```
$ ~ dig @127.0.0.1 cloudnativeazure.io:53  
; <>> DIG 9.10.6 <>> @127.0.0.1 cloudnativeazure.io:53  
; (1 server found)  
;; global options: +cmd  
;; Got answer:  
;; ->HEADER<- opcode: QUERY, status: NXDOMAIN, id: 48995  
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1  
  
;; OPT PSEUDOSECTION:  
; EDNS: version: 0, flags:; udp: 4096  
;; QUESTION SECTION:  
;cloudnativeazure.io:53. IN A  
  
;; AUTHORITY SECTION:  
. 1800 IN SOA a.root-servers.net. nstld.verisign-grs.com. 2021021001 \  
1800 900 604800 86400  
  
;; Query time: 82 msec  
;; SERVER: 127.0.0.1#53(127.0.0.1)  
;; WHEN: Wed Feb 10 23:45:28 IST 2021  
;; MSG SIZE rcvd: 126
```

Puede ver los registros en el terminal de CoreDNS de la siguiente manera:

```
$ coredns -conf Corefile  
.53  
cloudnativeazure.io.:53  
CoreDNS-1.8.1  
darwin/amd64, go1.15.7, 95622f4  
  
[INFO] 127.0.0.1:59741 - 48995 "A IN cloudnativeazure.io:53. udp 51 false 4096" NXDOMAIN \  
qr,rd,ra,ad 115 0.05160016s
```

Aquí solo cubrimos los aspectos básicos de CoreDNS. Si desea obtener más información, le sugerimos leer [Learning CoreDNS](#) de John Belamaric y Cricket Liu (O'Reilly, 2019).

Ahora, analizaremos cómo se usa CoreDNS en la detección de servicios en Kubernetes.

## Detección de servicios de Kubernetes con CoreDNS

Como mencionamos antes, CoreDNS es el mecanismo de detección de servicios predeterminado de Kubernetes v1.13 y versiones posteriores, y puede comenzar a usarlo desde el principio. Como puede ver en el siguiente bloque de código, los pods que se ejecutan en el espacio de nombres `kube-system` son del tipo `core-dns`:

```
$ kubectl get pods --namespace=kube-system -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP
NODE          NOMINATED NODE  READINESS GATES
coredns-748cdb7bf4-9xggk   1/1     Running   0        4h31m  10.244.0.3
  aks-agentpool-13363041-vmss000002  <none>   <none>
coredns-748cdb7bf4-f6glb   1/1     Running   0        4h30m  10.244.0.5
  aks-agentpool-13363041-vmss000002  <none>   <none>
coredns-autoscaler-868b684fd4-7ck6z  1/1     Running   0        4h30m  10.244.0.4
  aks-agentpool-13363041-vmss000002  <none>   <none>

$ kubectl get deployments --namespace=kube-system
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
coredns      2/2     2           2           4h31m
coredns-autoscaler  1/1     1           1           4h31m
```

El complemento CoreDNS Kubernetes observa los servicios de Kubernetes y los recursos de punto de conexión y almacena esos datos en caché. Cuando se realiza una solicitud, el complemento de CoreDNS verá si existe un recurso correspondiente al nombre solicitado y, a continuación, devolverá los datos correspondientes. Los registros de respuesta se crean sobre la marcha en función de las solicitudes entrantes mediante la característica *watch* del servidor de la API de Kubernetes.

Revisemos la forma en que CoreDNS ayuda en la detección de servicios en un clúster de Kubernetes. Supongamos que está ejecutando un pod de Nginx en un clúster de Kubernetes de la siguiente forma:

```
$ kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx  1/1     Running   0          3m20s

$ kubectl get services
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
kubernetes  ClusterIP  10.0.0.1      <none>        443/TCP      41m
nginx      ClusterIP  10.0.148.122  <none>        80/TCP       3m26s
```

Para acceder a este servidor Nginx, debe ejecutar un pod `infoblox/dnstoools` que tenga algunas utilidades DNS, como `dig` y `curl`. Para ejecutar este pod, puede usar el siguiente comando:

```
$ kubectl run --restart=Never -it --image infoblox/dnstoools dnstoools
If you don't see a command prompt, try pressing enter.
dnstoools#
```

El comando anterior lo registrará dentro del contenedor recién activado y le dejará un símbolo del sistema.

En Kubernetes, la detección de servicios basada en DNS define una especificación, o un esquema DNS, que define el nombre DNS utilizado para ubicar los servicios que se ejecutan en el clúster. Todos los registros DNS de un clúster de Kubernetes se encuentran en un solo dominio al que se hace referencia como el *dominio del clúster*, que generalmente se establece en `cluster.local` de forma predeterminada en la mayoría de los clústeres de Kubernetes. La especificación DNS define que para cada IP asignada a un servicio (es decir, la IP del clúster), existe un registro A que contiene la IP del clúster con un nombre derivado del nombre del servicio y el espacio de nombres como `service.namespace.svc.cluster-domain`. Así, para acceder al pod de Nginx que creamos, puede emitir el siguiente comando del contenedor `infoblox/dnstable`:

```
dnstable# curl nginx.default.svc.cluster.local
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
dnstable#
```

También puede emitir un comando `dig` a fin de ver el registro A para el servicio:

```
dnstable# dig nginx.default.svc.cluster.local

; <>> DiG 9.11.3 <>> nginx.default.svc.cluster.local
;; global options: +cmd
;; Got answer:
;; WARNING: .local is reserved for Multicast DNS
;; You are currently testing what happens when an mDNS query is leaked to DNS
;; ->>HEADER<- opcode: QUERY, status: NOERROR, id: 52488
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 5086e790e261bf20 (echoed)
;; QUESTION SECTION:
;nginx.default.svc.cluster.local. IN      A
```

```
;; ANSWER SECTION:  
nginx.default.svc.cluster.local. 5 IN A 10.0.148.122  
  
;; Query time: 1 msec  
;; SERVER: 10.0.0.10#53(10.0.0.10)  
;; WHEN: Sat Feb 27 07:37:10 UTC 2021  
;; MSG SIZE rcvd: 119
```

Microsoft Azure también proporciona DNS como servicio, a través de un servicio de hospedaje conocido como Azure DNS. Lo examinaremos a continuación.

## Azure DNS

Azure DNS proporciona principalmente la infraestructura de Microsoft Azure a fin de manejar resoluciones de nombre para dominios DNS. Puede usar Azure DNS para hospedar su dominio DNS y administrar sus registros DNS. Al hospedar sus dominios en Azure, puede administrar sus registros DNS con las mismas credenciales, API, herramientas y facturación que sus otros servicios de Azure.

Azure DNS se puede utilizar para resolver nombres de host en dominios públicos y privados. La característica de DNS pública se puede utilizar para hospedar a través de Azure su nombre de dominio comprado previamente, mientras que el DNS privado le permite administrar y resolver los nombres de dominio dentro de las redes virtuales.

CoreDNS también proporciona un complemento específico de Azure que se utiliza para atender zonas de Azure DNS. Puede habilitar el complemento de Azure de la siguiente manera:

```
azure AZURE_RESOURCE_GROUP:ZONE... {  
    tenant <TENANT_ID>  
    client <CLIENT_ID>  
    secret <CLIENT_SECRET>  
    subscription <SUBSCRIPTION_ID>  
    environment <ENVIRONMENT>  
    fallthrough [ZONES...]  
    access private  
}
```

En el código anterior:

- AZURE\_RESOURCE\_GROUP:ZONE es el grupo de recursos al que pertenecen las zonas hospedadas en Azure y ZONE es la zona que contiene datos.
- CLIENT\_ID y CLIENT\_SECRET son las credenciales de Azure y tenant especifica el TENANT\_ID que se usará. SUBSCRIPTION\_ID es el ID de suscripción.
- environment especifica el ENVIRONMENT de Azure.

- **fallthrough** especifica que, si la zona coincide y no se puede generar ningún registro, la solicitud se debe pasar al siguiente complemento. Si se omite **ZONES**, el método fall-through se produce para todas las zonas para las que el complemento está autorizado.
- **access** especifica si la zona DNS es pública (**public**) o privada (**private**). El valor predeterminado es **public**.

Ya abordamos el rol y la importancia de la detección de servicios en entornos nativos de nube distribuida, así que analicemos lo que una malla de servicios aporta a la tabla.

## La malla de servicios

Una malla de servicios es una herramienta que agrega observabilidad, confiabilidad y seguridad en el nivel de plataforma de las aplicaciones nativas de la nube. Por lo general, una malla de servicios se implementa como un conjunto escalable de proxies de red implementados como sidecar junto a la aplicación. Estos proxies son principalmente responsables de controlar la comunicación entre los microservicios y operan en la capa 7 de la pila OSI. Junto con varias características adicionales, una malla de servicios también hace lo siguiente:

- Proporciona administración de tráfico a través de equilibrio de carga, limitación de velocidad, cambio de tráfico e interruptores.
- Ayuda en la administración del ciclo de versiones de aplicaciones con versiones de valor controlado.
- Agrega algunas comprobaciones de confiabilidad, como las comprobaciones de estado, los reintentos y los tiempos de espera inmediatos.
- Ayuda con la inserción de errores y el enrutamiento de depuración.
- Proporciona seguridad a través de directivas de TLS y ACL.
- Proporciona métricas, registros y seguimientos automáticos para todo el tráfico dentro de un clúster.
- Protege la comunicación de servicio a servicio en un clúster mediante la autenticación y autorización sólidas y basadas en la identificación.

La arquitectura general de una malla de servicios consta de dos componentes de alto nivel: un plano de datos y un plano de control. En la [Figura 7-2](#), se muestran cuatro servicios (A - D) implementados en una malla de servicios.

Como puede ver, cada instancia de servicio tiene una instancia de proxy sidecar que se encarga de todo el flujo y la administración del tráfico. Esto básicamente significa que cuando el servicio A quiere comunicarse con el servicio C, por ejemplo, el servicio A inicia el flujo de comunicación a través de su proxy sidecar local, que luego llega al proxy de sidecar del servicio C. De esta manera, el servicio no reconoce el proxy de red local y permanece aislado de la red más grande. Por lo tanto, cada vez que un servicio necesita comunicarse con otro, el proxy intercepta la solicitud y, luego, la reenvía al receptor.

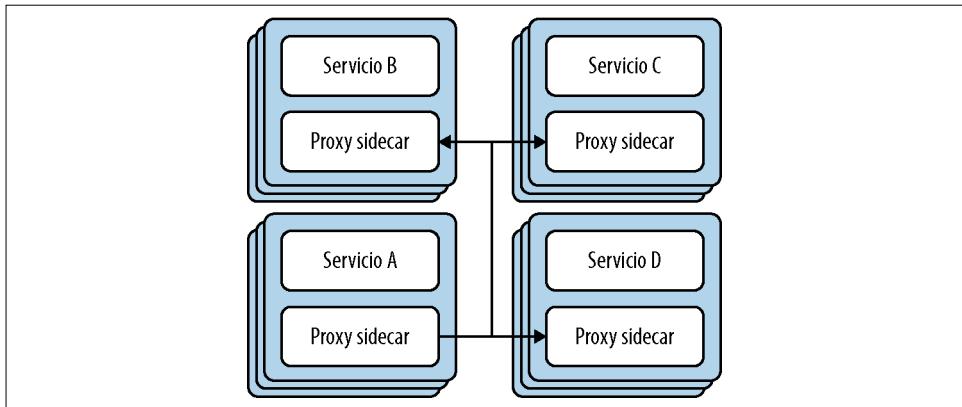


Figura 7-2. Arquitectura general de la malla de servicios

La colección de todos los proxies sidecar constituye el *plano de datos* de la arquitectura de malla de servicios. El plano de datos es responsable de lo siguiente:

- Reenvío de la solicitud
- Detección de servicios
- Revisión de estado
- Enrutamiento de solicitudes de servicio
- Equilibrio de carga
- Autenticación y autorización
- Proporcionar observabilidad

Por lo tanto, el plano de datos concierne fundamentalmente a todas las solicitudes de red o paquetes en el clúster.

Por otro lado, el plano de control proporciona la política y los detalles de configuración de todos los proxies sidecar (planos de datos) en la malla de servicios. El plano de control le permite controlar y configurar los planos de datos. Toda la configuración de alto nivel se envía al plano de control, lo que luego se traduce en la configuración específica del plano de datos. El plano de control ayuda al plano de datos mediante lo siguiente:

- Enrutamiento entre servicios
- Rellenado de datos de detección de servicios para uso del plano de datos
- Especificación de entradas para las configuraciones responsables del equilibrio de carga, la interrupción de circuito y los tiempos de espera
- Configuración de los parámetros de validación, autenticación y autorización en el clúster

En la siguiente sección, examinaremos Istio, que actúa como el plano de control para una malla de servicios en un entorno nativo de nube moderna.

## Introducción a Istio

A medida que las aplicaciones monolíticas se convertían en microservicios, los desarrolladores y los equipos de operaciones experimentaban desafíos con respecto a la forma en que los servicios se comunicarían entre ellos en la malla de servicios. A medida que una malla de servicios aumenta de tamaño, se vuelve difícil de administrar y mantener. También resulta difícil entender cómo funcionan en conjunto las diferentes partes en movimiento.

Istio es una malla de servicios open source que simplifica la comunicación entre los microservicios en aplicaciones nativas de nube distribuida. Proporciona conjuntos de características integradas enriquecidas, que aplican patrones de resiliencia, como reintentos, interruptores, tráfico de modelado, comportamiento de enrutamiento, implementación de valores controlados y más. Además, Istio le permite crear fácilmente una red de microservicios implementados mediante el equilibrio de carga, la supervisión y la autenticación, con casi ningún cambio en el código de la aplicación.

Como analizamos en la sección anterior, una arquitectura de malla de servicios consta de un plano de datos y un plano de control. Istio utiliza Envoy como su plano de datos e istiod como su plano de control (como se muestra en la [Figura 7-3](#)):

### *Envoy (plano de datos)*

Envoy es un proxy de alto rendimiento escrito en C++ que intercepta todo el tráfico entrante y saliente para cada aplicación en la malla de servicios. Se implementa como un sidecar junto con la aplicación y proporciona una gran cantidad de características, incluida la terminación de TLS, el equilibrio de carga, la interrupción de circuito, las comprobaciones de estado, la detección dinámica de servicios, la inserción de errores y mucho más. Este modelo de proxy sidecar le permite utilizar Istio como una malla de servicios sin ningún cambio de código.

### *Istiod (plano de control)*

Istiod administra y configura los proxies. Istiod consta de tres subcomponentes:

- Pilot, responsable de configurar los proxies de Envoy en tiempo de ejecución
- Citadel, responsable de la emisión y rotación de certificados
- Galley, responsable de validar, ingerir, agregar, transformar y distribuir la configuración dentro de Istio

Istiod convierte las reglas de enrutamiento de alto nivel que controlan el tráfico en una configuración específica de Envoy y se propaga a los sidecars de Envoy en tiempo de ejecución.

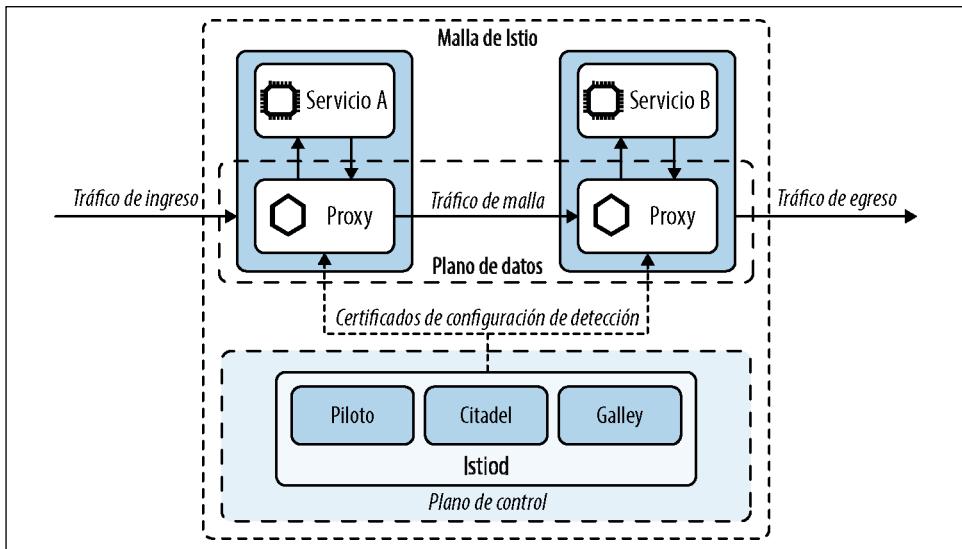


Figura 7-3. Arquitectura de Istio

Examinemos la forma en que puede usar Istio y Envoy en un entorno de Azure Kubernetes.

## Instalación de Istio en Azure Kubernetes Service

Puede instalar Istio en un clúster de Kubernetes de tres maneras:

- Istio proporciona una herramienta de línea de comandos llamada *istioctl*, que se puede utilizar para proporcionar un nivel enriquecido de personalización al plano de control de Istio y al plano de datos del proxy de Envoy.
- Puede utilizar el operador de Kubernetes para Istio a fin de administrar la instalación a través de las definiciones de recursos personalizadas (CRDs) de la API de Kubernetes.
- Puede usar gráficos de Helm para instalar Istio directamente.

Como los gráficos de Helm también se utilizan indirectamente en los métodos 1 y 2, los usaremos aquí para instalar Istio.

### Instalación de Istio con Helm

Para instalar Istio con Helm, necesita un clúster de AKS que ya se esté ejecutando (revise el [Capítulo 5](#) si necesita un repaso sobre cómo activar un clúster de AKS en Azure). Siga estos pasos para instalar Istio en un clúster de AKS:

1. Descargue Istio. Para ello, ejecute los siguientes comandos desde su equipo local:

```
$ curl -L https://istio.io/downloadIstio | ISTIO_VERSION=1.9.2 TARGET_ARCH=x86_64 sh -  
$ cd istio-1.9.2
```

2. Cree un nuevo espacio de nombres para los componentes de Istio, llamado `istio-system`:

```
$ kubectl create namespace istio-system  
namespace/istio-system created
```

3. Instale el gráfico base de Istio que contiene recursos de todo el clúster utilizados por el plano de control `istiod`:

```
$ helm install istio-base manifests/charts/base -n istio-system  
NAME: istio-base  
LAST DEPLOYED: Sun Mar 28 15:59:16 2021  
NAMESPACE: istio-system  
STATUS: deployed  
REVISION: 1  
TEST SUITE: None
```

4. Instale el gráfico de detección de Istio que implementa el servicio de `istiod`:

```
$ helm install istiod manifests/charts/istio-control/istio-discovery -n istio-system  
NAME: istiod  
LAST DEPLOYED: Sun Mar 28 16:02:23 2021  
NAMESPACE: istio-system  
STATUS: deployed  
REVISION: 1  
TEST SUITE: None
```

5. Compruebe la instalación:

```
$ helm list -A  
NAME           NAMESPACE      REVISION UPDATED                         STATUS  
CHART  
istio-base     istio-system   1        2021-03-28 15:59:16.429126 +0530 IST  deployed  
    base-1.9.2  
istiod         istio-system   1        2021-03-28 16:02:23.950541 +0530 IST  deployed  
    istio-discovery-1.9.2  
$ kubectl get pods -n istio-system  
NAME                  READY   STATUS    RESTARTS   AGE  
istiod-6d68c86c8d-5nv2r  1/1     Running   0          10m
```

Es importante asegurarse de que cada vez que se implemente un nuevo pod en el clúster, también se inserte un proxy sidecar de Istio (proxy Envoy) en el mismo pod, de manera que pueda aprovechar todas las características de Istio. Analizaremos cómo hacerlo en la siguiente sección.

## Inserción automática del proxy sidecar (proxy Envoy)

Para aprovechar al máximo Istio como una malla de servicios, los pods en el clúster también deben utilizar el proxy sidecar de Istio. `istio-proxy`, que es básicamente el proxy Envoy, se puede implementar de dos maneras: manual o automáticamente. El método manual le permite modificar la configuración de inserción y proxy. El método automático, como su nombre lo indica, permite una configuración automática de proxy en el momento de la creación del pod mediante el uso de un controlador de admisión en evolución. Aunque el método manual es útil cuando se necesita una configuración específica, se prefiere el método automático. A continuación, veremos cómo usar el método automático.



### Controladores de admisión de Kubernetes

Un controlador de admisión es un fragmento de código que intercepta las solicitudes al servidor de la API de Kubernetes antes de que el objeto se conserve, pero después de que la solicitud se autentique y autorice. Kubernetes proporciona una gama de controladores de admisión que se compilan en la biblioteca `kubeapiserver`.

Un `MutatingAdmissionController` puede modificar los objetos que admite. Es decir, llama a cualquier webhook en evolución que coincida con la solicitud.

Para utilizar la inserción automática de sidecar, utilice la etiqueta `istio-injection` en el espacio de nombres deseado y defínala en `enabled`. Esto permite que los pods nuevos creados en ese espacio de nombres tengan un sidecar de proxy que se implemente junto con la aplicación. Para comprender mejor cómo sucede esto, primero implementaremos un pod de Nginx sin inserción de sidecar y, luego, habilitaremos la inserción de sidecar de manera que, con las implementaciones futuras, el proxy sidecar se inserte automáticamente.

Supongamos que implementamos Nginx con un pod:

```
$ kubectl apply -f nginx_deploy.yaml
deployment.apps/nginx-deployment created

$ kubectl get deployment -o wide
NAME           READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES      SELECTOR
nginx-deployment   1/1     1           1          105s  nginx        nginx:1.14.2   app=nginx

$ kubectl get pod
NAME                  READY   STATUS    RESTARTS   AGE
nginx-deployment-6b474476c4-bqgh8   1/1     Running   0          4m5s
```

Etiquetemos el espacio de nombres `istio-injection=enabled`:

```
$ kubectl label namespace default istio-injection=enabled --overwrite
namespace/default labeled
$ kubectl get namespace -L istio-injection
NAME      STATUS   AGE   ISTIO-INJECTION
default   Active   24m   enabled
istio-system   Active   18m
```

```
kube-node-lease Active 25m
kube-public Active 25m
kube-system Active 25m
```

Dado que la inserción se produce cuando se crea el pod, podemos cerrar el pod de Nginx e iniciar uno nuevo, solo que esta vez el pod tendrá un sidecar insertado automáticamente:

```
$ kubectl delete pod -l app=nginx
pod "nginx-deployment-6b474476c4-bqgh8" deleted
```

```
$ kubectl get pod -l app=nginx
NAME                  READY   STATUS    RESTARTS   AGE
nginx-deployment-6b474476c4-pfgqv   2/2     Running   0          57s
```

Tenga en cuenta que la implementación ha activado dos contenedores. Este es el proxy sidecar que se ejecuta en el pod. Puede comprobarlo con el comando `describe` de la siguiente forma:

```
$ kubectl describe pod -l app=nginx

Events:
Type      Reason     Age       From               Message
----      ----      --       ----              -----
Normal    Scheduled  <unknown> default-scheduler        Successfully
          assigned default/nginx-deployment-6b474476c4-pfgqv to aks-agentpool-20139558-vmss000001
Normal    Pulling    2m35s    kubelet, aks-agentpool-20139558-vmss000001  Pulling image
          "docker.io/istio/proxyv2:1.9.2"
Normal    Created    2m34s    kubelet, aks-agentpool-20139558-vmss000001  Created
          container nginx
Normal    Created    2m34s    kubelet, aks-agentpool-20139558-vmss000001  Created
          container istio-init
Normal    Started    2m34s    kubelet, aks-agentpool-20139558-vmss000001  Started
          container istio-init
Normal    Pulled     2m34s    kubelet, aks-agentpool-20139558-vmss000001  Container image
          "nginx:1.14.2" already present on machine
Normal    Pulled     2m34s    kubelet, aks-agentpool-20139558-vmss000001  Successfully
          pulled image "docker.io/istio/proxyv2:1.9.2"
Normal    Started    2m34s    kubelet, aks-agentpool-20139558-vmss000001  Started
          container nginx
Normal    Pulling    2m34s    kubelet, aks-agentpool-20139558-vmss000001  Pulling image
          "docker.io/istio/proxyv2:1.9.2"
Normal    Pulled     2m33s    kubelet, aks-agentpool-20139558-vmss000001  Successfully
          pulled image "docker.io/istio/proxyv2:1.9.2"
Normal    Created    2m33s    kubelet, aks-agentpool-20139558-vmss000001  Created
          container istio-proxy
Normal    Started    2m33s    kubelet, aks-agentpool-20139558-vmss000001  Started
          container istio-proxy
```

Ahora que sabe cómo implementar el proxy sidecar, puede usar Istio para la administración de tráfico, la seguridad, la aplicación de políticas y la observabilidad. No analizaremos en forma detallada cada uno de estos aspectos, puesto que están más allá del ámbito de este libro.

Cuando use una malla de servicios, debe ser capaz de visualizarla y administrarla. En la siguiente sección, nos centraremos en Kiali, una consola de administración para las mallas de servicios basadas en Istio.

## Administración de mallas de servicios de Istio con Kiali

Kiali es una consola de administración para Istio que le permite operar y configurar fácilmente una malla de servicios con una mayor observabilidad en su entorno. Kiali proporciona una imagen clara de la malla de servicios a través de paneles que utilizan la topología de tráfico para mostrar su estructura y estado. En esta sección, analizaremos cómo instalar Kiali en un clúster de Kubernetes para obtener visibilidad de la malla de servicios de Istio.

Comenzaremos por instalar la gateway de entrada y salida para Istio desde el directorio *istio-1.9.2* que usamos cuando instalamos Istio antes:

```
Istio-1.9.2 $ helm install istio-ingress manifests/charts/gateways/istio-ingress -n \
    istio-system
NAME: istio-ingress
LAST DEPLOYED: Mon Apr  5 15:19:01 2021
NAMESPACE: istio-system
STATUS: deployed
REVISION: 1
TEST SUITE: None

Istio-1.9.2 $ helm install istio-egress manifests/charts/gateways/istio-egress -n \
    istio-system
NAME: istio-egress
LAST DEPLOYED: Mon Apr  5 15:19:31 2021
NAMESPACE: istio-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Ahora podemos usar el gráfico de Helm para instalar Kiali:

```
$ kubectl create namespace kiali-operator
namespace/kiali-operator created
$ istio-1.9.2 helm install --set cr.create=true --set cr.namespace=istio-system \
--namespace kiali-operator --repo https://kiali.org/helm-charts \
kiali-operator kiali-operator

NAME: kiali-operator
LAST DEPLOYED: Mon Apr  5 15:27:03 2021
NAMESPACE: kiali-operator
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Welcome to Kiali! For more details on Kiali, see: https://kiali.io

The Kiali Operator [v1.32.0] has been installed in namespace [kiali-operator]. It will be
ready soon.
You have elected to install a Kiali CR in the namespace [istio-system]. You should be able
to access Kiali soon.

If you ever want to uninstall the Kiali Operator, remember to delete the Kiali CR first
before uninstalling the operator to give the operator a chance to uninstall and remove all
the Kiali Server resources.

(Helm: Chart=[kiali-operator], Release=[kiali-operator], Version=[1.32.0])
```

Esto instalará el último operador de Kiali junto con el recurso personalizado (CR) de Kiali, que es básicamente un archivo YAML que contiene la configuración de Kiali.

Para acceder al panel de Kiali, primero tenemos que exponerlo correctamente a Internet. Para hacerlo, necesitamos obtener la dirección de gateway de entrada de la siguiente forma:

```
$ export INGRESS_HOST=$(kubectl -n istio-system get service istio-ingressgateway -o \
  jsonpath='{.status.loadBalancer.ingress[0].ip}')
$ export INGRESS_DOMAIN=${INGRESS_HOST}.nip.io
```

Ahora tenemos que crear un certificado autofirmado:

```
$ CERT_DIR=/tmp/certs
$ mkdir -p ${CERT_DIR}
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -subj "/O=example Inc./ \
  CN=*.${INGRESS_DOMAIN}" -keyout ${CERT_DIR}/ca.key -out ${CERT_DIR}/ca.crt
$ openssl req -out ${CERT_DIR}/cert.csr -newkey rsa:2048 -nodes -keyout \
  ${CERT_DIR}/tls.key -subj "/CN=*.${INGRESS_DOMAIN}/O=example organization"
$ openssl x509 -req -days 365 -CA ${CERT_DIR}/ca.crt -CAkey ${CERT_DIR}/ca.key \
  -set_serial 0 -in ${CERT_DIR}/cert.csr -out ${CERT_DIR}/tls.crt
$ kubectl create -n istio-system secret tls telemetry-gw-cert --key=${CERT_DIR}/tls.key \
  --cert=${CERT_DIR}/tls.crt
```

En este punto, podemos exponer a Kiali mediante las siguientes configuraciones, que crearán una gateway, un servicio virtual y una regla de destino:

- **Gateway** representa un plano de datos que enruta el tráfico a un clúster de Kubernetes mediante la configuración de un equilibrador de carga para el tráfico HTTP/TCP, independientemente de dónde se ejecutará.
- **VirtualService** define un conjunto de reglas de enrutamiento de tráfico que se aplican cuando se aborda un host. Una regla de enrutamiento puede coincidir con cierto tipo de tráfico, y cuando se produce una coincidencia para una regla, el tráfico se envía a un servicio de destino con nombre.
- **DestinationRule** configura el conjunto de directivas que se aplicarán al reenviar el tráfico a un servicio.

Aquí está el código:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: kiali-gateway
  namespace: istio-system
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
        name: https-kiali
        protocol: HTTPS
      tls:
        mode: SIMPLE
        credentialName: telemetry-gw-cert
```

```

hosts:
- "kiali.${INGRESS_DOMAIN}"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: kiali-vs
  namespace: istio-system
spec:
  hosts:
  - "kiali.${INGRESS_DOMAIN}"
  gateways:
  - kiali-gateway
  http:
  - route:
    - destination:
        host: kiali
        port:
          number: 20001
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: kiali
  namespace: istio-system
spec:
  host: kiali
  trafficPolicy:
    tls:
      mode: DISABLE
---
EOF

```

Ahora puede acceder a la interfaz de usuario de Kiali en [https://kiali.\\$\{INGRESS\\_DOMAIN\}](https://kiali.$\{INGRESS_DOMAIN\}) (en nuestro caso es <https://kiali.40.118.246.227.nip.io>). Cuando vaya a esta URL, se le mostrará la pantalla de inicio de sesión de Kiali, donde deberá ingresar el token para el operador de Kiali. Puede obtener el token de la siguiente forma:

```

$ kubectl get secrets --namespace=kiali-operator -o wide
NAME                           TYPE                                  DATA   AGE
default-token-7ng8l            kubernetes.io/service-account-token  3      142m
kiali-operator-token-jj88t     kubernetes.io/service-account-token  3      141m
sh.helm.release.v1.kiali-operator.v1  helm.sh/release.v1           1      141m

$ ~ kubectl describe secret kiali-operator-token-jj88t --namespace=kiali-operator
Name:         kiali-operator-token-jj88t
Namespace:    kiali-operator
Labels:      <none>
Annotations: kubernetes.io/service-account.name: kiali-operator
              kubernetes.io/service-account.uid: 9a30ba7a-7d21-484d-9e7a-0a38458690a4

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:  1765 bytes
namespace: 14 bytes

```

```

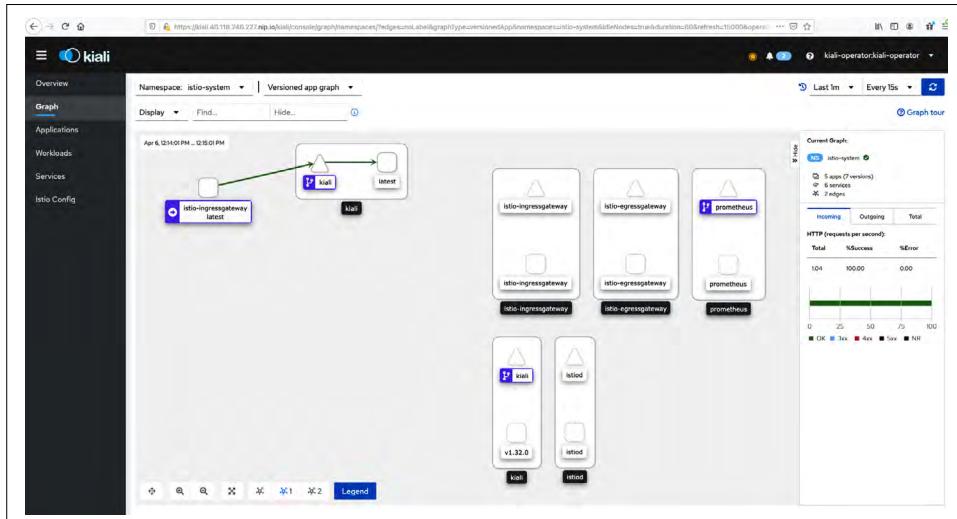
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6InVvQXNpV01QRG1fRjhma0RRZjVIWXBNVHRnZDhJdHhoOXI2RmQ2
eGxDZDQifQ.eyJpc3MiOiJrdWJlc5ldGvZL3NlcnPzY2VhY2NvdW50Iiwi3ViZXJuZXRLcy5pbv9zZXJ2aWNLYWnb
3VudC9uYW1l3BhY2UiOijraWFsaS1vcGVyYXRvcIiS1mt1YmVybmv0ZXMuaw8vc2VydmljZWfjY291bnQvc2Vjcmv0L
m5hbWUiOijraWFsaS1vcGVyYXRvcIi0b2tlbi1qaig4dCIS1mt1YmVybmv0ZXMuaw8vc2VydmljZWfjY291bnQvc2Vyd
mljZS1hY2NvdW50LmhbwUiOijraWFsaS1vcGVyYXRvcIiS1mt1YmVybmv0ZXMuaw8vc2VydmljZWfjY291bnQvc2Vyd
mljZS1hY2NvdW50LnVpZCI6IjlhMzBiYTdhLTdkMjEtNDg0ZC05TdhLTBhMzg0NTg20TBhNCIsInN1iI6InN5c3Rl
TpZXJ2aWNlyMNjb3VudprawFsaS1vcGVyYXRvcjprawFsaS1vcGVyYXRvcij9.uj6056HMQGv00QIK100Mj1PC5rW
MpYZL0jKiG27EBurhUcZ73ngr1S0IfzcNrnJluWukMPzzK0zVvrUUMf6biPb0MieWn001HX6mfH0zNgXRAxdP-0Ape
hmEiCyENQS4DEQL2Eg8muuaFsExuwzpx8LAibdLRpbI7sQa4P5B7he2Huol5FzJsoySjaKcP6eJG0aIbscIz-qtcDm3
11ExhZr6xx8G7b507VEYuzs-LNKNygCJL_iDyxV73WGgaPA2KHUjM-ESpBF-qkowZDMy6oLqbe24Kcv-Mznji_wLtmC2
8mXttjiMgVPlluEoFFTKw4YVnpnFEQCJ6ogYC4JJoRLSuPsLSC2JX_Bi40aiGus1BH3jtFQsxxmL3f7ZnPa-Xy66Zk_
ZyKZF1rqLKnxzCa85KKzZnCA83kpKT4ksM9MnqTBGBLMW70V4gTDL9zUvQm--VgaoN2lnHZ-oqHPzeNym7690YHzSA
A_C_g1un1gtc8RlkUDMu4F-DUSDGnJT8qjUQLiwXdyJS_cepCpluaA-6xCtx9DLkTcGy886SPX1u30LVeZfRRllunj
eEdId8Z884Ixq1T4V8Qo6AmdID0mLjbyFbpKoDLTfaTA98SuaEbko0oF5dzxo2xBgYxy-b1iE4K09r_PnBZ-BLVQ0zf0
_1Bjym9JYWo

```

Ahora puede copiar el token e iniciar sesión con él. También debe instalar Prometheus con Istio para registrar métricas que realicen un seguimiento del estado de Istio y de las aplicaciones dentro de la malla de servicios (consulte el [Capítulo 6](#) si necesita un repaso sobre cómo hacerlo). Use el siguiente manifiesto para instalar y configurar Prometheus:

```
$ kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.9/samples/ \
addons/prometheus.yaml
```

En la [Figura 7-4](#), se muestra el panel de Kiali resultante. Observará que la gateway de entrada de Istio envía tráfico a Kiali junto con otras implementaciones en el espacio de nombres `istiosystem`. El panel presenta una variedad de configuraciones y características de Istio que se pueden usar directamente desde la interfaz de usuario. Por ejemplo, si comprueba el espacio de nombres `default` en la pestaña Información general, puede ver que ya tiene habilitada la inserción automática del proxy sidecar para todas las implementaciones en este espacio de nombres (vea la [Figura 7-5](#)). Puede usar esta característica de la interfaz de usuario para habilitar o deshabilitar la inserción automática del proxy sidecar.



*Figura 7-4. Panel de Kiali que muestra un tráfico entrante simple*

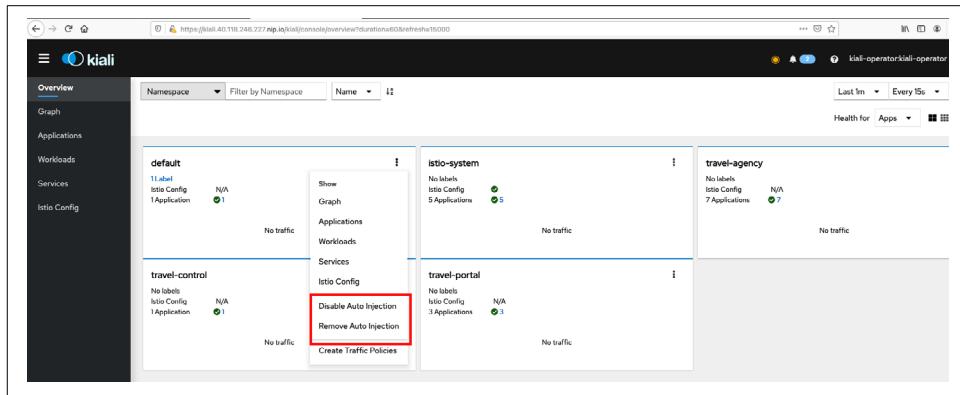


Figura 7-5. Panel de Kiali que muestra el estado del proxy sidecar de inserción automática en el espacio de nombres predeterminado

### Exploración del panel de Kiali

En esta sección, explicaremos cómo puede usar las características de Istio directamente desde el panel de Kiali. Para esta demostración, usaremos un ejemplo de un pequeño servicio de aplicaciones de viaje. Aunque no nos centraremos mucho en la aplicación, destacaremos las características de Kiali y la eficiencia con la que proporciona cada detalle sobre su aplicación nativa de la nube. También proporcionaremos una descripción general de alto nivel de la lógica de negocios de la aplicación de demostración. Si desea obtener más información, consulte la [documentación oficial de Kiali para la demostración de viajes](#).

Esta aplicación simula dos dominios empresariales organizados en diferentes espacios de nombres. El primer espacio de nombres se denomina `travel-portal` y tiene varias tiendas de viajes, donde los usuarios pueden buscar y reservar vuelos, hoteles, automóviles y seguros de viaje. Las aplicaciones de la tienda pueden comportarse de manera diferente en función de las características de la solicitud, como el canal (web o móvil) y el usuario (nuevo o existente). Todos los portales consumen un servicio llamado `travels` que se implementa en el espacio de nombres `travel-agency`.

El segundo espacio de nombres se denomina `travel-agency` y hospeda un conjunto de servicios creado a fin de proporcionar cotizaciones para viajes. Un servicio de viajes principal será el punto de entrada de la empresa para la agencia de viajes. Recibe una ciudad de destino y un usuario como parámetros, y calcula todos los elementos que componen un presupuesto de viaje: pasajes de avión, hospedaje, reserva de automóvil y seguro de viaje.

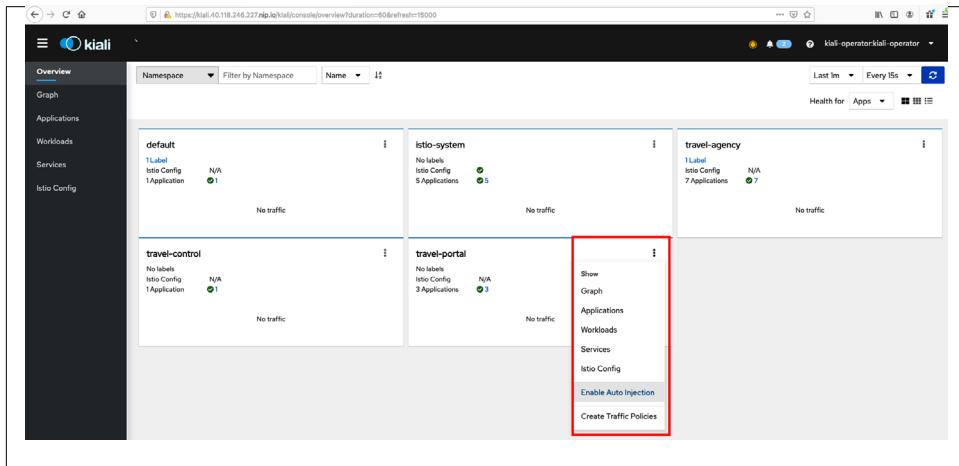
Cada servicio puede proporcionar una cotización independiente y, a continuación, el servicio de viajes debe incorporarlos en una sola respuesta.

También hay un tercer espacio de nombres, llamado `travel-control`, que ejecuta el panel principal de la empresa con diferentes funciones.

Hemos implementado estos tres espacios de nombres en los servicios de la siguiente manera:

```
$ kubectl create namespace travel-agency  
$ kubectl create namespace travel-portal  
$ kubectl create namespace travel-control  
  
$ kubectl apply -f <(curl -L https://raw.githubusercontent.com/kiali/demos/master/travels/ travel_agency.yaml) -n travel-agency  
$ kubectl apply -f <(curl -L https://raw.githubusercontent.com/kiali/demos/master/travels/ travel_portal.yaml) -n travel-portal  
$ kubectl apply -f <(curl -L https://raw.githubusercontent.com/kiali/demos/master/travels/ travel_control.yaml) -n travel-control
```

Lo primero que debe hacer después de implementar la aplicación de demostración de viajes es habilitar el soporte del proxy sidecar. Empiece por habilitar la inserción automática para todos los espacios de nombres recién creados haciendo clic en los tres puntos en el cuadro resaltado en la [Figura 7-6](#).



*Figura 7-6. Habilitación de la inserción automática en el espacio de nombres `travel-portal`*

También puede habilitar la inserción automática en una carga de trabajo que ya se ha implementado (es decir, pods) haciendo clic en el botón Actions (Acciones) en la esquina superior derecha (vea la [Figura 7-7](#)).

The screenshot shows the Kiali interface for managing workloads. In the top navigation bar, 'Workloads > Namespace: travel-agency > cars-v1'. Below this, there are tabs for Overview, Traffic, Logs, Inbound Metrics, Outbound Metrics, Traces, and Envoy Metrics. The 'Overview' tab is selected.

**Workload Properties**

- Name: cars-v1 (Missing Sidecar)
- Template Labels: app: cars, version: v1
- Istio Sidecar Inject Annotation: false
- Type: Deployment
- Created at: Apr 5, 10:00 PM
- Resource Version: 249184

**Graph Overview**

A network diagram showing the flow of traffic between various components: travel-v1, cars, cars-v1, mysql, and mysql-v1. The 'cars-v1' node is highlighted.

**Health Overview**

- Overall Health: Healthy
- Pod Status: cars-v1:1/1

**Actions**

- Enable Auto Injection (highlighted with a red box)
- Remove Auto Injection
- Show Envoy Details

**Pods (2) Services (0) Istio Config (0)**

Status	Name	Created at	Created by	Labels	Istio Init Containers	Istio Containers	Phase
Green	cars-v1-54740899bd-wmhzh	Apr 6, 1:16 PM	cars-v1-54740899bd (ReplicaSet)	app: cars, istio.io/rev: default More labels...	docker.io/istio/proxyv2:1.2	docker.io/istio/proxyv2:1.2	Running

Figura 7-7. Habilitación de la inserción automática en la carga de trabajo de los automóviles

Cuando habilite la inyección automática para las cargas de trabajo a través del panel de Kiali, volverá a implementar el pod de aplicaciones con un sidecar automáticamente insertado, lo que es similar al método que analizamos antes para el pod de Nginx.

También puede utilizar la característica de enruteamiento de solicitudes para un servicio en la pestaña Services (Servicios), como se muestra en la Figura 7-8, a fin de generar una regla de tráfico que crearía una configuración de Istio para enrutar una cantidad deseada de tráfico a un determinado host de gateway o utilizar interruptores.

The screenshot shows the Kiali interface for managing services. In the top navigation bar, 'Services > Namespace: travel-control > control'. Below this, there are tabs for Overview, Traffic, Inbound Metrics, and Traces. The 'Overview' tab is selected.

**Service Info**

- Name: control
- Labels: app: control
- Resource Version: 70442
- Selectors: app: control
- Created at: Apr 5, 10:00 PM

**Graph Overview**

An empty graph message: 'Empty Graph. Not enough data to generate a graph.'

**Health Overview**

- Overall Health: No health information
- Traffic Status (Last 10m): Inbound: No requests

**Actions**

- Create
- Request Routing
- Fault Injection
- Traffic Shifting
- TCP Traffic Shifting
- Request Timeouts
- Delete Traffic Routing

**Workloads (1) Istio Config (0)**

Name	Type	Labels	Created at	Resource version
control	Deployment	app: control, version: v1	Apr 5, 10:00 PM	220833

Figura 7-8. Acciones de la pestaña Servicios

Puede usar la característica de enrutamiento de solicitudes en el servicio de control para generar las reglas de tráfico, además de hacer clic en el botón Add Rule (Agregar regla), que agregará una regla predeterminada donde cualquier solicitud se enrutaría a la carga de trabajo del control. Del mismo modo, puede agregar una gateway con un host al equilibrador de carga y actualizarlo haciendo clic en el botón Update (Actualizar), como se muestra en la [Figura 7-9](#).

Rule order	Request Matching	Route To	
1	Any request	<b>WS</b> control (100% routed traffic)	⋮

**Gateway Hosts:** 40.118.246.227  
One or more hosts exposed by this gateway. Enter one or multiple hosts separated by comma.

*Figura 7-9. Incorporación de una ruta de solicitud para un servicio en una configuración de Istio a través de Kiali*

Kiali también proporciona registros listos para usar, métricas a través de Prometheus y seguimientos a través de Jaeger si los implementó en su entorno. Recomendamos encarecidamente explorar la interfaz de usuario de Kiali para ver todas sus capacidades.

## Resumen

En este capítulo, exploramos la detección de servicios y las mallas de servicios, dos conceptos importantes que lo ayudan a unir eficazmente sus aplicaciones en un entorno de nube como Azure. Presentamos CoreDNS como el DNS predeterminado que reemplaza a kube-dns como un mecanismo de detección de servicios en el entorno de Kubernetes. También presentamos la malla de servicios de Istio, que utiliza Envoy como su plano de datos para obtener información útil sobre los servicios y proporciona el conjunto completo de características de la malla de servicios de Istio. Concluimos el capítulo con una sección sobre Kiali, que proporciona una consola de administración completa en una malla de servicios basada en Istio. Kiali proporciona una vista única de todo en la pila para ofrecer observabilidad en un servicio nativo de la nube.

En el siguiente capítulo, hablaremos en detalle sobre cómo funciona la red de contenedores en Kubernetes y cómo puede proteger estrictamente su infraestructura con diversas técnicas de administración de políticas.



## CAPÍTULO 8

# Redes y administración de políticas: He aquí los guardianes

En los capítulos anteriores, creamos infraestructura en Azure y explicamos cómo descubrir y supervisar nuestras aplicaciones. Ahora es el momento de proteger esas aplicaciones. Aunque, con el tiempo, han existido numerosas filtraciones de datos en la nube debido a errores de configuración sencillos, la realidad es que proteger la infraestructura en la nube no es difícil. La tecnología de redes en la nube ha evolucionado rápidamente, y hoy en día una serie de proveedores proporciona software nativo en la nube que puede ayudarlo a mejorar su configuración de red, además de protegerla.

Azure viene con una oferta, Azure Policy, que le permite establecer una política para un inquilino, un grupo de administración o una suscripción, lo que proporciona una capa de seguridad de forma predeterminada. Por ejemplo, en Azure puede establecer una política que garantizará que no se pueda acceder públicamente a ninguna cuenta de almacenamiento dentro de un grupo de administración o una suscripción.

En este capítulo, exploraremos el poder de las redes de contenedores y las diversas formas en que puede usarlas para mejorar su infraestructura. También explicaremos cómo aplicar una política a su infraestructura para mantenerla segura.

Comenzaremos con un análisis sobre la red de contenedores y los estándares en los que se basan varios proyectos. Luego, nos centraremos en productos, como Calico y Flannel, que proporcionan conectividad de red y cumplimiento de políticas de redes. Concluiremos el capítulo con un análisis sobre la aplicación de políticas del sistema con Open Policy Agent (OPA).

# La interfaz de red de contenedores (CNI)

Como mencionamos en capítulos anteriores, los contenedores ofrecen una amplia gama de características de seguridad y portabilidad. Uno de los mayores retornos proviene de la pila de redes. Los cgroups de Linux y las características de espacio de nombres de red abren una variedad de funcionalidades de red que se puede utilizar para mejorar la seguridad y la telemetría, además de administrar el rendimiento. Un buen ejemplo de la flexibilidad y el poder de la funcionalidad de la red de contenedores es Cilium, una herramienta de software que proporciona equilibrio de carga, seguridad, telemetría, administración del ancho de banda y otras capacidades como complemento para Kubernetes. Analizaremos Cilium en detalle más adelante en este capítulo.

En el [Capítulo 3](#), mencionamos brevemente cómo la industria tiene una especificación estándar para el tiempo de ejecución del contenedor y la imagen del contenedor. Container Network Interface (CNI) se creó en 2015 (y finalmente se agregó a la lista de proyectos de incubación de la Cloud Native Computing Foundation [CNCF]) como un estándar para configurar las interfaces de red de contenedores y la configuración de red asociada. La [especificación CNI](#) permite la administración de los siguientes primitivos:

- Nombres de interfaz y enlaces de espacio de nombres de red
- Direcciones IP
- Enrutamiento
- Configuración DNS
- Configuración Sysctl

Un ejemplo de configuración de CNI se ve así:

```
{
  "cniVersion": "1.0.0",
  "name": "dbnet",
  "plugins": [
    {
      "type": "bridge",
      "bridge": "cni0",
      "args": {
        "labels" : {
          "appVersion" : "1.0"
        }
      },
      "ipam": {
        "type": "host-local",
        "subnet": "10.1.0.0/16",
        "gateway": "10.1.0.1"
      },
      "dns": {
        "nameservers": [ "10.1.0.1" ]
      }
    },
    {
      "type": "tuning",
      "latency": 100,
      "loss": 0
    }
  ]
}
```

```
        "sysctl": {  
            "net.core.somaxconn": "500"  
        }  
    }  
}
```

## ¿Por qué usar una CNI?

Es posible que se pregunte por qué la industria necesita una especificación solo para las redes. Como mencionamos en el [Capítulo 3](#), el ecosistema de contenedores tiene un gran número de proyectos de red que se conectan a sistemas de contenedores, como Docker y Kubernetes. Varios grupos crean complementos relacionados con la red disponibles públicamente, por lo que es necesario contar con una forma estándar de describir y controlar la capa de red del contenedor.

La plataforma CNI permite una verdadera virtualización de infraestructura, lo que significa que no necesita preocuparse por la administración de la infraestructura de red, incluso en la nube. La utilización de la plataforma de contenedores como infraestructura de red reduce en gran medida el costo de implementar y operar una infraestructura de red compleja y proporciona la capacidad de escalar y actualizar sin tener que invertir en dispositivos de red virtual físicos o dedicados.

## ¿Cómo funciona la CNI con Azure?

Azure es compatible con CNI, pero tiene dos complementos especiales que conectan sus contenedores a Azure Virtual Network (VNet):

- `azure-vnet`, que implementa el complemento de red de CNI
- `azure-vnet-ipam`, que implementa el complemento de Administración de direcciones IP (IPAM) de CNI

Estos complementos aseguran que el contenedor pueda interactuar correctamente con el plano de control de VNet de Azure. Puede utilizar estos complementos tanto en Azure Kubernetes Service (AKS) como en máquinas virtuales de Azure independientes.



### Redes de Windows

Aunqelas pilas de red de Linux y Windows son fundamentalmente muy diferentes, los complementos de Azure CNI tienen soporte de primera clase para Windows, lo que significa que puede ejecutar contenedores de Windows y Linux en la misma infraestructura sin tener que crear una infraestructura de red aparte.

Una configuración de Azure CNI se ve así:

```
{  
    "cniVersion": "1.0.0",  
    "name": "azure",  
    "plugins": [  
        {  
            "type": "azure-vnet",  
            "mode": "bridge",  
            "bridge": "azure0",  
            "ipam": {  
                "type": "azure-vnet-ipam"  
            }  
        },  
        {  
            "type": "azure-vnet-ipam",  
            "Environment": "azure"  
        },  
        {  
            "dns": {  
                "nameservers": [ "10.1.0.1" ]  
            }  
        },  
        {  
            "type": "tuning",  
            "sysctl": {  
                "net.core.somaxconn": "500"  
            }  
        }  
    ]  
}
```

El uso de los complementos `azure-vnet` y `azure-vnet-ipam` traslada la configuración y administración de IP lejos del host y al plano de control de Azure VNet.

## Diversos proyectos de CNI

El proyecto de CNI contiene un conjunto de complementos de CNI predeterminados divididos en tres categorías:

### *Principal*

Estos complementos crean interfaces (bridge, ipvlan, macvlan, ptp, host-device).

### *IPAM*

Estos complementos asignan direcciones IP a las interfaces (dhcp, host-local, static).

### *Meta*

Estos son otros complementos nativos que permiten acciones como:

- Ajuste de Sysctl
- Límite de ancho de banda
- Firewalls
- Flannel (que se tratará más adelante en este capítulo)

Puede obtener más información sobre estos complementos nativos en la [página de complementos de CNI](#).

Además de los complementos de CNI que mencionamos aquí, hay un vasto ecosistema de proyectos de terceros que escriben complementos compatibles con CNI. En el momento de redactar este documento, más de 25 proyectos externos han aprovechado la norma CNI. Estos proyectos proporcionan diferentes tipos de funcionalidad, incluido el equilibrio de carga, IPAM, redes superpuestas, seguridad y telemetría de red.

Algunos de los proyectos más pequeños incluyen los siguientes:

- Infoblox
- Juniper Contrail
- VMware NSX

Si bien algunos complementos de CNI están diseñados para una infraestructura muy específica (por ejemplo, ACI CNI de Cisco), muchos funcionan perfectamente bien con Azure y los servicios de Azure. En las siguientes secciones, exploraremos Calico, Cilium, Flannel y OPA, y analizaremos la forma en que estos sistemas ayudan a crear y proteger la infraestructura de red.

## Calico

Calico es una solución de seguridad y políticas de red de open source para contenedores, máquinas virtuales e instalaciones sin sistema operativo en Windows y Linux. Proporciona la aplicación de políticas de seguridad de red, así como la capacidad de implementar redes de confianza cero. Calico admite varios planos de datos, incluido un plano de datos de Berkeley Packet Filter (eBPF) extendido de Linux de última generación, un plano de datos de red estándar de Linux y un plano de datos de Windows HNS. Calico se ejecuta sobre el plano de datos del host para realizar la aplicación de las políticas de red. Es compatible con la pila de red estándar de Linux.

### ¿Por qué usar Calico?

Administrar la conectividad y la política de red se vuelve potencialmente más difícil a medida que crece la infraestructura. Además, es excepcionalmente difícil garantizar que la política prevista sea correcta, que se aplique de forma correcta y que se apliquen los comentarios sobre ella. Aquí es donde se destaca Calico. Le permite definir su política de red de forma abstracta y escalable y, a continuación, garantiza que la política se aplique sin sacrificar el rendimiento de la red.

## Arquitectura básica

Calico tiene una arquitectura de cliente/almacén de datos que simplifica el funcionamiento de la plataforma. Incluye los siguientes componentes clave (que se representan también en la [Figura 8-1](#)):

### *Felix*

El demonio de programación de red que instala listas de control de acceso (ACL) a la red, información de enrutamiento y administración de interfaces e informes de estado

### *BIRD*

Obtiene la información de enrutamiento de Felix y redistribuye las rutas sobre Protocolo de gateway de borde (BGP)

### *confd*

Escucha los cambios de enrutamiento desde el almacén de datos de Calico y los pasa a BIRD

### *Complemento de CNI de Calico*

La interfaz para las interfaces de red del contenedor

### *Almacén de datos*

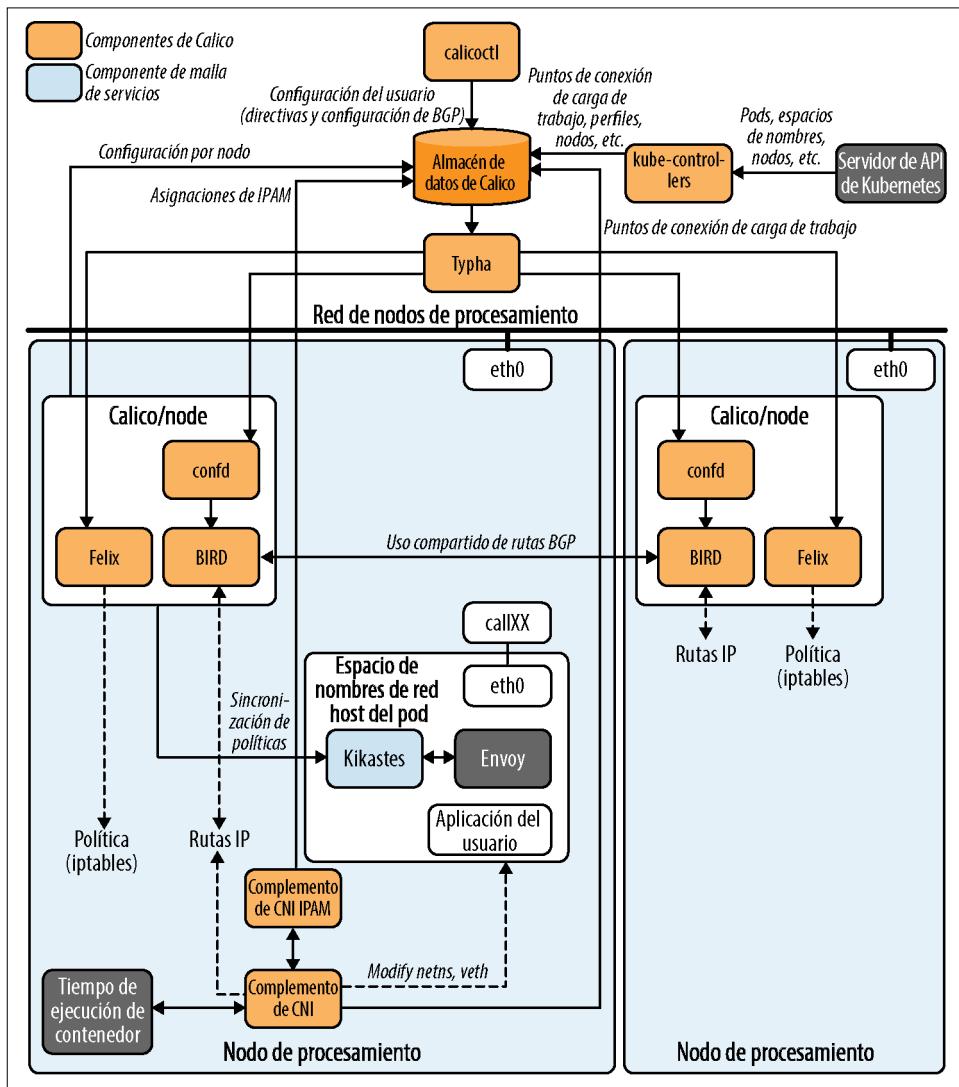
Contiene información operativa sobre las políticas, las cargas de trabajo y las asignaciones de IPAM

### *Typha*

Un proxy de almacenamiento en caché para el almacén de datos que se utiliza para escalar el número de nodos que se conectan al almacén de datos

### *calicectl*

CLI para crear, leer, actualizar y eliminar objetos de Calico



*Figura 8-1. Arquitectura de Calico*

## Implementación de Calico

Hay dos opciones de implementación de Calico: instálelo como parte del motor de políticas de red en una instalación de AKS o impleméntelo manualmente en Kubernetes.

## Implementación de Calico a través de la instalación de AKS

Azure introdujo soporte de primera clase para Calico como un motor de *políticas* de red. Se puede instalar como parte de la creación de una instancia de AKS en el paso de configuración de red, como se muestra en la [Figura 8-2](#).



Figura 8-2. Configuración de la política de red para AKS

En este paso, si incluye Calico como parte de la instalación de AKS, se instalará un pod Typha de Calico en cada nodo de proceso de Kubernetes.

## Instalación manual de Calico

Si desea instalar manualmente Calico en su clúster de Kubernetes, puede usar kubectl para descargar el operador e instalar los componentes:

1. En primer lugar, instale el operador y, luego, los recursos de Calico. Esto creará un espacio de nombres `tigera-operator`:

```
$ kubectl create -f https://docs.projectcalico.org/manifests/tigera-operator.yaml  
$ kubectl create -f https://docs.projectcalico.org/manifests/custom-resources.yaml
```

2. Valide que todos los nodos de Calico se estén ejecutando ahora:

```
$ kubectl get pods -n calico-system
```

3. Espere hasta que todos los pods se encuentren en el estado en ejecución.



La configuración `ipPools` en el archivo `custom-resources.yaml` no se puede modificar después de la implementación. Esto significa que `ipPools` (las direcciones IP que se asignan a cada pod) se configuran después de que aplica el archivo `custom-resources.yaml`.

## Instalación de calicctl

Para controlar Calico, deberá tener acceso a su utilidad de CLI. Hay algunas formas de acceder a la utilidad, como se describe en la [documentación de instalación de Calico](#). En este ejemplo, simplemente vamos a descargar la CLI en nuestra máquina y ejecutarla como un complemento kubectl:

1. Inicie sesión en un host y vaya hasta `/usr/local/bin`:

```
$ cd /usr/local/bin
```

2. Descargue el archivo binario:

```
$ curl -o kubectl-calico -O -L "https://github.com/projectcalico/calicoctl/releases/ \
download/v3.21.0/calicoctl-linux-ppc64le"
```

3. Configure el archivo binario como ejecutable:

```
$ chmod +x kubectl-calico
```

4. Verifique que el complemento funcione:

```
$ kubectl calico -h
```

## Análisis detallado de Calico

Analizamos a grandes rasgos los beneficios de usar Calico y algunos conceptos básicos sobre cómo funciona y cómo instalarlo. Ahora examinaremos cómo usarlo. En lo que queda de esta sección, utilizaremos eBPF como el plano de datos en lugar de la pila de red estándar de Linux.

Azure no permite IP desconocidas en su plano de datos, por lo que solo puede usar una red de superposición basada en VXLAN entre nodos. Esto solo es posible si está compilando un clúster autoadministrado ([Figura 8-3](#)) y no AKS. En AKS debe usar Azure como el proveedor de red, como se muestra en la [Figura 8-4](#).

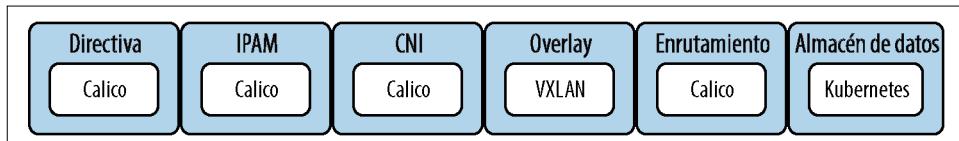


Figura 8-3. Configuración autoadministrada (con VXLAN)

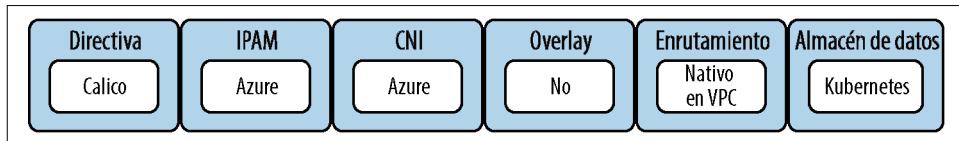


Figura 8-4. Configuración de red de AKS

## Habilitación de eBPF

El siguiente paso a fin de configurar nuestra infraestructura de Calico es habilitar el plano de datos de eBPF. Habilitar eBPF le permite aprovechar las nuevas características del kernel y eliminar el uso de kube-proxy para el equilibrio de carga. Cuando se habilite eBPF, se deshabilitará el uso de kube-proxy. Puede obtener más información sobre la implementación de eBPF de Calico en [esta entrada de blog](#).

Para habilitar eBPF, siga estos pasos:

1. Para obtener el servidor de API, ejecute:

```
$ kubectl get configmap -n kube-system kube-proxy -o jsonpath='{.data.kubeconfig}' \|  
grep server'  
server: https://d881b853ae312e00302a84f1e346a77.hcp.us-east-1.azmk8s.io
```

2. Actualice el espacio de nombres **tigera-operator**:

```
kind: ConfigMap  
apiVersion: v1  
metadata:  
  name: kubernetes-services-endpoint  
  namespace: tigera-operator  
data:  
  KUBERNETES_SERVICE_HOST: "<API server host>"  
  KUBERNETES_SERVICE_PORT: "<API server port>"
```

3. Reinicie el operador para usar el cambio:

```
$ kubectl delete pod -n tigera-operator -l k8s-app=tigera-operator
```

4. Deshabilite la implementación de kube-proxy (para ahorrar recursos):

```
$ kubectl patch networks.operator.openshift.io cluster --type merge -p \  
'{"spec":{"deployKubeProxy": false}}'
```

5. Con calicoctl, habilite eBPF:

```
$ calicoctl patch felixconfiguration default --patch='{"spec": {"bpfEnabled": true}}'
```

Si tiene problemas para ejecutar eBPF, Calico proporciona una [guía de solución de problemas](#).



Si usa eBPF y tiene una versión de kernel posterior a la 4.16, Calico intentará automáticamente usar XDP para procesar paquetes, lo que proporcionará mejoras de rendimiento. Esto es útil para servicios de alto rendimiento o servicios a los que se aplica DOS.

## Implementación de la política de seguridad de Calico

Ahora que tenemos configurada nuestra red, estamos listos para aplicar algunas políticas de seguridad. La política de red de Calico tiene un conjunto de capacidades más grande que Kubernetes, que incluye:

- Ordenación de políticas
- Reglas de denegación
- Mayor flexibilidad en las reglas de coincidencia

La política de red de Calico admite la protección de aplicaciones con criterios de Capas OSI de la 5 a la 7, además de identidad criptográfica. Encontrará ejemplos en la [página de configuración de la política de red de Calico](#).

La política de red se puede aplicar de varias maneras:

- Globalmente (se aplica a todos los pods en todos los espacios de nombres), lo que se conoce como GlobalNetworkPolicy
- Por red o host, lo que se conoce como HostEndpoint
- Por espacio de nombres, lo que se conoce como NetworkPolicy

### **GlobalNetworkPolicy: permitir el tráfico ICMP**

La primera política que crearemos es una política de red global que permite el tráfico ICMP para ping y traceroute desde todos los hosts dentro del clúster. Cree el archivo *global-policy.yaml* como se indica a continuación:

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: allow-ping-in-cluster
spec:
  selector: all()
  types:
  - Ingress
  ingress:
  - action: Allow
    protocol: ICMP
    source:
      selector: all()
    icmp:
      type: 8 # Ping request
  - action: Allow
    protocol: ICMPv6
    source:
      selector: all()
    icmp:
      type: 128 # Ping request
```

Ejecute el siguiente código para aplicar la política:

```
$ calicoctl create global-policy.yaml
```

### **NetworkPolicy: permitir el tráfico dentro de un espacio de nombres entre dos etiquetas**

En este ejemplo, permitiremos el tráfico desde el espacio de nombres de producción con el selector verde en un pod en el mismo espacio de nombres con el selector azul en el puerto TCP/1234. Cree un archivo nuevo llamado *network-policy.yaml*:

```
apiVersion: projectcalico.org/v3
kind: NetworkPolicy
metadata:
  name: allow-tcp-1234
  namespace: production
spec:
  selector: color == 'blue'
  ingress:
  - action: Allow
    protocol: TCP
```

```
source:  
  selector: color == 'green'  
destination:  
  ports:  
    - 1234
```

Ejecute el siguiente código para aplicar la política:

```
$ calicoctl create network-policy.yaml
```

Una vez aplicada esta política, el tráfico podrá fluir al puerto TCP 1234 a cualquier pod verde.

## Cilium

El proyecto Cilium es una de las herramientas de red más innovadoras introducidas en los últimos cinco años. Utiliza ampliamente eBPF para proporcionar un conjunto de características de red, observabilidad y seguridad orientadas a la infraestructura nativa de la nube.

Cilium utiliza la tecnología de kernel de Linux eBPF que permite baja sobrecarga, visibilidad de alta potencia y lógica de control para el kernel y las aplicaciones que se ejecutan en ella. El uso de eBPF hace que Cilium sea una opción popular para los arquitectos de infraestructura, puesto que la plataforma proporciona características en torno a tres pilares clave:

### Redes

Cilium implementa su propia CNI que utiliza eBPF de Linux. Esto hace que Cilium sea altamente eficaz y le permite realizar un equilibrio de carga, en lugar de depender de kube-proxy. También permite la conectividad de varios clústeres, lo que es importante para instalaciones más grandes.

### Observabilidad

El uso de eBPF de Cilium hace que sea una pieza única de software de observabilidad. Cilium puede realizar una inspección de paquetes profunda en todas las capas de conectividad y datos de red.

### Seguridad

Una de las mejores características de Cilium es que puede realizar un cifrado transparente entre hosts a través de IPSec. Esto significa que obtiene un cifrado de extremo a extremo a través de un mecanismo altamente eficiente sin tener que realizar cambios en su aplicación. Si bien Cilium proporciona su propia política de seguridad que coincide con etiquetas+CIDR, también realiza filtrado de capa 7 (OSI) (por ejemplo, filtra las consultas de DNS salientes o filtra las solicitudes HTTP entrantes).

Por último, Cilium le permite realizar una introspección de Capa 7 para tomar decisiones de políticas (consulte [esta entrada de blog](#) para obtener más información).

El uso de eBPF de Cilium como núcleo permite al software realizar introspección y manipular profundamente (cuando sea necesario) todo el tráfico de red sin afectar el rendimiento del sistema.

El agente Cilium (cilium-agent) se ejecuta en cada nodo (consulte la [Figura 8-5](#)) y acepta la configuración a través de la API de Kubernetes. El agente convertirá esa configuración en un programa de eBPF que se ejecuta en la interfaz de red del contenedor, en la red del host o en una de las tarjetas de red del host.

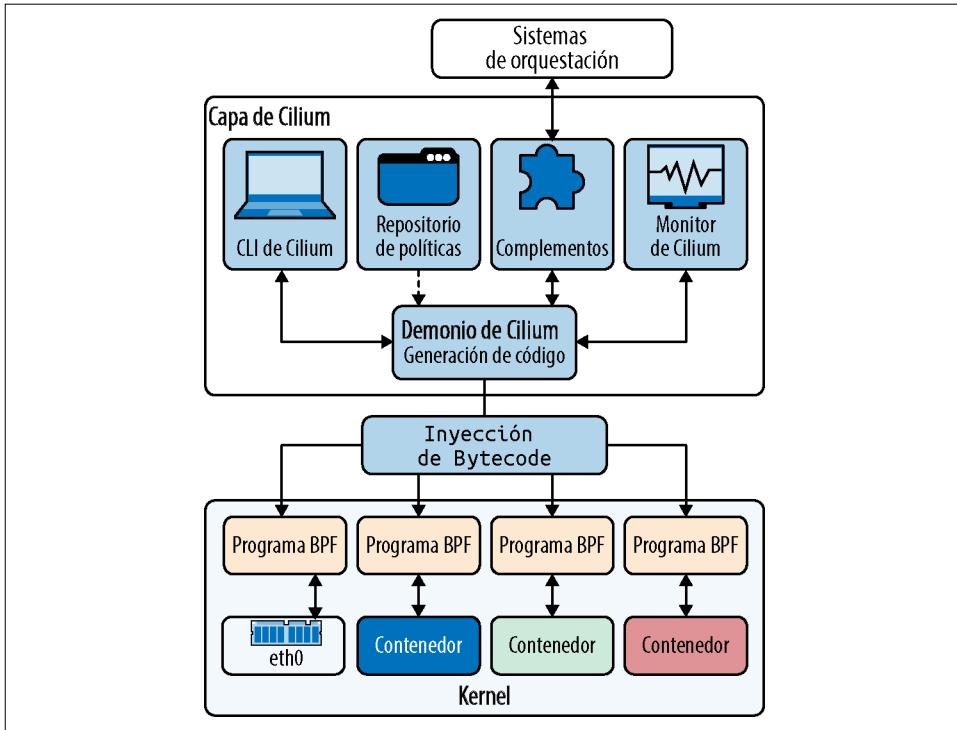


Figura 8-5. La arquitectura de Cilium

## Implementación de Cilium

Cilium se puede instalar en un clúster de Kubernetes autoadministrado o en un clúster de AKS administrado. En ambos casos, se puede implementar a través de Helm, lo que facilita la instalación. Además, Cilium proporciona un contenedor para realizar comprobaciones de conectividad a fin de verificar que el tráfico de red pueda fluir libremente entre pods en su clúster.

Si desea obtener más información, consulte la [documentación especializada para implementar Cilium en Azure](#).

### Instalación de Cilium autoadministrada

Instalaremos Cilium con su archivo de instalación rápida:

```
$ kubectl apply -f https://raw.githubusercontent.com/cilium/cilium/v1.9/install/ \
  kubernetes/quick-install.yaml
```

El siguiente código instalará la comprobación de conectividad entre hosts:

```
$ kubectl apply -f https://raw.githubusercontent.com/cilium/cilium/v1.9/examples/ \
  kubernetes/connectivity-check/connectivity-check.yaml
```

Esto implementará una serie de pods que utilizarán varias rutas de red para conectarse entre sí con el fin de verificar la conectividad. Las rutas de conectividad incluyen con y sin equilibrio de carga de servicio y varias combinaciones de políticas de red.

## Instalación de Cilium en AKS

Para configurar Cilium en AKS, utilizaremos un gráfico de Helm:

1. Primero, crearemos un nuevo clúster de AKS:

```
$ export RESOURCE_GROUP_NAME=aks-test
$ export CLUSTER_NAME=aks-test
$ export LOCATION=westus

$ az group create --name $RESOURCE_GROUP_NAME --location $LOCATION
$ az aks create \
  --resource-group $RESOURCE_GROUP_NAME \
  --name $CLUSTER_NAME \
  --location $LOCATION \
  --node-count 2 \
  --network-plugin azure
```

2. Ahora crearemos una entidad de servicio para interactuar con las API de Azure y llenar las variables de entorno para pasárlas a Helm:

```
$ az ad sp create-for-rbac --name cilium-operator > azure-sp.json

$ AZURE_SUBSCRIPTION_ID=$(az account show | jq -r .id)
$ AZURE_CLIENT_ID=$(jq -r .appId < azure-sp.json)
$ AZURE_CLIENT_SECRET=$(jq -r .password < azure-sp.json)
$ AZURE_TENANT_ID=$(jq -r .tenant < azure-sp.json)
$ AZURE_NODE_RESOURCE_GROUP=$(az aks show --resource-group $RESOURCE_GROUP_NAME \
  --name $CLUSTER_NAME | jq -r .nodeResourceGroup)
```

3. Instalaremos el repositorio de Cilium localmente y, luego, instalaremos Cilium en nuestro clúster de Kubernetes:

```
$ helm repo add cilium https://helm.cilium.io/
$ helm install cilium cilium/cilium --version 1.9.9 \
  --namespace kube-system \
  --set azure.enabled=true \
  --set azure.resourceGroup=$AZURE_NODE_RESOURCE_GROUP \
  --set azure.subscriptionID=$AZURE_SUBSCRIPTION_ID \
  --set azure.tenantID=$AZURE_TENANT_ID \
  --set azure.clientID=$AZURE_CLIENT_ID \
  --set azure.clientSecret=$AZURE_CLIENT_SECRET \
  --set tunneldisabled \
  --set ipam.mode=azure \
  --set masquerade=false \
  --set nodeinit.enabled=true
```

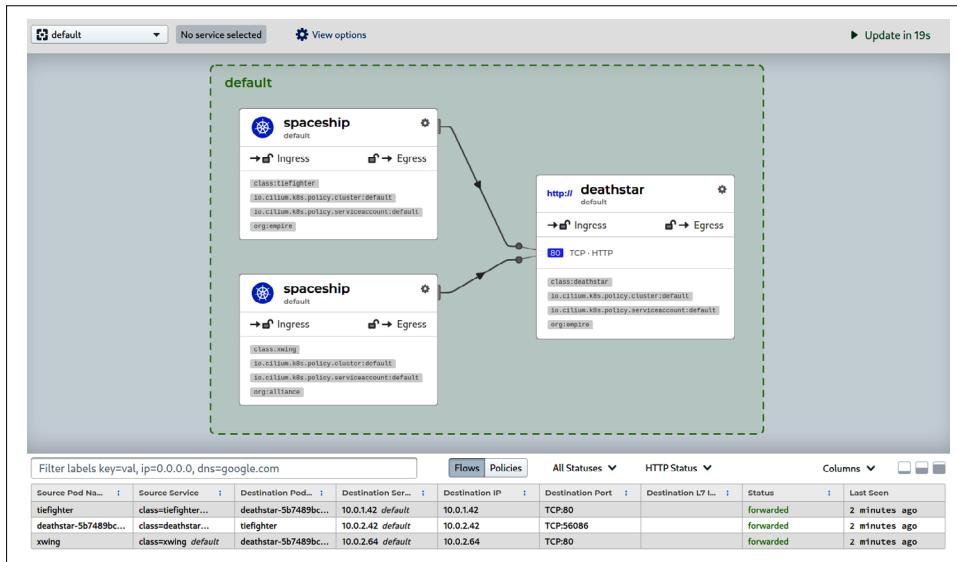
```
--set kubeProxyReplacement=strict \
--set hostFirewall=true \
--set loadBalancer.algorithm=maglev \
--set k8sServiceHost=REPLACE_WITH_API_SERVER_IP \
--set k8sServicePort=REPLACE_WITH_API_SERVER_PORT
```

4. Podemos verificar la instalación mediante la comprobación de que se crearon los cuatro pods iniciales y que se están ejecutando tanto un pod `cilium` como uno `cilium-operator`:

```
$ kubectl -n kube-system get pods --watch
cilium-operator-ad4375ds5-2x2q3           1/1      Running   0          4m18s
cilium-s5x8xk                            1/1      Running   0          4m19s
```

## Instalación de Hubble

También le recomendamos instalar **Hubble**, una interfaz de usuario que le permite ver un mapa de servicio que representa las dependencias entre los servicios y las estadísticas sobre el uso y el rendimiento de la red. La interfaz similar a un gráfico y que se actualiza en vivo de Hubble (consulte la [Figura 8-6](#)) le permite presentar mucha información sobre el diseño y el flujo de datos entre contenedores. Además, proporciona una interfaz para modelar la política de Cilium.



*Figura 8-6. Captura de pantalla de la interfaz de usuario de Hubble que muestra una arquitectura sencilla*

Puede instalar Hubble con kubectl:

```
kubectl apply -f https://raw.githubusercontent.com/cilium/cilium/v1.9/install/kubernetes/ \
quick-hubble-install.yaml
```

# Integración de Cilium con la nube

Como ya mencionamos, Cilium tiene un conjunto de características enriquecidas que puede aprovechar. En esta sección, utilizaremos Cilium como un proveedor de red y un ejecutor de políticas (firewall). Luego analizaremos la observabilidad con Cilium.

## Firewall del host

Al igual que cuando profundizamos en Calico, aquí implementaremos un conjunto de políticas de red. La ventaja de Cilium es que puede aplicar la política en las capas L3, L4 o L7. La aplicación de políticas de red tiene tres modos, como se muestra en la [Tabla 8-1](#).

Tabla 8-1. Modos de aplicación de políticas de red

Modo	Descripción
Predeterminado	Si una regla selecciona puntos de conexión y la regla tiene una sección de entrada o salida, el punto de conexión lo denegará según la política.
Siempre	La aplicación de políticas está habilitada en todos los puntos de conexión, incluso si no hay reglas que seleccionen puntos de conexión específicos.
Nunca	La aplicación de políticas está deshabilitada en todos los puntos de conexión y se permite todo el tráfico.

A fin de configurar el modo de aplicación de políticas en tiempo de ejecución para todos los puntos de conexión administrados por un agente Cilium, use el siguiente comando y elija un modo predeterminado, siempre o nunca:

```
$ cilium config PolicyEnforcement={default,always,never}
```

La política se escribe en un modelo de lista de permitidos en el que el tráfico se bloqueará si no hay ninguna regla de permiso explícita.

Las reglas L3 y L4 se pueden especificar con los siguientes métodos:

### Etiquetas

Use un selector para agrupar pods.

### Servicios

Defina una política con un servicio de Kubernetes.

### Entidades

Ámbitos predefinidos especiales administrados por Cilium.

### IP/CIDR

Una IP o intervalo IP (CIDR).

### Nombres DNS

Un nombre de DNS.

Por ejemplo, si desea permitir el tráfico entre dos pods etiquetados, puede crear la siguiente política:

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: "l3-rule"
spec:
  endpointSelector:
    matchLabels:
      role: backend
  ingress:
    - fromEndpoints:
      - matchLabels:
          role: frontend
```

Puede aplicar esta política mediante la ejecución del siguiente código:

```
$ cilium policy import l3-rule-example-1.yaml
```

Además, si desea crear un ejemplo de L4 en el que quiera limitar el DNS de salida saliente a los servidores DNS públicos de Google, puede escribir la regla que se muestra en el siguiente ejemplo:

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "allow-to-google-dns"
spec:
  endpointSelector:
    {}
  egress:
    - toCIDR:
      - 8.8.8.8/32
      - 8.8.8.4/32
      toPorts:
        - ports:
          - port: '53'
            protocol: UDP
        - ports:
          - port: '53'
            Protocol: TCP
```

Para aplicar la regla, ejecute el siguiente código:

```
$ cilium policy import l4-rule-example-1.yaml
```

Por último, Cilium también permite la visibilidad L7 para tráfico HTTP(s), Kafka y DNS. En el siguiente ejemplo, permitimos el tráfico al punto de conexión `/admin` en el puerto 80 a todos los pods:

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "rule1"
spec:
  description: "Allow HTTP GET /admin
  endpointSelector:
    {}
  ingress:
```

```
- fromEndpoints:  
  toPorts:  
    - ports:  
      - port: "80"  
        protocol: TCP  
    rules:  
      http:  
        - method: "GET"  
        path: "/admin"
```

Puede ver más detalles específicos sobre las reglas de políticas de L7 en [la documentación de Cilium sobre la política de capa 7](#).



Cilium proporciona una herramienta interactiva para modelar políticas de red y visualizarlas. Puede probarla en la [página de Cilium Editor](#).

## Observabilidad

Como mencionamos brevemente antes, Hubble es donde los operadores de Cilium pueden observar lo que está sucediendo en su infraestructura. Una vez que instale Hubble y la prueba de conectividad (de los pasos de instalación), podrá observar los flujos de red dentro del espacio de nombres. En esta sección iremos un paso más allá y usaremos una nueva característica de Cilium que proporciona visibilidad del protocolo de capa 3 y capa 4, lo que le ofrecerá detalles más específicos sobre el tráfico en cada puerto. Tendrá que implementar políticas de capa 7 para obtener las características de observabilidad de la capa 7.

Para habilitar la visibilidad de la capa 3 y la capa 4, tendrá que habilitar las anotaciones en los pods. Este es un ejemplo:

```
$ kubectl annotate pods --all io.cilium.proxy-visibility="<Egress/53/UDP/DNS>, \  
<Ingress/80/TCP/HTTP>"
```

Esto habilitará las métricas en el tráfico de DNS saliente (de salida), así como el tráfico HTTP entrante (de entrada) al pod.

La información de visibilidad se representa mediante una lista de tuplas separada por comas en la anotación:

```
<{Traffic Direction}/{L4 Port}/{L4 Protocol}/{L7 Protocol}>
```

Cuando esté activado, Hubble mostrará información específica del protocolo. Consulte la [Figura 8-7](#) como un ejemplo de la visibilidad de DNS.



Figura 8-7. Ejemplo de observabilidad del tráfico de DNS

Si desea obtener información sobre cómo inspeccionar las conexiones cifradas de Seguridad de la capa de transporte (TLS), visite la [página Inspección de conexiones cifradas con TLS con Cilium](#).

## Flannel

**Flannel** es un sistema de tejido de red de capa 3 (OSI) para Kubernetes que originalmente formaba parte del proyecto CoreOS de Red Hat. Es uno de los CNI más sencillos para Kubernetes. Se basa en un agente binario, llamado flanneld, a fin de ejecutarse en cada host para configurar una red de pod de superposición y, a continuación, almacenar la configuración en el almacén de datos de Kubernetes. Flannel solo proporciona conectividad de red. No es compatible con ningún tipo de aplicación de políticas de red.

## Implementación de Flannel

Para instalar Flannel rápidamente cuando se configura el clúster, ejecute el siguiente código:

```
$ kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/ \
Documentation/kube-flannel.yml
```

La aplicación de `kube-flannel.yml` definirá lo siguiente:

- Un `ClusterRole` y un `ClusterRoleBinding` para el control de acceso basado en roles (RBAC)
- Una cuenta de servicio de Kubernetes para que la use Flannel
- Un `ConfigMap` que contiene una configuración CNI y una configuración de Flannel. La red en la configuración de Flannel debe coincidir con el esquema de Enrutamiento de interdominios sin clases (CIDR) de la red del pod. La elección del back-end también se realiza aquí, y el valor predeterminado es VXLAN.
- Un `DaemonSet` para cada arquitectura que implemente el pod de Flannel en cada nodo. El pod tiene dos contenedores: el demonio de Flannel y un `initContainer` para implementar la configuración de CNI en una ubicación que kubelet puede leer.

Cuando ejecuta pods, se le asignan direcciones IP desde el CIDR de la red de Pod. Independientemente del nodo en el que terminen esos pods, podrán comunicarse unos con otros.



Este manifiesto configurará la red de pods para que sea 10.244.0.0/16 y utilice el back-end de VXLAN de forma predeterminada. Se otorgará una máscara 10.244.X.0/24 a cada nodo del pod.

Puede volver a configurar el back-end para que use un espacio de direcciones diferente, una máscara más pequeña o diferentes configuraciones de back-end mediante la modificación de los datos que se escriben en `net-conf.json` en el archivo `fkube-flannel.yaml`. Puede obtener más información sobre la configuración de back-end en las páginas de la [documentación de back-end de Flannel](#) y la [configuración de Flannel](#).

Ahora los nodos de trabajador de Kubernetes tendrán direcciones IP de su Azure VNet, pero cualquier pod implementado tendrá direcciones IP del intervalo IP de red de pod configurado. Si desea tener varias redes de pods (para la separación de tráfico de red), tendrá que ejecutar un proceso `flanneld` separado.

## Análisis detallado de Flannel

Flannel es una CNI de Kubernetes más simple y requiere una configuración mínima. Tiene back-ends conectables que crean redes superpuestas entre las máquinas en una red (consulte la [Figura 8-8](#)). Entre los back-ends de transporte compatibles, se incluyen los siguientes:

- VXLAN
- Host-gw
- Protocolo de datagramas de usuario (UDP)
- Alivpc

- Alloc
- Nube privada virtual (VPC) de Amazon
- Google Compute Engine (GCE)
- IPIP
- IPsec

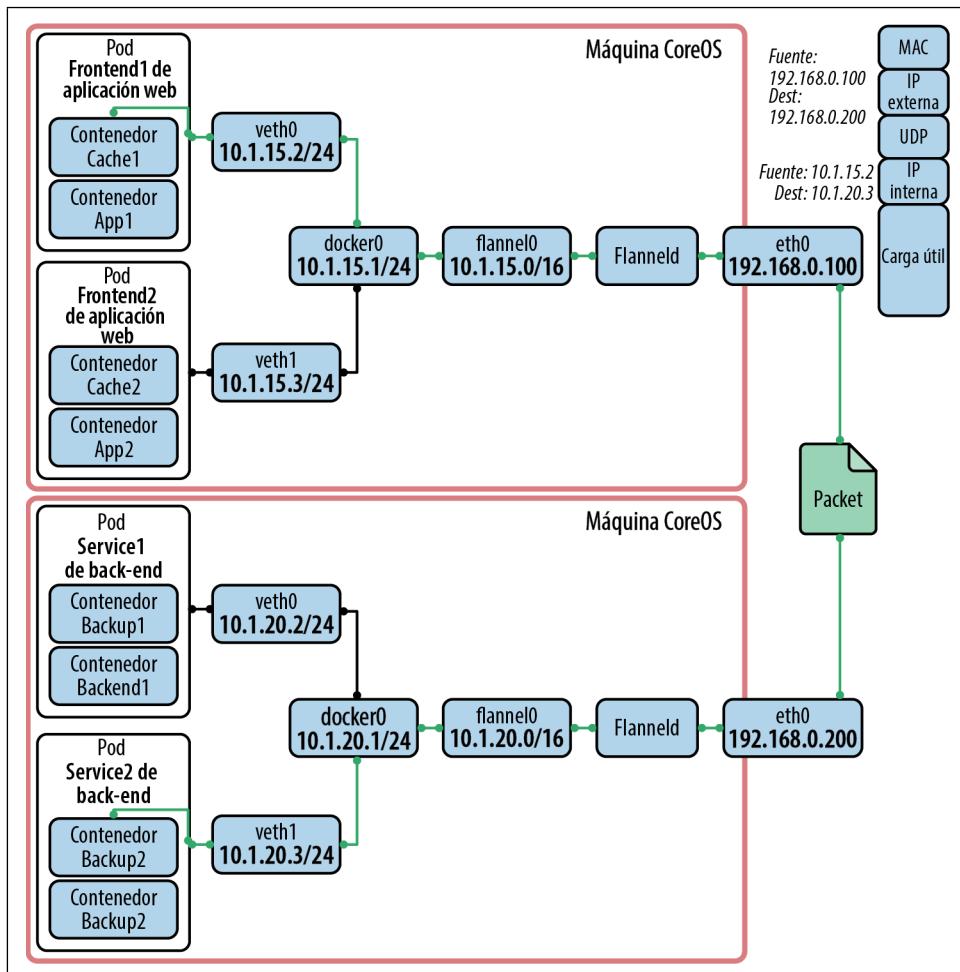


Figura 8-8. Ejemplo de la arquitectura de Flannel

El equipo del proyecto de Flannel recomienda VXLAN o host-gw como back-ends para la comunicación. Si necesita cifrar el tráfico, IPsec también es una opción. Puede obtener más información sobre los back-ends compatibles en la [documentación de Flannel](#).

Flannel no le permite ejecutar varias redes desde un demonio; tendrá que ejecutar demonios separados con diferentes archivos de configuración.

## Azure Policy

Aunque sin duda es útil ejecutar software que proteja su infraestructura basada en la nube, Azure Policy le permite establecer una política completa para muchos aspectos de su infraestructura de nube, en especial, en relación con la seguridad de los recursos.

Azure Policy realiza tres funciones clave:

- Crea políticas que impiden o aplican una configuración determinada.
- Permite informar sobre el cumplimiento de normas en su propia política, así como los puntos de referencia de la industria.
- Lleva a sus recursos a cumplir las normas a través de la corrección masiva de los recursos existentes y la corrección automática de los recursos nuevos.

Entre los casos de uso potenciales de Azure Policy, se encuentran implementar la gobernanza para lograr la coherencia de los recursos, el cumplimiento de normas, la seguridad, el costo y la administración.

Las definiciones de Azure Policy se encuentran en [formato JSON](#) y se aplican de manera jerárquica. Esto significa que una política se podría aplicar a un grupo de administración (y a todas sus suscripciones y recursos) y se podría establecer una política más indulgente en un grupo de administración distinto (y a todas sus suscripciones y recursos).

## Inicio rápido de Azure Policy

Azure contiene algunas políticas integradas de forma predeterminada. Para verlas, ejecute este comando en su terminal:

```
$ az policy definition list
```

O puede verlas en el [Portal](#) ([Figura 8-9](#)).

The screenshot shows the Azure Policy Definitions blade. On the left, there's a navigation sidebar with links like Overview, Getting started, Compliance, Remediation, Events, Authoring (with Definitions selected), Related Services (Blueprints, Resource Graph, User privacy), and a search bar. The main area has a header with 'Policy definition', 'Initiative definition', 'Export definitions', and a refresh button. Below the header are filters for Scope (Visual Studio En), Definition type (All definition types), Type (All types), Category (All categories), and a search bar. A note at the top says 'Now export your definitions and assignments to GitHub and manage them using actions! Click on 'Export definition' menu option. Learn more [here](#)'. The main table lists 18 policy definitions with columns for Name, Definition location, Policies, Type, Definition type, and Category. Some rows show preview status or specific compliance standards.

Name	Definition location	Policies	Type	Definition type	Category
[Preview]: NIST SP 800-1...		77	Built-in	Initiative	Regulatory Comp
Audit machines with inse...		9	Built-in	Initiative	Guest Configurat
IRS1075 September 2016		62	Built-in	Initiative	Regulatory Comp
[Preview]: Deploy prereq...		4	Built-in	Initiative	Guest Configurat
CIS Microsoft Azure Fou...		99	Built-in	Initiative	Regulatory Comp
Azure Security Benchmark		194	Built-in	Initiative	Security Center
[Preview]: Australian Gov...		61	Built-in	Initiative	Regulatory Comp
UK OFFICIAL and UK NHS		59	Built-in	Initiative	Regulatory Comp
[Preview]: SWIFT CSP-CS...		61	Built-in	Initiative	Regulatory Comp
Kubernetes cluster pod s...		8	Built-in	Initiative	Kubernetes
PCI v3.2.1:2018		39	Built-in	Initiative	Regulatory Comp
Canada Federal PBMM		59	Built-in	Initiative	Regulatory Comp
Enable Azure Monitor fo...		10	Built-in	Initiative	Monitoring
[Preview]: CIS Microsoft ...		104	Built-in	Initiative	Regulatory Comp
Flow lone chnclt ha crnf		?	Built-in	Initiative	Network

Figura 8-9. Información general de las definiciones de políticas de Azure disponibles

Azure Policy también incluye *iniciativas*, que son colecciones de políticas que se alinean con un estándar de la industria (por ejemplo, PCI v3.2.1:2018).

En este punto, se crearon las políticas, pero no tienen un ámbito asignado. Puede aplicar una política a una suscripción (y a todos los recursos dentro de la suscripción), o puede aplicarla a un grupo de administración que abarcará todas las suscripciones y los grupos de administración secundarios.

De cualquier manera, necesitará tener registrado el proveedor de recursos Microsoft. **PolicyInsights** para sus suscripciones. Para habilitarlo, ejecute el siguiente código:

```
$ az provider register --namespace 'Microsoft.PolicyInsights'
```

Azure Policy tiene varios modos de aplicación de políticas, también conocidos como *efectos*. Actualmente, existen siete tipos de efectos de Azure Policy:

- Append
- Audit
- AuditIfNotExists
- Deny

- DeployIfNotExists
- Disabled
- Modify

Puede leer sobre su uso en el artículo “[Comprenda los efectos de Azure Policy](#)”.

## Creación de su propia política de Azure

Crear su propia política de Azure es fácil. En el siguiente ejemplo, crearemos una política que solo permita la implementación de recursos en regiones de Azure que se encuentren dentro de los Estados Unidos. Si un usuario intenta crear un recurso en una región que no sea de EE. UU., se denegará:

```
{
  "properties": {
    "displayName": "Allowed locations",
    "description": "This policy enables you to restrict the locations your organization can specify when deploying resources.",
    "mode": "Indexed",
    "metadata": {
      "version": "1.0.0",
      "category": "Locations"
    },
    "parameters": {
      "allowedLocations": {
        "type": "array",
        "metadata": {
          "description": "The list of locations that can be specified when deploying resources",
          "strongType": "location",
          "displayName": "Allowed locations"
        },
        "defaultValue": [ "westus2", "eastus", "eastus2", "southcentralus",
          "centralus", "northcentralus", "westus", "westcentralus", "westus3" ]
      }
    },
    "policyRule": {
      "if": {
        "not": {
          "field": "location",
          "in": "[parameters('allowedLocations')]"
        }
      },
      "then": {
        "effect": "deny"
      }
    }
  }
}
```

Para crear la política, ejecutaremos el siguiente código:

```
$ az policy definition create --name 'allowed-regions' --display-name 'Deny non-US regions' \
--description 'This policy ensures that resources are only created in US regions.' \
--rules 'region-rule.json' --params 'region-params.json' --mode All
```

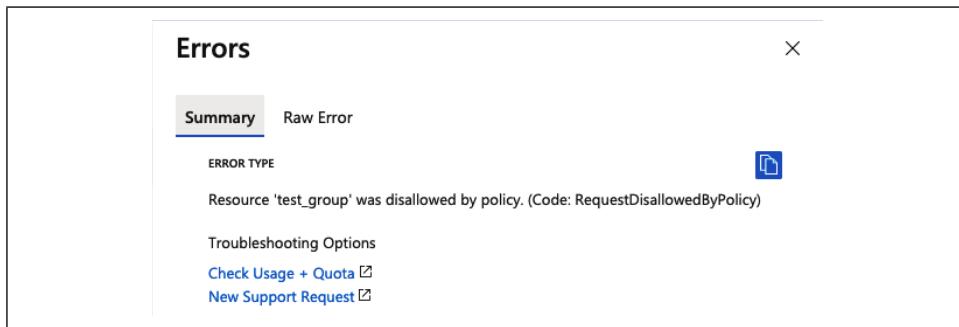
En la respuesta, veremos la ruta de acceso a la política nueva en el campo **id**, por ejemplo:

```
"id": "/subscriptions/<subscription-ID>/providers/Microsoft.Authorization/ \
policyDefinitions/allowed-regions"
```

Para asignar nuestra política a la suscripción, ejecutaremos el siguiente código:

```
$ az policy assignment create --name 'allowed-regions' --scope ' \
/subscriptions/<subscription-id>' --policy '/subscriptions/<subscription-ID>/ \
providers/Microsoft.Authorization/policyDefinitions/allowed-regions"
```

Ahora se aplicará la política. Si intentáramos crear un recurso en una región que no fuera de los Estados Unidos, recibiríamos un error similar al que se muestra en la [Figura 8-10](#).



*Figura 8-10. Mensaje de error sobre la implementación de un nuevo grupo de recursos debido a la política de Azure*



### Límites de Azure Policy

Azure Policy tiene límites en términos de la cantidad de políticas, excepciones, parámetros y condicionales que se pueden aplicar. Estos límites pueden cambiar según el ámbito de la política. Lea [la documentación](#) para obtener más detalles.

## Azure Policy para Kubernetes

Azure Policy también admite Kubernetes, lo que le permite obtener información sobre la política en la implementación de Kubernetes y crear políticas relacionadas con Kubernetes. Para obtener más información sobre cómo hacerlo, lea ["Comprenda Azure Policy para clústeres de Kubernetes"](#) y ["Proteja su clúster con Azure Policy"](#), que lo guiarán en la aplicación de Azure Policy en Kubernetes.

# Open Policy Agent

Open Policy Agent (OPA) es un motor de políticas de uso general de open source, que le permite realizar acciones de políticas de autorización en su software ([Figura 8-11](#)). OPA le permite especificar la política como código y proporciona API sencillas para que el software se integre con ellas.

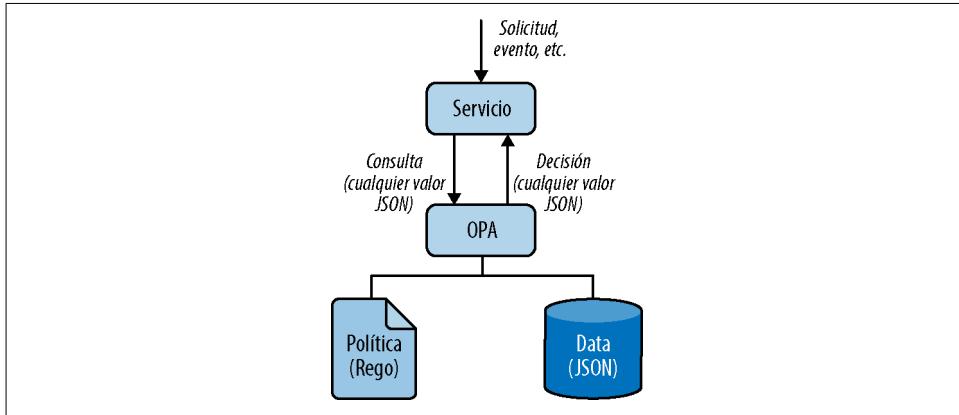


Figura 8-11. Cómo funciona OPA

Las políticas OPA se codifican en un lenguaje declarativo llamado Rego (consulte la [documentación sobre el lenguaje de las políticas](#)). Luego, las políticas se implementan en el servicio OPA en el host. A diferencia de varios de los sistemas que analizamos en este capítulo, OPA no realiza ninguna aplicación, más bien, toma decisiones respecto a las políticas. OPA evalúa las políticas y los datos para producir los resultados de las consultas que se envían al cliente. Las políticas se escriben en un lenguaje declarativo de alto nivel y se pueden cargar dinámicamente en OPA, de forma remota a través de API o a través del sistema de archivos local.

OPA proporciona una API de REST que los servicios o sistemas de terceros como Kubernetes, SSH, Sudo y Envoy pueden utilizar para responder a preguntas como las siguientes:

- Qué usuarios pueden acceder a qué recursos
- Qué tráfico de salida de subredes se permite
- En qué clústeres se debe implementar una carga de trabajo
- Qué archivos binarios de registros se pueden descargar
- Con qué capacidades de SO se puede ejecutar un contenedor
- A qué horas del día se puede acceder al sistema

Puede obtener más detalles en la [página de la documentación de la API de REST de OPA](#). Las API clave que ofrece OPA son las siguientes:

**GET /v1/policies**  
Obtiene todas las políticas

**GET /v1/policies/<name>**  
Devuelve una política específica

**PUT /v1/policies/<name>**  
Crea o actualiza una política

**DELETE /v1/policies/<name>**  
Elimina una política

**GET/POST /v1/data/<name>**  
Permite devolver o crear un documento

OPA le permite centralizar varias políticas de seguridad en múltiples piezas de infraestructura en un solo sistema. Puede consultar una lista completa de las integraciones de OPA en la [página Ecosistema de OPA](#). Sin OPA, necesitaría implementar la administración de políticas para su software desde cero. Los componentes requeridos, como el lenguaje de políticas (sintaxis y semántica) y el motor de evaluación, deben diseñarse, implementarse, probarse y documentarse cuidadosamente y, luego, mantenerse para garantizar un comportamiento correcto y una experiencia de usuario positiva para sus clientes.

## Implementación de OPA en Kubernetes

Es sencillo implementar OPA en Kubernetes:

1. Cree el archivo de definición de implementación (*deployment-opa.yaml*):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: opa
  labels:
    app: opa
spec:
  replicas: 1
  selector:
    matchLabels:
      app: opa
  template:
    metadata:
      labels:
        app: opa
        name: opa
    spec:
      containers:
        - name: opa
          image: openpolicyagent/opa:edge
          ports:
            - name: http
              containerPort: 8181
```

```

args:
- "run"
- "--ignore=.*" # exclude hidden dirs created by Kubernetes
- "--server"
- "/policies"
volumeMounts:
- readOnly: true
  mountPath: /policies
  name: policy
volumes:
- name: policy
  configMap:
    name: policy

```

- Cree el archivo de definición de servicio (*service-opa.yaml*):

```

kind: Service
apiVersion: v1
metadata:
  name: opa
  labels:
    app: opa
spec:
  type: NodePort
  selector:
    app: opa
  ports:
    - name: http
      protocol: TCP
      port: 8181
      targetPort: 8181

```

- Aplique ambos archivos:

```

$ kubectl create -f deployment-opa.yaml
$ kubectl create -f service-opa.yaml

```

## Implementación de políticas con OPA

Como se indicó antes, OPA es un motor de políticas generalizado y puede conectarse con una serie de **ecosistemas** de software. En este ejemplo, aprovisionaremos el acceso SSH a las máquinas mediante OPA.

Para este ejemplo, tendremos tres grupos:

*sre*

Estos son los administradores de todas las aplicaciones.

*foo-frontend*

Estos son los contribuyentes a la aplicación foo-frontend.

*bar-backend*

Estos son los contribuyentes a la aplicación bar-backend.

Implementaremos la siguiente política:

- El grupo `sre` puede aplicar SSH a cualquier host y ejecutar comandos sudo.
- Otros usuarios en grupos asociados con el desarrollo de la aplicación pueden iniciar sesión en el host, pero no pueden ejecutar comandos sudo.

Los siguientes ejemplos crearán una política que permite la autorización SSH para el grupo `sre` y el grupo asociado con la aplicación:

```
package sshd.authz

import input.pull_responses
import input.sysinfo

import data.hosts

# By default, users are not authorized.
default allow = false

# Allow access to any user that has the "admin" role.
allow {
    data.roles["sre"][_] == input.sysinfo.pam_username
}

# Allow access to any user who contributed to the code running on the host.
#
# This rule gets the "host_id" value from the file "/etc/host_identity.json."
# It is available in the input under "pull_responses" because we
# asked for it in our pull policy above.
#
# It then compares all the contributors for that host against the username
# that is asking for authorization.
allow {
    hosts[pull_responses.files["/etc/host_identity.json"].host_id].contributors[_] == \
        sysinfo.pam_username
}

# If the user is not authorized, then include an error message in the response.
errors["Request denied by administrative policy"] {
    not allow
}
```

Crearemos una segunda política que solo permita acceso sudo para el grupo `sre`:

```
package sudo.authz

# By default, users are not authorized.
default allow = false

# Allow access to any user that has the "admin" role.
allow {
    data.roles["sre"][_] == input.sysinfo.pam_username
}

# If the user is not authorized, then include an error message in the response.
errors["Request denied by administrative policy"] {
    not allow
}
```

Ahora podemos registrar estas dos políticas en OPA:

```
curl -X PUT --data-binary @sshd_authz.rego localhost:8181/v1/policies/sshd/authz  
curl -X PUT --data-binary @sudo_authz.rego localhost:8181/v1/policies/sudo/authz
```

Después de ejecutar correctamente estos comandos, se aplicará nuestra política y solo los usuarios del grupo `sre` tendrán acceso SSH y sudo a todas las máquinas. Mientras tanto, el grupo de servicios pertinente solo tendrá acceso SSH a la máquina en la que se implemente el servicio.

## Resumen

Aunque la infraestructura nativa de la nube operativa aporta inmensos beneficios al operador, presenta un riesgo significativo para su negocio si no se encuentra bien protegida. En este capítulo, revisamos un método para proteger y auditar su infraestructura. En primer lugar, revisamos el estándar CNI y de qué manera es la base para otras plataformas como Flannel y Cilium. Examinamos cómo se pueden usar esos sistemas para proporcionar conectividad y seguridad de red a su infraestructura de nube. Luego, analizamos los mecanismos de políticas que no son de red para administrar la infraestructura de nube a través de Azure Policy, además de la política para aplicaciones a través de Open Policy Agent. Crear su infraestructura de nube con una configuración de red segura, además de una política bien definida, lo ayudará a garantizar que su infraestructura escale de forma segura. Ahora que comprende los conceptos básicos de los contenedores y las redes en la nube, cambiaremos el análisis a comprender cómo almacenar y suministrar datos en la nube.

# **Almacenamiento y bases de datos distribuidas: El Banco Central**

Uno de los mayores obstáculos que superar cuando se crea una infraestructura nativa de la nube es operar un almacenamiento confiable y escalable para usar tanto en línea como sin conexión. Aunque la opinión anterior en torno a los servicios de almacenamiento que se ejecutan en la nube fue desaprobada, una serie de proyectos de almacenamiento nativos de la nube han tenido éxito, siendo el más destacado Vitess. Los servicios en la nube, incluido Azure, también han invertido significativamente en **discos administrados** para máquinas virtuales y conjuntos de escalado de máquinas virtuales, así como cuentas de almacenamiento avanzadas (como Gen2) y servicios de bases de datos administradas (como MySQL, SQL Server y Redis). Todo esto ha convertido el almacenamiento en un concepto de primera clase en la nube de Azure. En este capítulo, analizaremos por qué debe ejecutar soluciones de almacenamiento a escala en Azure y cómo puede organizarlas de una manera que sea fácil de administrar.

## **La necesidad de contar con bases de datos distribuidas en la arquitectura nativa de la nube**

Aunque Azure proporciona una serie de soluciones de datos administrados (tanto SQL como NoSQL), a veces, es posible que necesite ejecutar configuraciones más grandes y más especializadas mientras opera en la nube. El ecosistema de **proyectos de bases de datos operadas automáticamente de Cloud Native Computing Foundation (CNCF)** le permitirá escalar a un tamaño mayor que los servicios de plataforma como servicio (PaaS) de Azure, y le ofrecerá más características y opciones de rendimiento.

Como se analizará con más detalle en este capítulo, el panorama de CNCF establece un conjunto de soluciones de almacenamiento diseñadas principalmente para las operaciones en la nube. El beneficio de estos sistemas es que se minimiza la sobrecarga de administración de servidores, a la vez que aún tiene el control total del sistema.

En este capítulo, examinaremos algunos de los almacenes de datos maduros que forman parte del panorama de la CNCF. Vitess, Rook, TiKV y etcd son todos almacenes de datos en la nube maduros, capaces de escalar a un tamaño enorme y se ejecutan como sistemas de datos de primera clase en Azure. Estos sistemas representan los bloques de creación fundamentales de un ecosistema de datos que opera en una infraestructura de nube.



### Ejecución de cargas de trabajo con estado en AKS

En el momento de redactar este documento, Azure Kubernetes Service (AKS) admite varias zonas de disponibilidad (es decir, varios centros de datos dentro de una región). Sin embargo, no es compatible con varios dominios de error. Esto significa que, si el clúster de AKS está en una sola zona de disponibilidad, teóricamente es posible tener tiempo de inactividad porque varios nodos del clúster no están disponibles. Le recomendamos que divida el clúster de AKS en varias zonas de disponibilidad hasta que AKS admita los dominios de error.

## Opciones de bases de datos y almacenamiento de Azure

Antes de comenzar a crear sus almacenes de datos en la nube, debería evaluar exhaustivamente su caso de uso, en especial, el almacenamiento, el rendimiento y el costo de lo que está tratando de compilar. Es posible que descubra que los servicios de PaaS de Azure podrían adaptarse mejor a un caso de uso específico, en lugar de usar un producto de CNCF. Por ejemplo, la oferta de almacenamiento de Sistemas de archivos distribuido de Hadoop (HDFS) de Azure y las cuentas de almacenamiento Gen2 son soluciones extremadamente económicas para almacenar grandes volúmenes de datos.

En el momento de redactar este documento, Azure proporciona los siguientes servicios de PaaS de almacenamiento y base de datos:

- Azure Cosmos DB
- Azure Cache for Redis
- Azure Database for:
  - MySQL
  - PostgreSQL
  - MariaDB
- Cuentas de almacenamiento (que ofrecen blob, Network File System [NFS] y almacenamiento jerárquico)
- Azure Storage Explorer

Le recomendamos que evalúe el rendimiento de estas ofertas de Azure y considere específicamente el rendimiento, la disponibilidad y el costo de un servicio de PaaS, para asegurarse de estar usando el sistema correcto para su caso de uso. Azure proporciona una **herramienta de calculadora de precios** que es extremadamente útil para el modelado de costos.

Si cree que un servicio de PaaS de Azure no se adapta a sus requisitos, siga leyendo este capítulo para obtener información sobre otras posibles soluciones de almacenamiento de datos.

## Introducción a Vitess: MySQL distribuida y particionada

Vitess es un sistema de clúster de base de datos para el escalado horizontal de MySQL y MariaDB. Fue creado por los ingenieros de YouTube en 2010 como un medio para proteger y escalar sus bases de datos. Proporciona una serie de características que ayudan en el rendimiento, la seguridad y la supervisión, pero lo más importante es que ofrece herramientas eficaces para administrar topologías de bases de datos y proporcionar soporte de primera clase para particionar y volver a particionar vertical y horizontalmente. En muchos aspectos, le permite operar bases de datos SQL de forma NoSQL, a la vez que obtiene todos los beneficios de la ejecución de una base de datos relacional.

### ¿Por qué ejecutar Vitess?

Es posible que se pregunte: “¿por qué debo ejecutar Vitess, en lugar de usar Azure Cosmos DB?”. Aunque Cosmos DB es una opción atractiva para las escrituras y réplicas distribuidas de forma global, el modelo de costos actual hace que sea prohibitivamente costoso ejecutarlo a escala (miles de consultas por segundo), lo que hace que un sistema como Vitess sea una oferta atractiva para aquellos que necesitan una base de datos relacional de alto rendimiento con una configuración más personalizable.

Vitess proporciona los siguientes beneficios:

#### *Escalabilidad*

Puede escalar y reducir verticalmente los esquemas individuales.

#### *Capacidad de administración*

Un plano de control centralizado administra todas las operaciones relacionadas con la base de datos.

#### *Protección*

Puede reescribir las consultas, crear listas de denegación de consultas y agregar listas de control de acceso (ACL) a nivel de tabla.

#### *Administración de partición*

Puede crear y administrar fácilmente particiones de base de datos.

#### *Rendimiento*

Muchos de los datos internos están diseñados para obtener rendimiento, incluida la agrupación de conexiones del cliente y la eliminación de duplicados de consultas.

Vitess lo ayuda a administrar la sobrecarga operativa y de escalado de las bases de datos relacionales y el estado de su clúster. Además, ofrece una gama más amplia de configurables que las ofertas de bases de datos en la nube tradicionales (incluida Cosmos DB), lo que lo ayudará a satisfacer sus requisitos de escalado y replicación.

## La arquitectura de Vitess

Vitess proporciona particionamiento como servicio mediante la creación de un sistema de estilo middleware mientras se sigue utilizando MySQL para almacenar datos. La aplicación cliente se conecta a un demonio conocido como VTGate (consulte la Figura 9-1). VTGate es un enrutador compatible con clústeres, que enruta las consultas a las instancias apropiadas de VTTablet, que administran cada instancia de MySQL. El servicio de topología almacena la topología de la infraestructura, incluido el lugar donde se almacenan los datos y la capacidad de cada instancia de VTTablet para suministrar los datos.

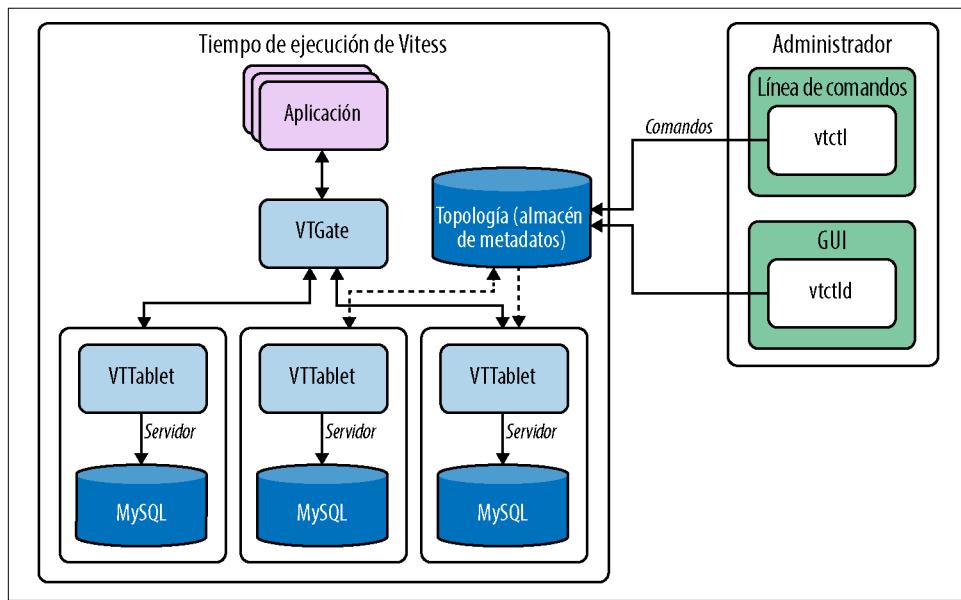


Figura 9-1. La arquitectura de Vitess

Vitess también proporciona dos interfaces administrativas: vtctl, una CLI; y vtctld, una interfaz web.

VTGate es un enrutador de consultas (o proxy) para enrutar las consultas del cliente a las bases de datos. El cliente no necesita saber nada, excepto donde se encuentra la instancia de VTGate, lo que significa que la arquitectura de cliente a servidor es sencilla.

# Implementación de Vitess en Kubernetes

Vitess tiene soporte de primera clase para Kubernetes y es muy sencillo comenzar a usarlo. En este tutorial, crearemos un clúster y un esquema y, luego, expandiremos el clúster:

1. En primer lugar, clone el repositorio *vitess* y mírela a la carpeta *vitess/examples/operator*:

```
$ git clone git@github.com:vitessio/vitess.git  
$ cd vitess/examples/operator
```

2. Ahora, ejecute el siguiente código para instalar el operador de Kubernetes:

```
$ kubectl apply -f operator.yaml
```

3. A continuación, ejecute lo siguiente para crear un clúster de Vitess inicial:

```
$ kubectl apply -f 101_initial_cluster.yaml
```

4. Podrá verificar si la instalación inicial del clúster se realizó correctamente de la siguiente manera:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
example-etcd-faf13de3-1	1/1	Running	0	78s
example-etcd-faf13de3-2	1/1	Running	0	78s
example-etcd-faf13de3-3	1/1	Running	0	78s
example-vttablet-zone1-2469782763-bfadd780	3/3	Running	1	78s
example-vttablet-zone1-2548885007-46a852d0	3/3	Running	1	78s
example-zone1-vtctld-1d4dcad0-59d8498459-kwz6b	1/1	Running	2	78s
example-zone1-vtgate-bc6cde92-6bd99c6888-vwcj5	1/1	Running	2	78s
vitess-operator-8454d86687-4wfnc	1/1	Running	0	2m29s

Ahora tiene un clúster de Vitess pequeño de una sola zona que se ejecuta con dos réplicas.

Puede obtener más ejemplos de operación de base de datos en la [página de ejemplo de Operaciones de Kubernetes de Vitess](#).

Uno de los casos de uso más comunes que encontrará es la necesidad de agregar o quitar réplicas de la base de datos. Puede hacerlo fácilmente mediante la ejecución del siguiente código:

```
$ kubectl edit planetscale.com example
```

```
# To add or remove replicas, change the replicas field of the appropriate resource
# to the desired value. E.g.

keyspaces:
- name: commerce
  turndownPolicy: Immediate
  partitionings:
  - equal:
      parts: 1
      shardTemplate:
        databaseInitScriptSecret:
          name: example-cluster-config
          key: init_db.sql
      replication:
        enforceSemiSync: false
    tabletPools:
```

```
- cell: zone1
  type: replica
  replicas: 2 # Change this value
```



Algo que debe considerar en la creación de almacenamiento de datos distribuido es determinar el tamaño de su dominio de error. Vitess le recomienda almacenar un máximo de 250 GB de datos por servidor. Aunque hay algunas razones de rendimiento de MySQL para esto, otras tareas operativas normales se vuelven más difíciles con un mayor tamaño de partición/dominio de error. Puede leer más sobre por qué Vitess recomienda 250 GB por servidor en el [blog de Vitess](#).

## Introducción a Rook: Orquestador de almacenamiento para Kubernetes

Hasta ahora, hemos hablado del almacenamiento de bases de datos SQL relacionales y del servicio de alto rendimiento, pero hay otros tipos de almacenamiento que deben suministrarse y operarse en la nube. Estos se conocen como *sistemas de archivos blob*. Los almacenes blob son excelentes para almacenar imágenes, videos y otros archivos que no son adecuados para una base de datos relacional. El almacenamiento blob incluye desafíos adicionales en torno al almacenamiento y la replicación de grandes cantidades de datos de manera eficiente.

[Rook](#) es un orquestador de almacenamiento open source de archivos, bloques y objetos nativos de la nube que proporciona almacenamiento de blobs definido por software. Rook se graduó como un proyecto CNCF en 2018 y está específicamente diseñado en torno a la ejecución de almacenamiento en Kubernetes. Al igual que Vitess, Rook administra una gran cantidad de las operaciones diarias de un clúster, incluido el escalado automático y la recuperación automática, y automatiza otras tareas, como la recuperación ante desastres, la supervisión y las actualizaciones.

### La arquitectura de Rook

Rook es un orquestador de Kubernetes, no una solución de almacenamiento real. Rook admite los siguientes sistemas de almacenamiento:

- Ceph, una solución de almacenamiento distribuido altamente escalable para almacenamiento en bloque, almacenamiento de objetos y sistemas compartidos con años de implementaciones de producción
- Cassandra, una base de datos NoSQL altamente disponible con un rendimiento ultrarrápido, coherencia ajustable y escalabilidad masiva
- NFS, que permite a los hosts remotos montar sistemas de archivos en una red e interactuar con esos sistemas como si estuvieran montados localmente

Rook puede orquestar varios proveedores de almacenamiento en un clúster de Kubernetes. Cada uno tiene su propio operador para implementar y administrar los recursos.

# Implementación de Rook en Kubernetes

En este ejemplo, implementaremos un clúster de Cassandra con el operador Rook (Cassandra)Kubernetes. Si está implementando un clúster de Ceph mediante Rook, también puede usar un [gráfico de Helm](#) para realizar la implementación:

1. Clone el operador Rook:

```
$ git clone --single-branch --branch master https://github.com/rook/rook.git
```

2. Instale el operador:

```
$ cd rook/cluster/examples/kubernetes/cassandra  
$ kubectl apply -f operator.yaml
```

3. Ejecute el siguiente código para verificar que el operador esté instalado:

```
$ kubectl -n rook-cassandra-system get pod
```

4. Ahora, vaya a la carpeta *cassandra* y cree el clúster:

```
$ cd rook/cluster/examples/kubernetes/cassandra  
$ kubectl create -f cluster.yaml
```

5. Para comprobar que se estén ejecutando todos los nodos deseados, emita el siguiente comando:

```
$ kubectl -n rook-cassandra get pod -l app=rook-cassandra
```

Es muy fácil escalar y reducir verticalmente el clúster de Cassandra con el comando `kubectl edit` de Kubernetes. Simplemente debe aumentar o reducir el valor `Spec.Members` para escalar o reducir el clúster:

```
$ kubectl edit clusters.cassandra.rook.io rook-cassandra  
# To scale up a rack, change the Spec.Members field of the rack to the desired value.  
# To scale down a rack, change the Spec.Members field of the rack to the desired value  
# After editing and saving the yaml, check your cluster's Status and Events for information  
# on what's happening:  
  
apiVersion: cassandra.rook.io/v1alpha1  
kind: Cluster  
metadata:  
  name: rook-cassandra  
  namespace: rook-cassandra  
spec:  
  version: 3.11.6  
  repository: my-private-repo.io/cassandra  
  mode: cassandra  
  annotations:  
  datacenter:  
    name: us-east2  
  racks:  
    - name: us-east2  
      members: 3 # Change this number up or down  
  
$ kubectl -n rook-cassandra describe clusters.cassandra.rook.io rook-cassandra
```



Puede encontrar más detalles sobre los configurables de Cassandra en la [documentación de CRD de Cassandra](#).

Del mismo modo, los clústeres [Ceph](#) y [NFS](#) se pueden implementar fácilmente con el operador Rook.

## Introducción a TiKV

[TiKV](#) (Titanium Key-Value), creado por PingCAP, Inc., es un almacén de clave-valor open source, distribuido y transaccional. Al contrario de muchos sistemas clave-valor y NoSQL, TiKV proporciona API simples (sin procesar), así como API transaccionales que proporcionan cumplimiento de atomicidad, coherencia, aislamiento y durabilidad (ACID)

### ¿Por qué usar TiKV?

TiKV proporciona las siguientes características:

- Replicación geográfica
- Escalabilidad horizontal
- Transacciones distribuidas coherentes
- Compatibilidad con coprocesador
- Particionamiento automático

TiKV es una opción atractiva debido a sus características de particionamiento automático y replicación geográfica, así como su capacidad de escalar para almacenar más de 100 TB de datos. También proporciona garantías de consenso sólidas con su uso del protocolo Raft para la replicación.

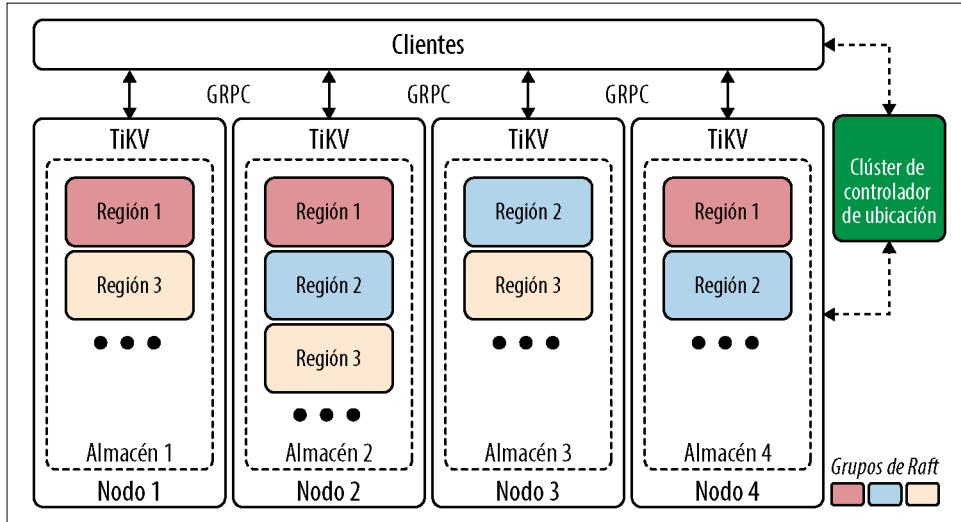
## La arquitectura de TiKV

Como se mencionó antes, TiKV tiene dos API que se pueden utilizar para diferentes casos de uso: sin procesar y transaccional. En la [Tabla 9-1](#) se describen las diferencias entre estas dos API.

*Tabla 9-1. API de TiKV: sin procesar en comparación con transaccional*

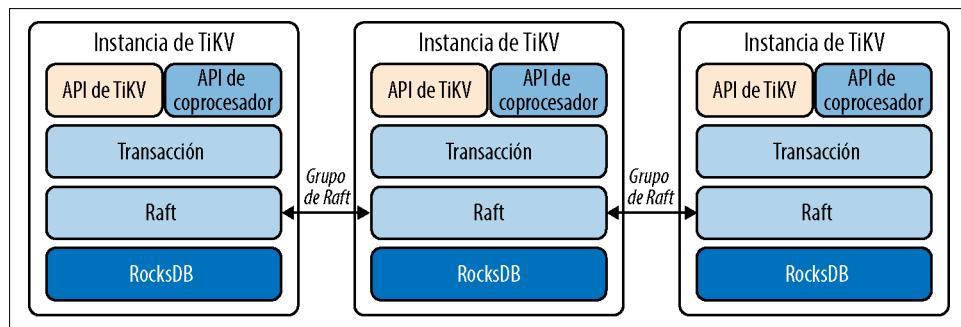
	Sin procesar	Transaccional
Descripción	Una API de clave-valor de nivel inferior para interactuar directamente con pares de clave-valor individuales	Una API de clave-valor de mayor nivel que proporciona semántica ACID
Atomicidad	Clave única	Varias claves
Usar cuando...	Su aplicación no requiera transacciones distribuidas o control de concurrencia de varias versiones (MVCC)	Su aplicación requiera transacciones distribuidas y/o MVCC

TiKV utiliza RocksDB como contenedor de almacenamiento en cada nodo y utiliza grupos Raft para proporcionar transacciones distribuidas, como se muestra en la [Figura 9-2](#). El controlador de ubicación actúa como un administrador de clústeres y garantiza que todas las particiones hayan cumplido con sus restricciones de replicación y que se haya equilibrado la carga de los datos en el grupo. Por último, los clientes se conectan a los nodos de TiKV mediante el protocolo Google Remote Procedure Call (gRPC), que está optimizado para el rendimiento.



*Figura 9-2. La arquitectura de TiKV*

Dentro de un nodo TiKV (que se muestra en la [Figura 9-3](#)), RocksDB proporciona los mecanismos de almacenamiento subyacentes y Raft proporciona consenso para las transacciones. La API de TiKV proporciona una interfaz para que los clientes interactúen, y el coprocesador maneja consultas similares a las de SQL y reúne el resultado del almacenamiento subyacente.



*Figura 9-3. Arquitectura de instancia de TiKV*

# Implementación de TiKV en Kubernetes

TiKV se puede implementar a través de varios métodos de automatización, incluidos Ansible, Docker y Kubernetes. En el siguiente ejemplo, implementaremos un clúster básico con Kubernetes y Helm:

1. Instale la definición de recursos personalizados de TiKV:

```
$ kubectl apply -f https://raw.githubusercontent.com/tikv/tikv-operator/master/ \
  manifests/crd.v1beta1.yaml
```

2. Tendremos que explorar el operador de Helm. En primer lugar, agregue el repositorio PingCap:

```
$ helm repo add pingcap https://charts.pingcap.org/
```

3. Ahora cree un espacio de nombres para `tikv-operator`:

```
$ kubectl create ns tikv-operator
```

4. Instale el operador:

```
$ helm install --namespace tikv-operator tikv-operator pingcap/tikv-operator \
  --version v0.1.0
```

5. Implemente el clúster:

```
$ kubectl apply -f https://raw.githubusercontent.com/tikv/tikv-operator/master/ \
  examples/basic/tikv-cluster.yaml
```

6. Ejecute el siguiente código para comprobar el estado de la implementación:

```
$ kubectl wait --for=condition=Ready --timeout 10m tikvcluster/basic
```

Esto creará un clúster de almacenamiento de un solo host con 1 GB de almacenamiento y una instancia de controlador de ubicación (PD).

Puede modificar los parámetros `replicas` y `storage` en la definición del clúster para que coincidan con sus requisitos. Le recomendamos que ejecute el almacenamiento con al menos tres réplicas para fines de redundancia. Por ejemplo, si ejecuta `kubectl edit tikv. org TikvCluster` y modifica `replicas` a 4 y `storage` a 500Gi:

```
apiVersion: tikv.org/v1alpha1
kind: TikvCluster
metadata:
  name: basic
spec:
  version: v4.0.0
  pd:
    baseImage: pingcap/pd
    replicas: 4
    # if storageClassName is not set, the default Storage Class of the Kubernetes cluster
    # will be used
    # storageClassName: local-storage
    requests:
      storage: "1Gi"
      config: {}
  tikv:
    baseImage: pingcap/tikv
```

```

replicas: 4
# if storageClassName is not set, the default Storage Class of the Kubernetes cluster
# will be used
# storageClassName: local-storage
requests:
  storage: "500Gi"
config: {}

```

obtendrá cuatro réplicas de una partición de almacenamiento de 500 GB.

## Más información sobre etcd

Utilizamos etcd (sigilosamente) en todo este libro. Aquí dedicaremos algo de tiempo a hablar de este almacén en detalle.

etcd es un almacén de clave-valor distribuido muy coherente ([consenso Raft](#)), que está diseñado para su uso en sistemas distribuidos. Está diseñado para manejar fácilmente las elecciones de líderes y los errores, como las interrupciones de la red y el tiempo de inactividad inesperado de los nodos. etcd tiene una API sencilla ([Figura 9-4](#)), por lo que descubrirá que varios de los sistemas de los que hablamos en este libro (por ejemplo, Kubernetes, TiKV, Calico e Istio) lo aprovechan en segundo plano.

A menudo, etcd se utiliza para almacenar los valores de configuración, las marcas de características y la información de detección de servicios en un formato de clave-valor. etcd permite a los clientes mirar estos elementos en busca de cambios y se reconfigura cuando cambian. etcd también es adecuado para las elecciones de liderazgo y los usos de bloqueo. etcd se utiliza como back-end de descubrimiento de servicios en Kubernetes y para la orquestación en Rook.

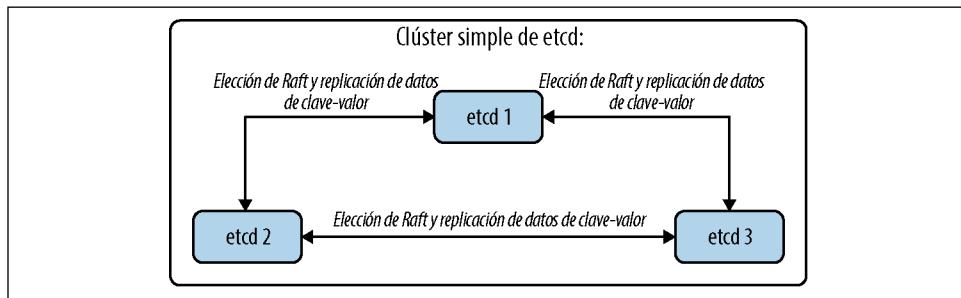


Figura 9-4. Arquitectura operativa de etcd

Si utiliza etcd para su infraestructura principal, entenderá rápidamente lo importante que es para esta infraestructura ejecutarse de forma confiable. Debido a que un clúster etcd es una parte fundamental de su infraestructura, recomendamos lo siguiente para garantizar un clúster etcd estable en Azure.

## Plataforma de hardware

El rendimiento coherente del almacén de clave-valor se basa en gran medida en el rendimiento del hardware subyacente. Un clúster de clave-valor coherente que tiene un mal rendimiento puede afectar gravemente el rendimiento de su sistema distribuido. Lo mismo se aplica para etcd. Si está realizando cientos de transacciones por segundo, debe utilizar una unidad de estado sólido (SSD) administrada con su máquina virtual (recomendamos SSD premium o ultra) para evitar la degradación del rendimiento. Necesitará al menos cuatro núcleos para que etcd se ejecute, y el rendimiento se escalará con un aumento de los núcleos de CPU. Puede encontrar más información sobre la planificación de capacidad y las configuraciones de hardware para la nube en la [página de hardware general](#) y la [guía de AWS](#) (etcd actualmente no publica una guía específica de Azure).

Le recomendamos ejecutar etcd en las siguientes SKU de Azure:

- Standard\_D8ds\_v4 (CPU de ocho núcleos)
- Standard\_D16ds\_v4 (CPU de 16 núcleos)

Estas SKU le permiten conectar de forma explícita SSD premium o ultra, lo que aumentará el rendimiento del clúster.



Al igual que con cualquier caso de uso de producción que hayamos comentado en este libro, le recomendamos encarecidamente que habilite las redes aceleradas en todas las instancias de proceso de Azure.

## Escalado automático y corrección automática

Seríamos negligentes si no hablábamos de cómo el escalado automático y la corrección automática funcionan con etcd. No se recomienda el escalado automático para etcd, puesto que podría tener consecuencias imprevistas si las adiciones o sustracciones de capacidad afectan negativamente el rendimiento del clúster. Dicho esto, se considera que está bien habilitar la corrección automática (recuperación automática) de las instancias de etcd incorrectas. Puede desactivar el escalado automático para etcd ejecutando el siguiente comando en el clúster de Kubernetes:

```
$ kubectl delete hpa etcd
```

Esto deshabilitará el Horizontal Pod Autoscaler (HPA), que se utiliza en Kubernetes para escalar y reducir verticalmente los clústeres de forma automática. Puede obtener más información sobre el HPA en la [documentación de Kubernetes](#).



## Escalado automático de Azure

Azure proporciona escalado automático como una característica en varios de sus productos, incluidos los conjuntos de escalado de máquinas virtuales y las funciones de Azure. Las características de escalado automático le permiten escalar y reducir verticalmente de forma eficiente sus recursos cuando sea necesario. Con los conjuntos de escalado de máquinas virtuales, puede haber un retraso de hasta 10 minutos para que las nuevas máquinas virtuales estén disponibles cuando se escala horizontalmente.

## Disponibilidad y seguridad

etcd es un servicio de plano de control, por lo que es importante que esté altamente disponible y que las comunicaciones con él sean seguras. Le recomendamos que ejecute al menos tres nodos etcd en un clúster para garantizar su disponibilidad. El rendimiento del clúster dependerá del rendimiento del almacenamiento. Recomendamos encarecidamente no usar mecanismos de detección estáticos para los nodos del clúster y, en su lugar, utilizar registros SRV de DNS según la [guía de agrupación en clústeres de etcd](#).

### TLS de etcd

etcd admite el cifrado de Seguridad de la capa de transporte (TLS) de cliente a servidor y de pares (servidor a servidor). Le recomendamos encarecidamente que lo habilite, puesto que etcd es un sistema de plano de control. Aquí no entraremos en detalles sobre la implementación (porque dependen en gran medida de su infraestructura). Puede encontrar opciones de configuración en la [documentación de seguridad de etcd](#).

### Control de acceso basado en roles

Desde etcd v2.1, el control de acceso basado en roles (RBAC) ha sido una característica de la API de etcd. Mediante RBAC, puede crear patrones de acceso que solo permitan a ciertos usuarios (o aplicaciones) acceder a un subconjunto de los datos de etcd. Tradicionalmente, esto se realiza mediante la entrega de autenticación básica HTTP al realizar solicitudes HTTP a etcd. A partir de etcd v3.2, si usa la opción de configuración `--client-cert-auth=true`, el nombre común (CN) del certificado de cliente TLS se utilizará como usuario (en lugar de una combinación de nombre de usuario/contraseña).

Puede encontrar ejemplos de cómo aplicar RBAC a su espacio de datos de etcd en la [documentación](#).

## Resumen

A pesar de las ideas erróneas comunes, es posible ejecutar sistemas de datos a gran escala de forma nativa en Azure. En este capítulo, revisamos por qué es beneficioso usar los sistemas de datos nativos de la nube en la nube de Azure. Abarcamos cuatro sistemas: Vitess (relacional), Rook (blob), TiKV (clave-valor) y etcd (configuración clave-valor/detección de servicios).

Estos sistemas son la piedra angular de una arquitectura nativa de la nube, que almacena y suministra datos en línea y ofrece una gran ventaja sobre la utilización de componentes PaaS. Por ahora, debe conocer qué software tiene sentido para que lo administre, así como qué servicios de PaaS utilizar y cómo implementarlos en su infraestructura.

La nube y, en especial, Kubernetes, se ha convertido en un lugar mucho más apto para ejecutar su infraestructura con estado. Aunque las cuentas de almacenamiento Gen2 de Azure son un excelente recurso para almacenamiento de blob, el software nativo de la nube puede ayudarlo a crear una infraestructura con estado a gran escala y duradera.

Ahora que comprende cómo puede almacenar y suministrar datos en un entorno de nube, examinemos cómo migrar datos entre sistemas mediante la mensajería en tiempo real.

# Recibir el mensaje

En este capítulo, analizaremos los beneficios de la mensajería en las arquitecturas nativas de la nube y proporcionaremos una breve historia de los patrones de mensajería para que pueda comprender su origen y la evolución de la comunicación entre procesos hacia los sistemas distribuidos nativos de la nube.

Al final de este capítulo, podrá implementar las implementaciones de mensajería más comunes que se usan actualmente en las aplicaciones nativas de la nube y los microservicios. Compararemos las capacidades de las soluciones de mensajería maduras, como RabbitMQ y Kafka, que han estado en producción durante años, e implementaremos patrones de mensajería comunes en NATS. Debido al enfoque del libro en Azure, podrá administrar las ofertas de mensajería de Azure con Terraform. También proporcionaremos pequeños ejemplos de código en Python, que muestran lo fácil que es producir y consumir mensajes y validar la implementación correcta de la infraestructura de mensajería para Azure y NATS.

## La necesidad de contar con mensajes

La necesidad de que los componentes de la aplicación y del sistema transmitan mensajes casi en tiempo real e intercambien datos no es exclusiva de la nube, ni nueva para los microservicios basados en contenedores, o incluso para las aplicaciones cliente/servidor en red. Cualquier persona que haya canalizado comandos en un comando shell ha utilizado la *comunicación entre procesos*, que es una forma básica de mensajería.

Incluso los sistemas operativos más primitivos ofrecen la capacidad de transmitir mensajes y compartir datos. Este espacio problemático (y las diversas soluciones) se remonta a la década de 1970 y continuó evolucionando en la década de 2000, en especial, en la industria de los servicios financieros, donde existía la necesidad de comunicar información de existencias a los clientes, intercambiar datos entre los proveedores de servicios y otras instituciones, y procesar pedidos. A diferencia de las interacciones cliente/servidor que encontramos con mayor frecuencia en Internet o en aplicaciones empresariales, la

mensajería se basa en un intermediario (llamado *agente*) que se coloca entre el emisor y el receptor, que generalmente no sigue el paradigma de solicitud-respuesta, de manera que la comunicación se vuelve asíncrona. Analizaremos por qué esto es importante en breve.

A medida que la nube maduraba en 2010, el uso de servicios administrados de mensajería, como los servidores de Amazon SQS y RabbitMQ que se ejecutan en máquinas virtuales, desempeñó un papel en el aumento de la confiabilidad y la escalabilidad de los sitios web. A mediados de 2010, Kafka se convirtió en una solución popular para los casos de uso de mensajería, incluida la incorporación de registros, el análisis de secuencia de clics y las canalizaciones de datos. NATS es otra implementación de mensajería ligera y de alto rendimiento que se ha vuelto popular para los análisis de perímetro y los casos de uso de la Internet de las Cosas (IoT), debido a su capacidad de reducir verticalmente hasta dispositivos de menor tamaño y su facilidad de operabilidad.

Las colas y los patrones de publicación/suscripción de mensajes (conocidos como “pub/sub”) se encontraban entre los primeros servicios administrados ofrecidos por los proveedores de nube. Azure tiene varios servicios de mensajería, incluidos Azure Event Grid, Azure Event Hubs y Azure Service Bus, que proporcionan protocolos e implementaciones basados en estándares que son compatibles con Java, .NET, Python y otros SDK. Estos servicios permiten la migración de cargas de trabajo existentes y/o el desarrollo de nuevas aplicaciones nativas de la nube.

Antes de profundizar en la historia de la mensajería, las siguientes son las razones por las que debe implementar la mensajería dentro de su infraestructura o para que la usen las aplicaciones:

#### *Mejora el rendimiento y la productividad*

El uso de patrones de mensajería (ya sean colas sencillas o patrones de publicación/suscripción más complejos) permite la escalabilidad horizontal y la parallelización de las cargas de trabajo. Escalar los productores, consumidores y agentes en función de la carga es posible a través de enfoques de escalado automático de máquinas virtuales, como los conjuntos de escalado de máquinas virtuales de Azure, a través de Kubernetes o mediante el servicio en la nube.

#### *Aumenta la resiliencia*

La capacidad de tener varios emisores (productores) y receptores (consumidores) que intercambian mensajes proporciona una mejor resiliencia. Además, el hecho de que los mensajes se puedan acumular en una cola (los emisores siguen enviándolos) mientras los consumidores no están procesándolos, garantiza que los mensajes no se pierdan y que pueda realizar tareas de mantenimiento en sus trabajadores. En función del patrón de cola, el sistema puede sobrevivir a errores de conectividad entre componentes, además de errores de aplicaciones en productores o consumidores.

### *Garantiza una operación asíncrona*

Aunque los temidos errores de tiempo de espera de gateway 502 en Apache y Nginx ocurren con menos frecuencia hoy en día, cuando los vemos, recordamos cómo los primeros sitios web alcanzaron problemas de escalabilidad cuando un gran número de solicitudes desbordaron la capacidad de responder de los servidores web. El mismo principio se aplica cuando un cliente de base de datos se conecta a una base de datos de back-end y la base de datos se queda sin subprocesos o se agota. Cuando los trabajadores escalan automáticamente en función de la profundidad de la cola, el resultado son altas tasas de procesamiento de mensajes que responden a la demanda. Las colas de mensajes también habilitan límites de servicio de menor ámbito y son comunes en las arquitecturas de aplicaciones de microservicios. Por último, debido a que las aplicaciones no se comunican directamente (como en el caso de las API de REST o RPC), pueden centrarse en los datos que se intercambian, suponiendo que están implementando un protocolo de mensajería estándar.

## **Ejemplo de caso de uso de mensajería: Ingesta y análisis de registros**

Para ilustrar algunas de las mejoras reales que introducen las colas de mensajes en una arquitectura, describiremos una solución de software como servicio (SaaS) de análisis de registros que podemos compilar y ejecutar. El procesamiento de registros y eventos es un caso de uso común para la implementación de servicios de mensajería y streaming.

El producto recopilará registros de seguridad y del sistema de dispositivos locales, y los analizará e ingerirá en un motor de búsqueda de texto completo que permite que los analistas de seguridad detecten y encuentren ataques en su red. Hemos simplificado drásticamente la descripción de esta plataforma, pero los siguientes ejemplos muestran los beneficios de introducir las colas de mensajes que mencionamos antes.

Analicemos tres generaciones de la arquitectura de este producto y exploremos cómo evolucionó el uso de la mensajería.

### **Generación 1: Sin colas**

La primera generación de la solución fue una colección de eventos en tiempo real desde un dispositivo local que se publicó en un equilibrador de carga, que distribuía registros a varios nodos de ingesta y búsqueda (recopiladores de registros), como se muestra en la [Figura 10-1](#). Se trataba de una arquitectura estrechamente acoplada en la que las grandes cantidades de eventos que se recopilaban (según los patrones de uso del cliente) podían provocar retrasos en la ingesta, lo que, a su vez, podría provocar problemas en la búsqueda de la interfaz de usuario.

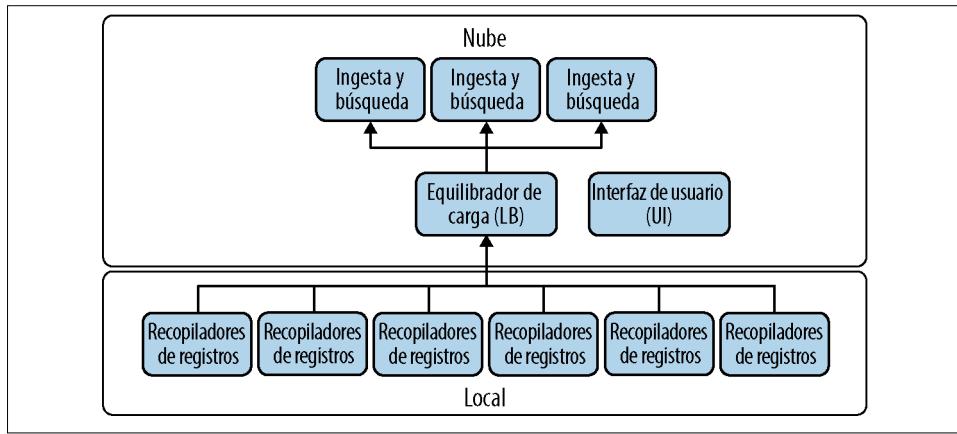


Figura 10-1. Generación 1, sin colas

## Generación 2: Con colas de la nube y almacenamiento de objetos

La segunda iteración de esta solución usaba el almacenamiento de objetos del proveedor de nube y las colas de mensajes para ingerir paquetes de mensajes, en lugar de enviarlos directamente a una API de ingestión. En cada nodo de índice/búsqueda, había un proceso de trabajo ligero que extraía los mensajes de la cola. El proceso de trabajo contenía la ruta al paquete de registros en el almacenamiento de objetos, como se muestra en la Figura 10-2.

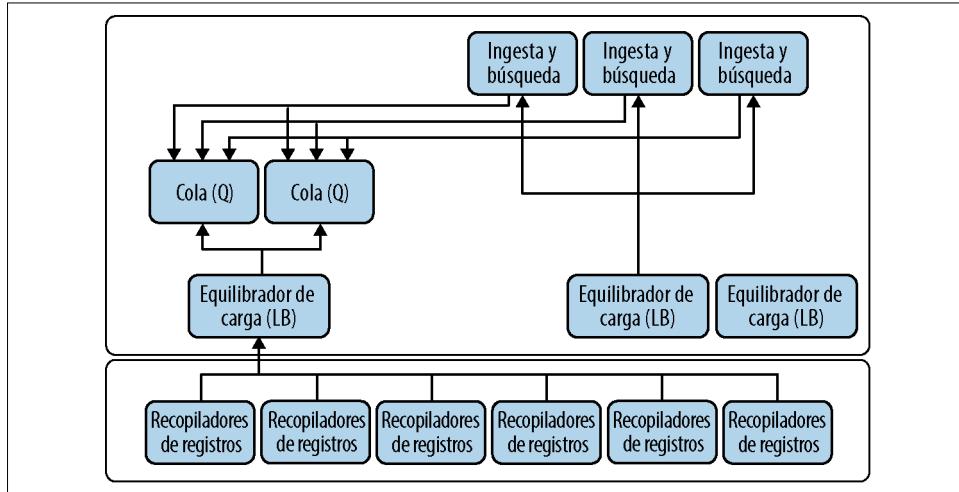


Figura 10-2. Generación 2, con colas de la nube y almacenamiento de objetos

Ambas soluciones tenían una arquitectura de aplicaciones monolítica, que se abordó en la tercera generación de la plataforma. Esta generación también trasladaba la lógica de análisis a la nube y reducía la sobrecarga operativa de los equipos de clientes locales.

## Generación 3: Con cola Pub/Sub basada en la memoria

La tercera generación de la plataforma implementó una arquitectura de microservicios y colas basada en memoria de alto rendimiento, que permitía a los servicios individuales procesar mensajes para proporcionar análisis avanzados mediante una base de datos de documentos, en lugar del motor de búsqueda respaldado por RDBMS anterior. Esto separaba los dominios de error y rendimiento, y permitía el escalado horizontal del sistema, como se muestra en la [Figura 10-3](#).

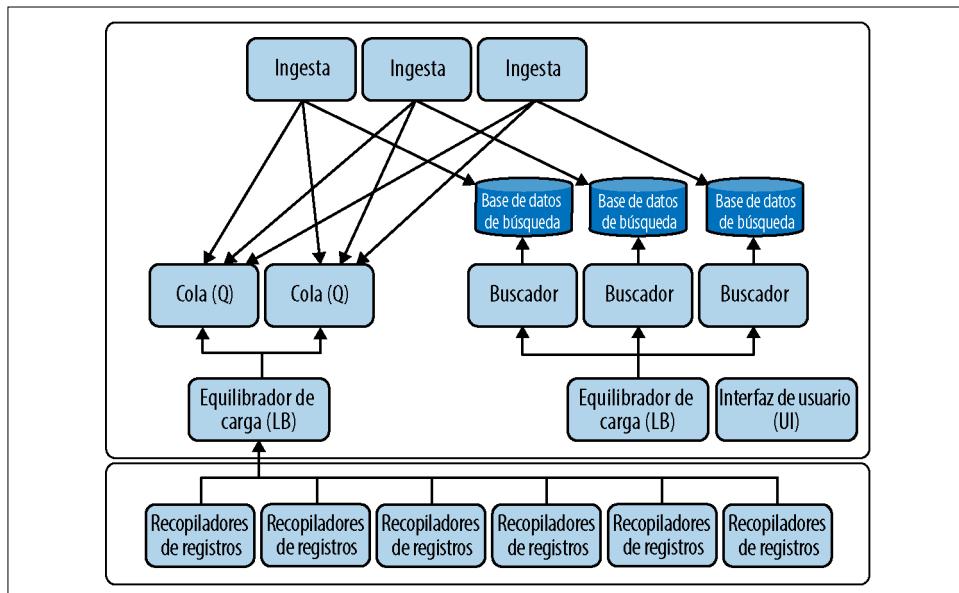


Figura 10-3. Generación 3, con arquitectura de microservicios y colas escaladas horizontalmente

La adopción de sistemas de mensajería puede ser un componente clave para ayudar a escalar un producto. En el ejemplo anterior, el sistema de mensajería se usó para escalar el procesamiento de eventos. Sin embargo, los sistemas de mensajería se pueden usar para otros fines, como el transporte de análisis, registros o cualquier otro dato con formato sobre el que no se tenga que aplicar una acción en tiempo real.

# Conceptos básicos de las plataformas de mensajería

La necesidad de componentes de software (ya sean aplicaciones cliente/servidor o programas individuales que se ejecuten en un solo equipo) es anterior a la nube e incluso a Internet. Todos menos los sistemas operativos integrados y de propósito general más primitivos proporcionan capacidades para que los programas envíen datos a otros programas y los reciban.

## La mensajería en comparación con el streaming

El panorama de Cloud Native Computing Foundation (CNCF) (que se muestra en la [Figura 10-4](#)) nombra a la categoría de mensajes *Streaming y mensajería*, pero en este capítulo nos centraremos principalmente en la mensajería. Kafka, NATS y RabbitMQ proporcionan capacidades de streaming, pero a diferencia de las plataformas de análisis de streaming, como Spark y Flink, estas no están habilitadas de forma predeterminada y no fueron los casos de uso inicial de ninguna de estas plataformas de mensajería. Analizaremos los conceptos de mensajería en detalle en la siguiente sección, pero es importante tener en cuenta que una consideración clave para los sistemas de streaming es que las operaciones se producen en varios mensajes, a menudo, con una ventana de tiempo o evento ordenado dentro de las operaciones de memoria para incorporaciones o filtrado.

## Fundamentos de la mensajería

Comprender las similitudes y diferencias entre las soluciones de mensajería contemporánea puede ayudarlo a determinar qué herramienta es la mejor para el trabajo o el problema en cuestión. Como hemos visto a lo largo de este libro, siempre hay más de una manera de crear algo, y no siempre es una decisión fácil elegir el camino correcto hasta que las capacidades anunciadas de la herramienta se prueban y validan con su infraestructura y caso de uso específicos. Antes de explorar cómo RabbitMQ, Apache Kafka y NATS se ajustan en la infraestructura nativa de la nube y las aplicaciones, debemos entender los conceptos básicos y las capacidades comunes en los sistemas de mensajería, cola y streaming.

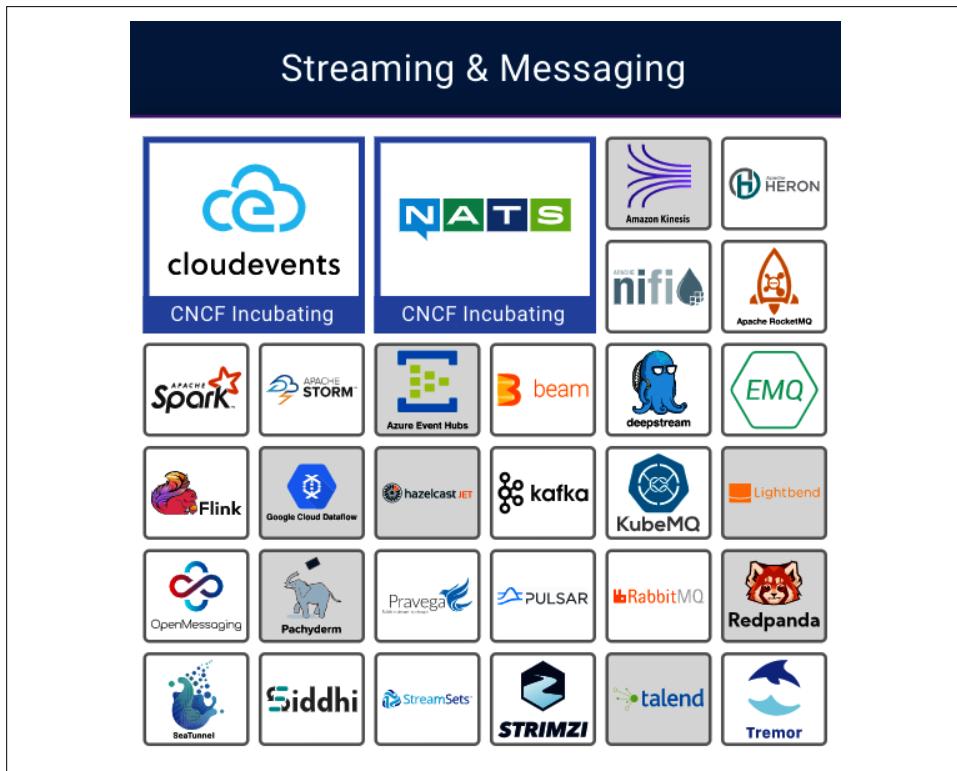


Figura 10-4. El panorama de streaming y mensajes de CNCF

## Productores y consumidores

Al igual que cuando se comunica con otro ser humano, un mensaje siempre tiene un emisor y un receptor, pero el mensaje puede o no enviarse directamente y se puede o no recibir una respuesta. Cuando envía un correo electrónico, hay varias partes intermedias (consulte la Figura 10-5). Estas partes incluyen su cliente de correo electrónico, los servidores de su proveedor de correo electrónico (agentes de transferencia de mensajes [MTAs], que desempeñan un rol de intermediario), el servidor del receptor (otro MTA) y un cliente de correo electrónico receptor. En este ejemplo, hay una serie de consumidores y productores: el cliente de correo electrónico pone en cola los mensajes para enviarlos y producirlos a los servidores de correo del ISP y, luego, el MTA pone en cola los mensajes enviados y los envía/produce al MTA del receptor. El cliente de correo electrónico del receptor recibirá/consumirá los mensajes del MTA de su ISP.

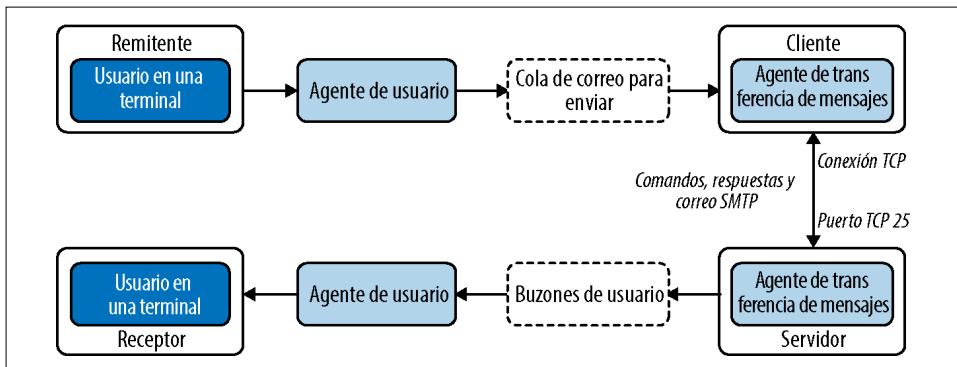


Figura 10-5. El protocolo de correo SMTP, un ejemplo de un sistema de formación de colas

Al igual que en el caso de las redes donde hay patrones de entrega de unidifusión, multidifusión y difusión, en la mensajería puede haber casos en los que un solo consumidor procesa los datos de un solo productor o de varios consumidores (como un correo electrónico). Los mensajes se pueden enviar o ser extraídos por los consumidores en función de la plataforma y la configuración.

## Agentes y clústeres

En nuestro ejemplo anterior de correo electrónico, los componentes intermedios, los MTAs, actúan como agentes entre el emisor y el receptor. En muchos casos, el software MTA se puede agrupar en clústeres. Uno o más agentes en un clúster permiten que los emisores y los receptores se desacoplen entre sí. RabbitMQ, Kafka y NATS, las implementaciones de mensajería que se analizan en este capítulo requieren al menos un agente y, siguiendo el principio de la mayoría de los protocolos de sistema distribuido, a menudo, requieren un mínimo de tres agentes y, por lo general, deben ser recuentos impares (para los algoritmos de coherencia). Las consideraciones para la agrupación en clústeres incluyen la replicación de estado, las elecciones de líderes, el comportamiento de pertenencia a nodos y la partición de clústeres en caso de error de red (y el temido problema de cerebro dividido).

Otras consideraciones para los clústeres (que pueden convertirse rápidamente en un cuello de botella) son cómo se pueden escalar sin tiempo de inactividad. La sobrecarga operativa de mantener clústeres de mensajería de alto rendimiento y baja latencia es la razón por la que los equipos optan por que el proveedor de nube administre la infraestructura, de manera que los desarrolladores puedan centrarse en el desarrollo y la optimización de aplicaciones de productores y consumidores.

## Durabilidad y persistencia

Cualquier persona que haya enviado un correo electrónico a una dirección inexistente ha recibido mensajes de error de un servidor de correo que indica que después de un cierto período de tiempo el mensaje ha expirado. La implicación es que el mensaje se almacena en un sistema externo después de que sale de su cliente de correo.

Según el caso de uso de mensajería y el tipo de datos que se almacenan y procesan, pueden tener más o menos criticidad, tener una mayor o menor imperdurabilidad y ser más o menos críticos que cada mensaje (o transacción) almacenada antes del procesamiento. Al igual que en el caso de las aplicaciones cliente/servidor del Protocolo de control de transmisión (TCP) y el Protocolo de datagramas de usuario (UDP), es importante tener en cuenta la cantidad de administración de errores y la lógica de reintento implementadas en la capa de la aplicación en comparación con la capa de transporte subyacente cuando se crea un sistema de producción. Los sistemas de mensajería distribuyen inteligencia a los productores, consumidores y agentes de diferentes maneras.

Además, los mecanismos utilizados para almacenar los mensajes varían de un sistema a otro. Algunos sistemas tienen almacenamiento puramente basado en la memoria, lo que los hace menos duraderos, en especial, en entornos de nube, y otros tienen soluciones respaldadas por disco que, a la vez que tienen desafíos en los entornos de nube, proporcionan una experiencia más duradera y confiable.

## Entrega de mensajes

Además del origen y el destino, hay tres patrones importantes que debe tener en cuenta para las plataformas orientadas a mensajes, que cubriremos en este capítulo:

### *Al menos una vez*

Implica que podría haber duplicados; en particular, si un consumidor se bloqueó y no notificó al agente de que se recibió el mensaje, o que un mensaje se puede entregar a varios consumidores

### *Como máximo una vez*

Supone que un consumidor puede enviar un mensaje, pero no lo recibe ni lo reconoce, lo que pone la carga de la redistribución en el cliente

### *Exactamente una vez*

Una situación muy limitada en la que se requieren las transacciones y los mensajes deben entregarse exactamente una vez

La arquitectura de mensajería puede admitir uno o más de estos patrones. Estos también pueden configurarse en el agente o en las bibliotecas cliente del consumidor y el productor

## Seguridad

La confidencialidad de los datos contenidos en el tema de su mensaje puede variar, por lo que es necesario tener controles flexibles sobre quién puede producir y consumir estos temas. Un sistema de mensajes debe verificar la identidad del productor y del consumidor y, a continuación, aplicar permisos de lectura/escritura al tema. Además, los datos que conserva un sistema deben cifrarse en reposo.

Ahora que comprende algunos de los fundamentos de la mensajería, examinemos algunos patrones de mensajería comunes con los que puede encontrarse.

## Patrones de mensajería comunes

Analizaremos tres arquitecturas de mensajería comunes que verá en la infraestructura moderna.

### Cola simple

Los productores (P) publican en una cola (B) y, a medida que los consumidores (C) leen la cola, los mensajes se eliminan y ningún otro consumidor procesará el mensaje.

### Publicar y suscribir

Como se muestra en la Figura 10-6, un productor (P) envía mensajes, y todos los consumidores (C) que escuchan (suscritos) reciben los mensajes. Puede o no tener confirmación, pero si los consumidores no están en línea, perderán el mensaje.

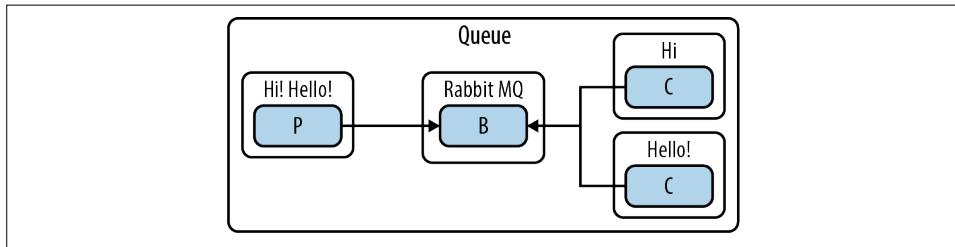


Figura 10-6. Un sistema de mensajes pub/sub simple

### Cola duradera

Como se muestra en la Figura 10-7, un productor (P) envía mensajes al agente (B) y (C) consume de forma independiente procesamiento de mensajes en una lista ordenada basada en un puntero, pero puede retroceder a un punto conocido de la lista, como el primer mensaje dentro de una ventana concreta.

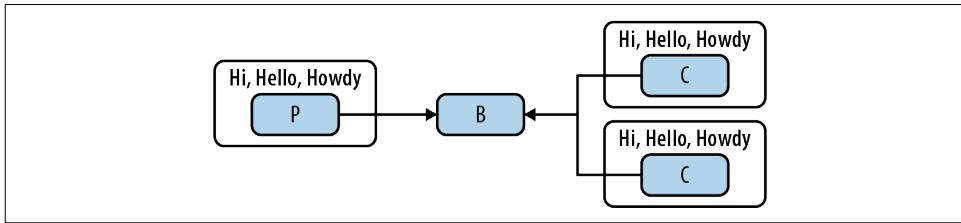


Figura 10-7. Una cola de mensajes ordenada duradera

Hasta ahora, hemos cubierto gran parte del aspecto teórico en torno a la arquitectura y los conceptos básicos de la mensajería. Ahora profundizaremos en las diversas plataformas de mensajería nativas de la nube.

## Descripción general de las plataformas populares de mensajería nativa de la nube

En esta sección, revisaremos una serie de plataformas de mensajería nativas de la nube populares que se han convertido en algo común en la infraestructura moderna.

### RabbitMQ

Aunque RabbitMQ existe hace más de 15 años y, obviamente, es anterior a la adopción pública generalizada de la nube y los microservicios, es un sistema de mensajería comprobado que ha potenciado cargas de trabajo críticas y ha seguido siendo relevante en el mundo de los contenedores y los clústeres de Kubernetes. Una de las razones de esto es que RabbitMQ se desarrolló en Erlang/OTP, un lenguaje de programación funcional que está diseñado para la escalabilidad, la simultaneidad y la confiabilidad, por lo que es ideal para la creación de sistemas distribuidos que procesan de millones a cientos de millones de mensajes al día con un tiempo de inactividad muy bajo.

### Apache Kafka

En 2010, los ingenieros de LinkedIn descubrieron que RabbitMQ no podía escalar a la naturaleza y el tamaño complejos de su ecosistema. Kafka se creó para permitir que la mensajería se convirtiera en el núcleo de la pila tecnológica de LinkedIn y se ajustó con éxito a billones de mensajes al día en todo el sitio.

Más adelante en este capítulo, analizaremos Azure Event Hubs, un servicio que es compatible con Apache Kafka y permite que las aplicaciones simplemente actualicen la cadena de conexión de consumidor/productor, pero es posible que no hagan que todas las características de Kafka estén disponibles. Kafka administrado también forma parte de [HDInsight](#), un amplio conjunto de servicios de análisis en la nube en Azure. HDInsight está fuera del ámbito de este libro.

## CNCF CloudEvents

**Cloud Events** se inició por primera vez en 2017 dentro del CNCF Serverless Working Group. Según las propias palabras de CNCF: “CloudEvents es una especificación para describir los datos de eventos en formatos comunes a fin de proporcionar interoperabilidad entre servicios, plataformas y sistemas”. Aunque este proyecto tiene un gran potencial (y es compatible con Azure Event Grids), no lo abordaremos con más detalle en este libro.

## Análisis de la mensajería en la nube con NATS

**NATS** es otra implementación de mensajería de open source que ha sido aceptada por CNCF. NATS se lanzó por primera vez en 2011 y se implementó inicialmente en Ruby como la capa de mensajería y detección de servicios para Cloud Foundry. Más tarde, se reescribió en Golang y, en la actualidad, se encuentra bajo un desarrollo muy activo. Aunque no es tan popular como RabbitMQ o Kafka, NATS ofrece varias opciones que se pueden implementar fácilmente debido a que su único servidor binario está disponible en varios sistemas operativos, arquitectura de procesador y repositorios oficiales de Docker. Las bibliotecas cliente están disponibles para todos los lenguajes de programación populares. La simplicidad, la operabilidad, el rendimiento y la seguridad se citan como las ventajas de NATS sobre otras implementaciones de mensajería, al igual que la capacidad de usar fácilmente un conjunto de características mínimo con una preferencia por la entrega “como máximo una vez”.

Dado que el protocolo pub/sub base no implementa la persistencia del mensaje, los consumidores deben estar activos para recibir el mensaje, lo que hace que NATS sea óptima para el intercambio de datos que son altamente perecederos y los datos que se actualizan con frecuencia, como los datos del sensor. De forma predeterminada, los mensajes se enrutarán a todos los suscriptores que escuchan ese tema, de manera similar a las redes de multidifusión.

Debido a su baja superficie de memoria y rendimiento extremadamente alto, NATS también es ideal para casos de uso de análisis de perímetro e IoT. Para una implementación de mensajería tan ligera que se centra en la simplicidad, NATS tiene capacidades de autenticación y autorización relativamente sofisticadas y admite el uso de varios inquilinos de forma predeterminada en las cuentas. NATS Streams y NATS JetStream admiten tanto la memoria como la persistencia basada en disco. Pub/sub y streaming de NATS son compatibles desde el principio con la alta disponibilidad y la escalabilidad horizontal, y JetStream las ha implementado desde principios de 2021.

## Arquitectura del protocolo NATS

Al igual que RabbitMQ, NATS ofrece varios patrones de mensajería y sus capacidades han seguido evolucionando desde la implementación inicial de pub/sub, streaming de NATS (que se muestra en la [Figura 10-8](#)), hasta su subsistema de almacenamiento más reciente, JetStream. Lanzado en 2020, JetStream ahora está disponible en NATS 2.2.x. NATS ahora también incluye compatibilidad con el protocolo MQTT nativo.

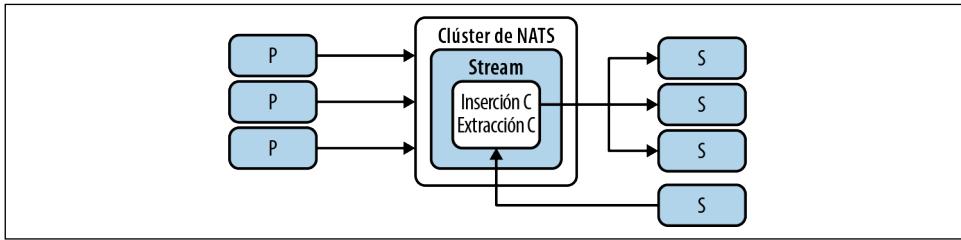


Figura 10-8. Arquitectura básica de NATS

Debido a que la arquitectura fundamental es publicación/suscripción, los productores de NATS se denominan editores y sus consumidores, suscriptores. NATS JetStream utiliza la abstracción interna del consumidor.

El intercambio más sencillo posible entre un publicador y un suscriptor mediante la CLI de NATS es la mejor manera de ver lo sencillo que es el protocolo.

Veamos un ejemplo. En el siguiente código, un productor envía el mensaje "arf" con el asunto `dog.food`:

```
$ echo "arf" | nats pub dog.food
16:10:43 Reading payload from STDIN
16:10:43 Published 4 bytes to "dog.food"
```

En el siguiente código, un consumidor se suscribe al servicio para todos los mensajes que comienzan con `dog..`:

```
$ nats sub dog.*
16:09:49 Subscribing on dog.*
[#1] Received on "dog.food"
```

### Agrupación en clústeres de varios servidores NATS

NATS permite agrupar varios servidores para proporcionar alta disponibilidad y equilibrio de carga.

El clúster más simple se puede iniciar haciendo lo siguiente:

- Configurar una URL de clúster con el argumento `-cluster` al iniciar el servidor
- Especificar rutas en los nodos de servidor adicionales

Si suponemos que contamos con dos servidores Linux distintos en 192.168.2.238 y 192.168.2.247, podemos configurar un clúster con los siguientes comandos:

```
nats-server -DV -cluster nats://192.168.2.238:4248
nats-server -DV -cluster nats://192.168.2.247:4248 -routes nats://192.168.2.238:4248
```

## **Uso del servidor de Docker NATS**

Además de RPM y deb, Synadia distribuye y mantiene imágenes de Docker que son probablemente la forma más fácil de ejecutar un servidor de NATS, como se muestra en el Ejemplo 10-1.

#### *Ejemplo 10-1. Ejecución de NATS con Docker*

Podemos pasar todos los argumentos de la línea de comandos. En este caso, estamos especificando un volumen que se montará en nuestro directorio `~/tmp` que se utilizará para la persistencia de JetStream.

## Supervisión de servidores NATS

NATS expone un punto de conexión de supervisión que se puede ver a través del explorador, la línea de comandos o un navegador web, como se muestra en la [Figura 10-9](#).

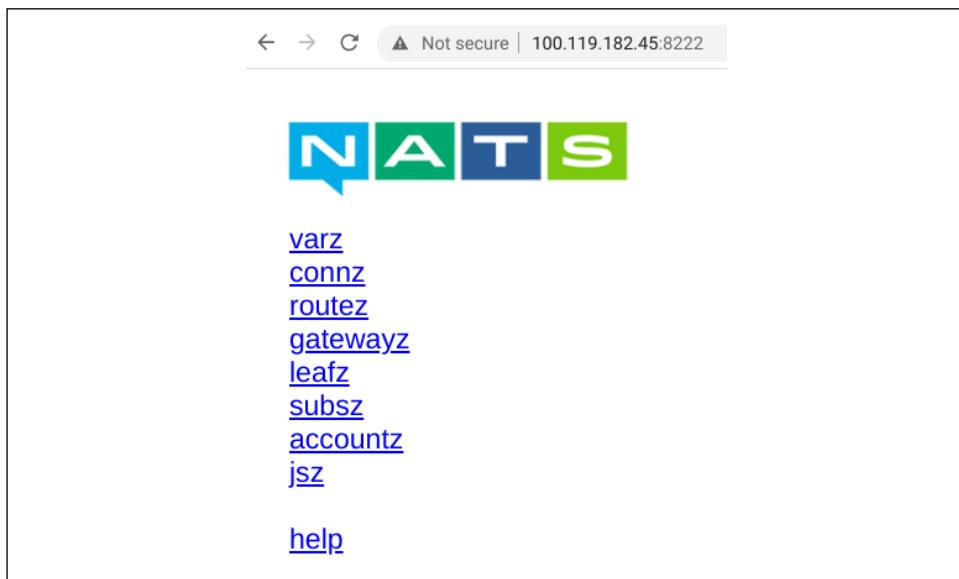


Figura 10-9. Punto de conexión de supervisión de NATS

El siguiente es un ejemplo del resultado del punto de conexión de supervisión de NATS:

```
$ curl http://100.119.182.45:8222/varz
{
  "server_id": "ND6FZROVHED32BDBTMCPXEN4LIK82R0R3H03GZ6DLDB64Z54SLG5GOD",
  "server_name": "ND6FZROVHED32BDBTMCPXEN4LIK82R0R3H03GZ6DLDB64Z54SLG5GOD",
  "version": "2.2.2",
  "proto": 1,
  "git_commit": "a5f3aab",
  "go": "go1.16.3",
  "host": "0.0.0.0",
  "port": 4222,
  "connect_urls": [
    "192.168.2.247:4222",
    "192.168.122.1:4222",
    "172.17.0.1:4222",
    "100.119.182.45:4222",
    "[fd7a:115c:a1e0:ab12:4843:cd96:6277:b62d]:4222",
    "192.168.122.1:4222",
    "172.17.0.1:4222",
    "100.119.135.53:4222",
    "[fd7a:115c:a1e0:ab12:4843:cd96:6277:8735]:4222",
    "192.168.2.238:4222"
  ],
  "max_connections": 65536,
  "ping_interval": 120000000000,
  "ping_max": 2,
  "http_host": "0.0.0.0",
  "http_port": 8222,
  "http_base_path": "",
  "https_port": 0,
  "auth_timeout": 2,
```

```

"max_control_line": 4096,
"max_payload": 1048576,
"max_pending": 67108864,
"cluster": [
    {
        "name": "xCQWc0Mb0GCpM7ESUuctIc",
        "addr": "192.168.2.247",
        "cluster_port": 4248,
        "auth_timeout": 2,
        "urls": [
            "192.168.2.238:4248"
        ],
        "tls_timeout": 2
    },
    "gateway": {},
    "leaf": {},
    "jetstream": {},
    "tls_timeout": 2,
    "write_deadline": 100000000000,
    "start": "2021-05-03T15:13:05.512147331Z",
    "now": "2021-05-03T15:41:53.509877951Z",
    "uptime": "28m47s",
    "mem": 12353536,
    "cores": 6,
    "gomaxprocs": 6,
    "cpu": 0,
    "connections": 1,
    "total_connections": 1,
    "routes": 1,
    "remotes": 1,
    "leafnodes": 0,
    "in_msgs": 117,
    "out_msgs": 115,
    "in_bytes": 67297,
    "out_bytes": 66346,
    "slow_consumers": 0,
    "subscriptions": 144,
    "http_req_stats": {
        "/": 3,
        "/accountz": 1,
        "/connz": 1,
        "/gatewayz": 0,
        "/leafz": 1,
        "/routez": 0,
        "/subsz": 1,
        "/varz": 2
    },
    "config_load_time": "2021-05-03T15:13:05.512147331Z",
    "system_account": "$SYS"
}

```

Puede usar una herramienta como Telegraf para extraer estos puntos de conexión HTTP y enviarlos a un back-end de métricas ascendentes o al [exportador Prometheus](#).

## Persistencia de NATS con JetStream

Aunque NATS introdujo una implementación de persistencia llamada NATS Streaming en 2019, quedará obsoleta en junio de 2023. La implementación de segunda generación, llamada JetStream, ha estado en desarrollo desde el año 2020 y será la opción preferida para los casos de uso futuros. Aunque JetStream tiene un repositorio de GitHub separado, ahora está integrado en el servidor de NATS 2.2. x y es compatible con la CLI de NATS, que analizaremos en breve. Pub/sub de NATS solo proporcionaba la entrega “como máximo una vez”. JetStream permite la entrega “exactamente una vez” y se parece más al comportamiento de las colas de RabbitMQ y los temas de Kafka (consulte la Figura 10-10).

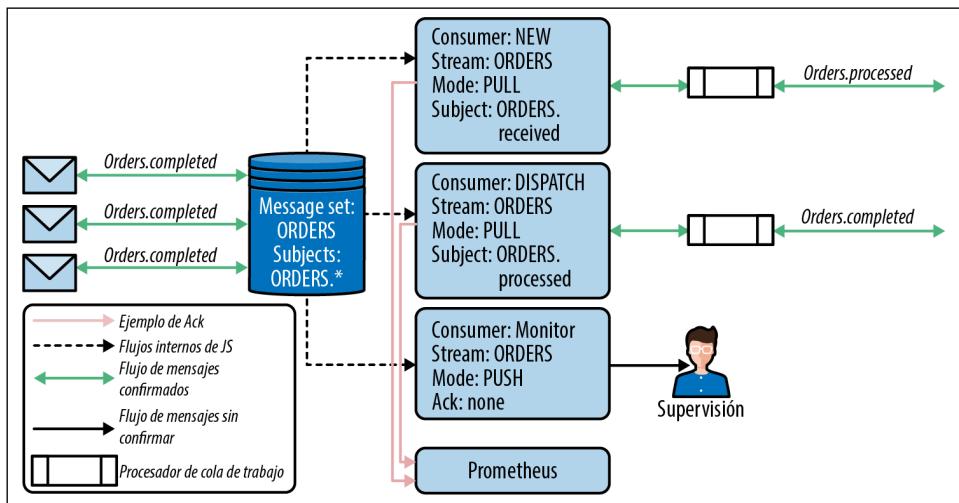


Figura 10-10. La arquitectura de procesamiento de JetStream

## Seguridad de NATS

Podría escribirse fácilmente un capítulo completo sobre cómo configurar la gama de capacidades de autenticación, autorización y cifrado de NATS 2.3.x. En esta sección, trataremos los pasos básicos necesarios para elevar la exigencia de los valores predeterminados no seguros que hemos estado usando en la mayoría de nuestros ejemplos hasta ahora, que son inadecuados para las implementaciones de producción. Una sólida infraestructura de administración de identidades y secretos es la base de los sistemas de nube seguros, pero está fuera del ámbito de este libro. NATS proporciona un subsistema de permisos sólido, que permite que el control granular se defina sobre una **base de temas**, y utiliza almacenes de identidad distribuidos.

## Autenticación basada en Nkey

Aunque NATS todavía admite la autenticación de token y nombre de usuario/contraseña, se recomienda la autenticación Nkey en el futuro. Utiliza criptografía asimétrica y no requiere que una clave compartida se almacene en el servidor en formato de texto sin formato o con hash. La administración de claves es extremadamente sencilla.

Para generar los pares de claves, use la utilidad `nk` que se encuentra dentro del repositorio de nkeys:

```
$ nk -gen user -pubout  
SUAGWCAMMXKR43EDIXVC5FEA5G3767ALGQR75N27NGPK37KZUURS7F32FE  
UBFHQJERKEBE323BXQEV6257CIRJ4CIC5LGT6R2LJ524GQ TUQHXCUK3
```

La clave pública (que comienza con una “U”) se configura en el servidor:

```
$ cat server.conf  
net: 0.0.0.0  
port: 4242  
  
authorization {  
    users: [  
        { nkey: UBFHQJERKEBE323BXQEV6257CIRJ4CIC5LGT6R2LJ524GQ TUQHXCUK3 }  
    ]  
}
```

La clave privada (o inicialización de clave (seed key), eso es lo que significa el prefijo “S”) se almacena en el disco y se pasa como argumento al cliente de NATS o como una variable en el código. Al igual que con cualquier material confidencial, es mejor si estas claves se almacenan en un almacén de secretos externos, como Azure Key Vault o como secreto de Kubernetes:

```
$ cat client.nkey  
SUAGWCAMMXKR43EDIXVC5FEA5G3767ALGQR75N27NGPK37KZUURS7F32FE  
  
$ nats -s nats://127.0.0.1:4242 --nkey client.nkey pub "foo"  
12:57:17 Published 4 bytes to "foo"
```

## Autenticación TLS

Aunque tenemos autenticación segura (y, posiblemente, autorización, en función de cómo hayamos configurado los permisos del sujeto), el uso de Nkey por sí solo no es suficiente para la seguridad de la red, puesto que NATS es, de forma predeterminada, un protocolo de texto sin formato ASCII.

Al igual que la mayoría de los protocolos nativos de la nube, NATS admite el cifrado TLS mutuo. En general, el mayor desafío con TLS es la administración de certificados y claves, que está fuera del ámbito de este libro. Para fines de demostración, `mkcert` proporciona una ruta más fácil que OpenSSL para generar los pares de claves de entidad de certificación (CA), cliente y servidor necesarios para el cifrado de extremo a extremo.

En primer lugar, configuramos la CA y creamos certificados de cliente y servidor dentro del directorio `~/nats`:

```
$ mkdir ~/nats && cd ~/nats

# Create the CA
$ mkcert -install

# Create the server
$ mkcert -cert-file server-cert.pem -key-file server-key.pem localhost 127.0.0.1 ::1

# Generate a certificate for client authentication.
$ mkcert -client -cert-file client-cert.pem -key-file client-key.pem localhost \
::1 127.0.0.1 email@localhost
$ cp -av `mkcert -CAROOT`/* ~/nats
```

Luego, iniciaremos el servidor (dentro del directorio `~/nats`) y agregaremos los parámetros del certificado TLS:

```
$ nats-server -DV --tls --tlscert=server-cert.pem --tlskey=server-key.pem -ms 8222
```

Por último, nos conectamos con el cliente:

```
$ nats --tlscert=client-cert.pem --tlskey=client-key.pem account info
```

En el servidor, deberíamos ver una conexión TLS exitosa:

```
[10108] 2021/06/28 14:46:40.126469 [DBG] 127.0.0.1:50192 - cid:11 - Client connection created
[10108] 2021/06/28 14:46:40.126573 [DBG] 127.0.0.1:50192 - cid:11 - Starting TLS client connection handshake
[10108] 2021/06/28 14:46:40.157446 [DBG] 127.0.0.1:50192 - cid:11 - TLS handshake complete
[10108] 2021/06/28 14:46:40.157468 [DBG] 127.0.0.1:50192 - cid:11 - TLS version 1.3, cipher suite TLS_AES_128_GCM_SHA256
[10108] 2021/06/28 14:46:40.157584 [TRC] 127.0.0.1:50192 - cid:11 - <- [CONNECT
{"verbose":false,"pedantic":false,"tls_required":true,"name":"NATS CLI Version 0.0.23",
"lang":"go","version":"1.11.0","protocol":1,"echo":true,"headers":true,"no_responders":true}]
```

## Implementación de NATS en Kubernetes

NATS se puede instalar rápidamente con un gráfico de Helm. El gráfico de Helm también instalará un operador Prometheus y un administrador de certificados que le permitirán aprovechar al máximo las métricas y la compatibilidad con TLS que vienen con NATS:

1. Empiece por agregar el repositorio de Helm:

```
$ helm repo add nats https://nats-io.github.io/k8s/helm/charts/
$ helm repo update
```

2. Luego, instale el servidor NATS:

```
$ helm install my-nats nats/nats
```

3. Instale el servidor de streaming de NATS (que se conoce como stan):

```
$ helm install my-stan nats/stan --set stan.nats.url=nats://my-nats:4222
```

4. Después de estos pasos, aparecerá Grafana Surveyor que puede ver mediante el enrutamiento de puerto al pod:

```
kubectl port-forward deployments/nats-surveyor-grafana 3000:3000
```

5. Podrá ver Grafana en su explorador en <http://127.0.0.1:3000/d/nats/nats-surveyor?refresh=5s&orgId=1>.

Su implementación de ayuda ofrece una serie de configurables de Helm. Puede leer más sobre esto en la [página de configuración de la implementación de Helm en NATS](#).

Puede codificar para NATS de forma muy sencilla en Python con el paquete `nats-py`. En el siguiente ejemplo, codificamos un productor básico y un consumidor en Python. NATS proporciona un servidor público para que pueda realizar pruebas. Si desea hacer pruebas en su propio entorno, reemplace `nc = await nats.connect ("nats://demo.nats.io:4222")` con la dirección de su instancia de NATS:

```
import time

import asyncio
import nats
from nats.aio.errors import ErrConnectionClosed, ErrTimeout, ErrNoServers

async def run():
    # The Nats project provides a public demo server `nats://demo.nats.io:4222`.
    # You can add your own Nats server on the next line
    nc = await nats.connect("nats://demo.nats.io:4222")

    async def message_handler(msg):
        subject = msg.subject
        reply = msg.reply
        data = msg.data.decode()
        print("Received a message on '{subject} {reply}': {data}".format(
            subject=subject, reply=reply, data=data))

    # Simple publisher and async subscriber via coroutine.
    sub = await nc.subscribe("foo", cb=message_handler)

    await nc.publish("foo", b'Message1')
    await nc.publish("foo", b'Message2')

    time.sleep(5)
    # Remove interest in subscription
    await sub.unsubscribe()

    # Terminate connection to NATS.
    await nc.drain()

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(run())
    loop.close()
```

# Servicios de mensajería de Azure

Cuando las ofertas de la nube pública comercial estuvieron disponibles por primera vez a mediados de la década de 2000, los servicios de almacenamiento y mensajería (como AWS S3 y AWS SQS) estuvieron entre los primeros en estar disponibles para los clientes. Microsoft Azure ofrece varios servicios de mensajería administrados para admitir implementaciones específicas de la nube, así como protocolos de mensajería basados en estándares, como [AMQP](#) y [JMS](#) y patrones como la cola de trabajos y pub/sub que ya exploramos en este capítulo. Todas las opciones de Azure proporcionan la entrega “al menos una vez”.

Cuando elija entre Azure Service Bus, Azure Event Hubs y Azure Event Grid, debe tener en cuenta los siguientes criterios:

- Compatibilidad con plataformas y protocolos de transferencia de mensajes locales (ActiveMQ, RabbitMQ) y herramientas
- Nivel de integración que se requiere con otros servicios administrados de Azure
- Necesidad de mensajes y transacciones ordenados
- Capacidad de manejar el streaming de eventos
- Uso de canalizaciones de macrodatos
- Tasa máxima de ingesta (eventos por segundo)
- Patrones de eventos asincrónicos disponibles
- Precios

Uno de los principales beneficios de usar los servicios de mensajería de Azure es la capacidad de aprovechar la administración de secretos y las funcionalidades de IAM de Azure. Ahora analicemos en detalle las ofertas de Azure.

## Azure Service Bus

Azure Service Bus es un servicio de cola administrado, que proporciona características de nivel empresarial comparables a las de ActiveMQ y RabbitMQ, pero sin la sobrecarga de administración de varios agentes en máquinas virtuales o implementaciones sobre Kubernetes. Azure Service Bus puede abarcar varias zonas de disponibilidad con [Azure Service Bus Premium](#) y está diseñado para admitir la migración de implementaciones de JMS 2.0 locales.

Service Bus es compatible con la cola de mensajes simple y los patrones de publicación/suscripción que analizamos al principio del capítulo. El patrón más común es el “modelo de extracción”, en el que los consumidores sondean el punto de conexión de servicio para los mensajes nuevos. Al igual que con RabbitMQ y Kafka, varios consumidores pueden procesar los mensajes lo más rápido posible cuando los mensajes están disponibles, lo que permite que el trabajo se distribuya entre varios nodos.

trabajadores. Azure Service Bus implementa **AMQP 1.0** y tiene SDK bien documentados y muestras de código para .NET, Java, Python, Go y más.

## Conceptos de Service Bus

Las abstracciones de Azure Service Bus (consulte las figuras 10-11 y 10-12 para ver las arquitecturas de referencia) proporcionan capacidades similares a aquellas disponibles en NATS.

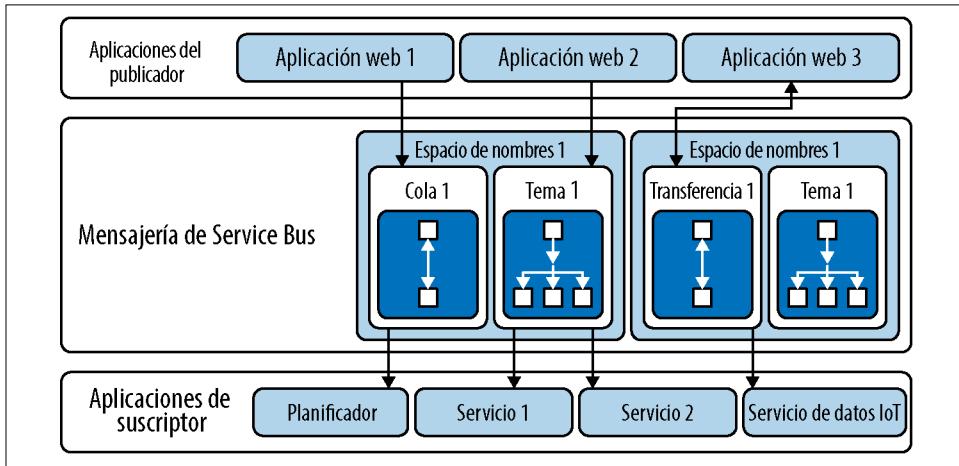


Figura 10-11. Arquitectura de Service Bus de referencia

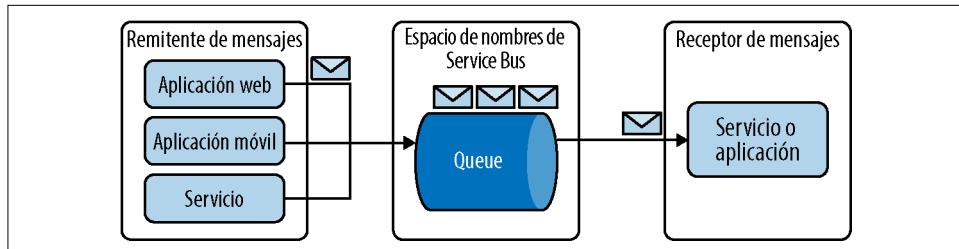


Figura 10-12. Arquitectura de mensajería simple

**Espacios de nombres.** Los espacios de nombres permiten segmentar los temas, las colas y los servicios como un contenedor para los componentes de mensajería, al igual que una red virtual suministran un límite para los componentes de red, como máquinas virtuales y equilibradores de carga. Kafka no proporciona una abstracción comparable, pero si lo hacen los hosts virtuales de RabbitMQ y las cuentas de NATS.

**Colas, temas y suscripciones.** Las colas de Azure Service Bus proporcionan un conjunto ordenado de mensajes FIFO que se extraen de forma similar a los consumidores de Kafka y RabbitMQ o a los consumidores basados en extracción de NATS, mientras que los temas son principalmente para patrones de pub/sub y comunicación de uno a varios

para mensajería con una gran cantidad de consumidores. Las suscripciones permiten a los consumidores recibir mensajes basados en reglas de filtrado. Al igual que con NATS JetStream, los mensajes se recuperan a partir de la suscripción (al igual que el consumidor de JetStream) y no del tema (comparable a la transmisión de NATS JetStream).

## Administración de Azure Service Bus con Terraform

Si bien Microsoft proporciona un SDK de Python para la administración de Service Bus, nos quedaremos con Terraform para la creación de espacios de nombres, colas y temas de Service Bus. En el siguiente ejemplo de código, se implementan nuevos recursos `servicebus`, `namespace`, `queue` y `topic`:

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "example" {
  name      = "terraform-servicebus"
  location = "Central US"
}

resource "azurerm_servicebus_namespace" "example" {
  name          = "mdfranz-servicebus-namespace"
  location      = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  sku           = "Standard"

  tags = {
    source = "terraform"
  }
}

resource "azurerm_servicebus_queue" "example" {
  name          = "mdfranz-servicebus-queue"
  resource_group_name = azurerm_resource_group.example.name
  namespace_name   = azurerm_servicebus_namespace.example.name
  enable_partitioning = true
}

resource "azurerm_servicebus_topic" "example" {
  name          = "mdfranz-servicebus-topic"
  resource_group_name = azurerm_resource_group.example.name
  namespace_name   = azurerm_servicebus_namespace.example.name
  enable_partitioning = true
}
```

Con `azure-cli`, podemos ver los recursos que se crearon:

```
$ az servicebus queue list --namespace-name mdfranz-servicebus-namespace -g \
  terraform-servicebus
[
  {
    "accessedAt": "0001-01-01T00:00:00+00:00",
    "autoDeleteOnIdle": "10675199 days, 2:48:05.477581",
    "countDetails": {
      "activeMessageCount": 0,
```

```

    "deadLetterMessageCount": 0,
    "scheduledMessageCount": 0,
    "transferDeadLetterMessageCount": 0,
    "transferMessageCount": 0
  },
  "createdAt": "2021-04-14T02:24:28.260000+00:00",
  "deadLetteringOnMessageExpiration": false,
  "defaultMessageTimeToLive": "10675199 days, 2:48:05.477581",
  "duplicateDetectionHistoryTimeWindow": "0:10:00",
  "enableBatchedOperations": true,
  "enableExpress": false,
  "enablePartitioning": true,
  "forwardDeadLetterredMessagesTo": null,
  "forwardTo": null,
  "id": "/subscriptions/1bf91ee3-5b21-4996-bd72-ae38e8f26ce9/resourceGroups/ \
    terraform-servicebus/providers/Microsoft.ServiceBus/namespaces/ \
      mdfranz-servicebus-namespace/queues/mdfranz-servicebus-queue",
  "location": "Central US",
  "lockDuration": "0:01:00",
  "maxDeliveryCount": 10,
  "maxSizeInMegabytes": 81920,
  "messageCount": 0,
  "name": "mdfranz-servicebus-queue",
  "requiresDuplicateDetection": false,
  "requiresSession": false,
  "resourceGroup": "terraform-servicebus",
  "sizeInBytes": 0,
  "status": "Active",
  "type": "Microsoft.ServiceBus/Namespace/Queues",
  "updatedAt": "2021-04-14T02:24:28.887000+00:00"
}
]
]

$ az servicebus topic list --namespace-name mdfranz-servicebus-namespace -g \
  terraform-servicebus
[
  {
    "accessedAt": "0001-01-01T00:00:00+00:00",
    "autoDeleteOnIdle": "10675199 days, 2:48:05.477581",
    "countDetails": {
      "activeMessageCount": 0,
      "deadLetterMessageCount": 0,
      "scheduledMessageCount": 0,
      "transferDeadLetterMessageCount": 0,
      "transferMessageCount": 0
    },
    "createdAt": "2021-04-14T12:32:46.243000+00:00",
    "defaultMessageTimeToLive": "10675199 days, 2:48:05.477581",
    "duplicateDetectionHistoryTimeWindow": "0:10:00",
    "enableBatchedOperations": false,
    "enableExpress": false,
    "enablePartitioning": true,
    "id": "/subscriptions/SUBSCRIPTION/resourceGroups/terraform-servicebus/providers/ \
      Microsoft.ServiceBus/namespaces/mdfranz-servicebus-namespace/topics/ \
        mdfranz-servicebus-topic",
    "location": "Central US",
    "maxSizeInMegabytes": 81920,
    "name": "mdfranz-servicebus-topic",
    "requiresDuplicateDetection": false,
  }
]
```

```

        "resourceGroup": "terraform-servicebus",
        "sizeInBytes": 0,
        "status": "Active",
        "subscriptionCount": 0,
        "supportOrdering": false,
        "type": "Microsoft.ServiceBus/Namespaces/Topics",
        "updatedAt": "2021-04-14T12:32:46.487000+00:00"
    }
]

```

También podemos ver los recursos dentro de la consola de Azure, como se muestra en la Figura 10-13.

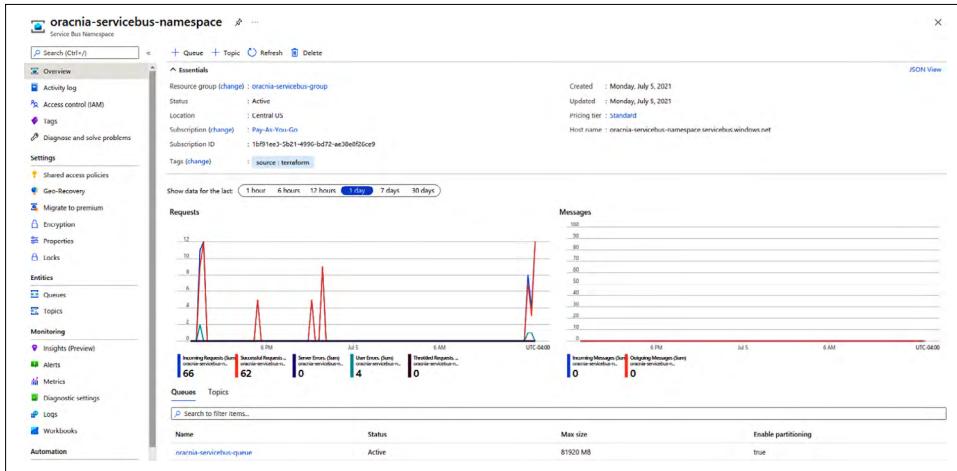


Figura 10-13. Visualización de Azure Portal de la implementación de Service Bus

También podemos ver las métricas generales en los espacios de nombres y hacer clic en una cola específica que proporcione más detalles sobre el almacenamiento y el tamaño del mensaje (consulte la Figura 10-14).

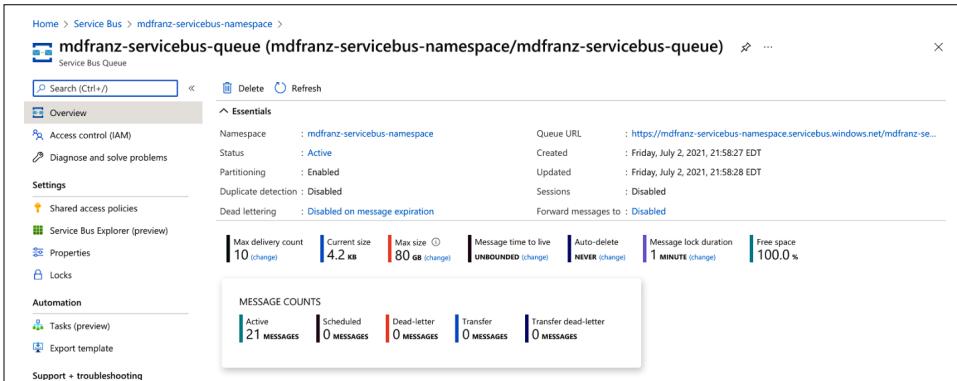


Figura 10-14. Visualización de Azure Portal de la cola de Service Bus

## Envío y recepción de mensajes para una cola de Service Bus en Python

Supongamos que creamos correctamente la infraestructura en los pasos anteriores, podemos recuperar una cadena de conexión que podemos usar mediante la ejecución de un comando de la CLI de Azure:

```
$ az servicebus namespace authorization-rule keys list --resource-group $RES_GROUP \
--namespace-name $NAMESPACE_NAME --name RootManageSharedAccessKey --query primaryConnectionString wBY=
```

Azure proporciona bibliotecas open source para enviar y recibir mensajes al servicio de Azure en todos los lenguajes de programación populares. En el siguiente ejemplo, se crea un productor y un consumidor sencillos en Python:

```
import os
connstr = os.environ['SERVICE_BUS_CONN_STR']
queue_name = os.environ['SERVICE_BUS_QUEUE_NAME']

messages = ["Testing 1", "Testing 2", "Testing 3"]

with ServiceBusClient.from_connection_string(connstr) as client:
    s = client.get_queue_sender(queue_name)
    r = client.get_queue_receiver(queue_name)

    print(s,r)

    for m in messages:
        s.send_messages(ServiceBusMessage(m))

    while True:
        print(r.receive_messages(max_message_count=1, max_wait_time=10))
```

Además de RabbitMQ, los lectores familiarizados con AWS SNS y SQS descubrirán que Service Bus es sencillo y puede cumplir con casos de uso similares.

## Azure Event Hubs

Azure Event Hubs está diseñado para proporcionar algunas de las capacidades de Apache Kafka a fin de satisfacer casos de uso de particiones y grupos de consumidores, pero en un servicio en la nube administrado (consulte la [Figura 10-15](#)). Aunque pierde la visibilidad de la infraestructura de back-end que tendría al ejecutar Kafka en máquinas virtuales o Kubernetes, obtiene una integración más estrecha con servicios de datos adicionales, como Azure Blob Storage, Azure Data Lakes y Azure Stream Analytics, por nombrar algunos. Azure también admite los protocolos AMQP 1.0 y Kafka 1.0 para mantener la compatibilidad con las bibliotecas cliente.

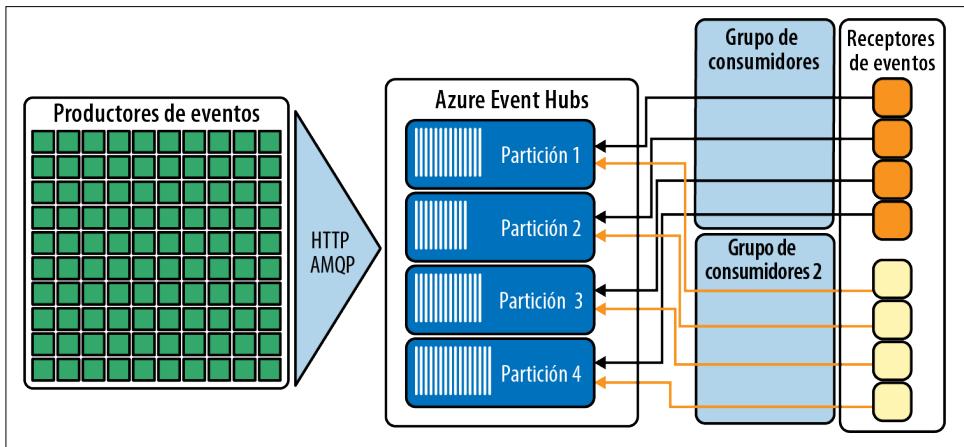


Figura 10-15. Arquitectura de Azure Event Hubs

Azure Event Hubs tiene algunos conceptos que se transfieren de Azure Service Bus. Los revisaremos rápidamente aquí:

#### *Productores*

Estos son los clientes que envían mensajes a Azure Event Hubs.

#### *Particiones*

Los mensajes se partitionan entre particiones para permitir que se almacenen réplicas del mensaje.

#### *Grupos de consumidores*

Estos son clientes que trabajan al unísono para consumir mensajes producidos de manera coordinada.

### Administración de Azure Event Hubs con Terraform

El código de Terraform necesario para aprovisionar los recursos que se requieren para enviar mensajes a Azure Event Hubs es similar al que anteriormente utilizamos para configurar una cola y un tema dentro de Azure Service Bus. Debemos declarar un grupo de recursos y un espacio de nombres de los que sea parte nuestra instancia de Event Hubs:

```

provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "example" {
  name      = "mdfranz-eventbus-group"
  location  = "East US"
}

resource "azurerm_eventhub_namespace" "example" {
  name          = "mdfranz-eventhub-namespace"
  location      = "East US"
}

```

```

resource_group_name = azurerm_resource_group.example.name
sku              = "Standard"
capacity         = 1
}

resource "azurerm_eventhub" "example" {
  name          = "mdfranz-eventhub"
  namespace_name = azurerm_eventhub_namespace.example.name
  resource_group_name = azurerm_resource_group.example.name
  partition_count    = 1
  message_retention = 1
}

resource "azurerm_eventhub_consumer_group" "example" {
  name          = "mdfranz-consumergroup"
  resource_group_name = azurerm_resource_group.example.name
  namespace_name   = azurerm_eventhub_namespace.example.name
  eventhub_name    = azurerm_eventhub.example.name
}

```

La creación de un grupo de consumidores es opcional, puesto que hay un grupo de consumidores predeterminado. Tenga en cuenta también que tener varios grupos de consumidores requiere la SKU estándar, en lugar de la SKU básica.

Como sucedía con Azure Event Bus, existen varias formas de autenticarse en los puntos de conexión de Event Hubs. La forma más fácil es utilizar una cadena de conexión, como se muestra en el siguiente ejemplo:

```

$ az eventhubs namespace authorization-rule keys list --resource-group mdfranz-eventbus- \
group --namespace-name mdfranz-eventhub-namespace --name RootManageSharedAccessKey
{
  "aliasPrimaryConnectionString": null,
  "aliasSecondaryConnectionString": null,
  "keyName": "RootManageSharedAccessKey",
  "primaryConnectionString": "Endpoint=sb://mdfranz-eventhub-namespace.servicebus.windows. \
  net;/SharedAccessKeyName=RootManageSharedAccessKey; \
  SharedAccessKey=jfMUbfpZX8UJglcCKLAEip5CZVLcwcSGpuKdJX/CMzk=",
  "primaryKey": "jfMUbfpZX8UJglcCKLAEip5CZVLcwcSGpuKdJX/CMzk=",
  "secondaryConnectionString": "Endpoint=sb://mdfranz-eventhub-namespace.servicebus. \
  windows.net;/SharedAccessKeyName=RootManageSharedAccessKey; \
  SharedAccessKey=gFy7SuuGjw12GfoHXd6UQeZdy5Y0xu/gStZtQuhaxsY=",
  "secondaryKey": "gFy7SuuGjw12GfoHXd6UQeZdy5Y0xu/gStZtQuhaxsY="
}

```



El parámetro **EntityPath=mdfranz-eventhub**, es decir:

```

Endpoint=sb://mdfranz-eventhub-namespace.servicebus.windows.net/;
SharedAccessKeyName=RootManageSharedAccessKey;EntityPath=mdfranz-
eventhubSharedAccessKey=jfMUbfpZX8UJglcCKLAEip5CZVLcwcSGpuKdJX/CMzk=

```

debe agregarse a la cadena de conexión, porque si está creando un cliente que usa `NewHubFromConnectionString`, el método `Send` no permite que se especifique un nombre de cuadrícula de eventos como parámetro, por lo que necesita agregar un `EntityPath` en su lugar.

## Azure Event Grid

Azure Event Grid se lanzó con disponibilidad general en 2018 y es el mejor de los agentes de mensajería ofrecidos por Azure, puesto que permite que varios orígenes de eventos se enruten a una variedad de destinos dentro del ecosistema de Azure. Los mensajes se pueden enviar hacia o desde los servicios de mensajes existentes que acabamos de analizar (Azure Service Bus y Azure Event Hubs) y a los servicios y puntos de conexión adicionales (que se muestran en la [Figura 10-16](#)), como Azure Functions, Webhooks, Logic Apps, y para otros fines en aplicación (por ejemplo, realizar un seguimiento de cómo los usuarios interactúan con la aplicación). Azure Event Hubs es una oferta “sin servidor” de primera clase en la que la facturación es por evento.

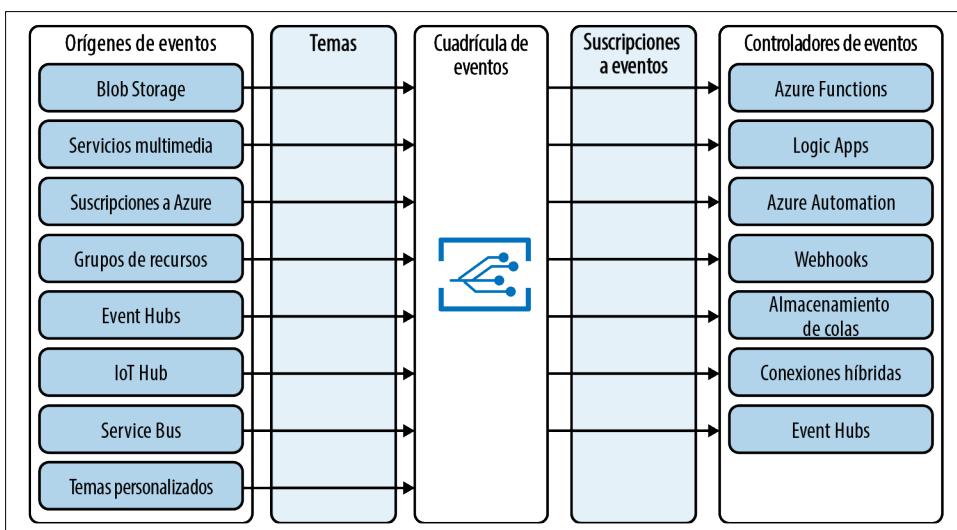


Figura 10-16. El ecosistema de Event Grid

En la [Figura 10-16](#), observe la naturaleza de inserción de Event Grid, a diferencia de la topología de inserción-extracción general a la que estamos acostumbrados en otros sistemas.

Azure Event Grid usa terminología que difiere ligeramente de las otras plataformas que analizamos:

### Dominios

Los dominios son análogos a los espacios de nombres, que separan los casos de uso dentro del inquilino.

### Temas

Para Event Grids, existen dos tipos de temas: los temas del sistema, que provienen de los sistemas de Azure (Event Hubs, Blob Storage, etc.), como se muestra a la izquierda en la [Figura 10-16](#); y los temas personalizados, que pueden ser su propio esquema personalizado incluido en el [Esquema de Event Grid](#).

## Implementación y uso de cuadrículas de eventos

Las cuadrículas de eventos de Azure tienen una implementación minimalista de Terraform, como se muestra en el siguiente ejemplo:

```
resource "azurerm_resource_group" "example" {
  name      = "example-resources"
  location  = "East US"
}

resource "azurerm_eventgrid_topic" "example" {
  name          = "my-eventgrid-topic"
  location      = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name

  tags = {
    environment = "Production"
  }
}
```

Le recomendamos cambiar el nombre del tema, puesto que es globalmente único en la región de Azure en la que se implementa. Una vez que se aplica esta implementación de Terraform, se creará un nuevo tema de Event Grid con el nombre <https://<event-grid-topicname>.<topic-location>.eventgrid.azure.net/api/events>; o en nuestro caso, <https://myeventgrid-topic.east-us.eventgrid.azure.net/api/events>. Para buscar la clave de acceso del tema, ejecute `az eventgrid topic key list --resourcegroup example-resources --name my-eventgrid-topic`.

Aquí hay un productor sencillo que usa el paquete `azure-eventgrid` de Python:

```
from azure.identity import DefaultAzureCredential
from azure.eventgrid import EventGridPublisherClient, EventGridEvent

event = EventGridEvent(
    data={"team": "azure-sdk"},
    subject="Door1",
    event_type="Azure.Sdk.Demo",
    data_version="2.0"
)

credential = DefaultAzureCredential("<access-key>")
endpoint = `https://my-eventgrid-topic.east-us.eventgrid.azure.net/api/events`
client = EventGridPublisherClient(endpoint, credential)
client.send(event)
```

Se encuentran disponibles varios ejemplos excelentes de cómo codificar con Event Grid en la [biblioteca cliente de Azure Event Grid para Python](#).

## Resumen

Tiene muchas opciones en el momento de elegir tecnologías de mensajería para implementar con aplicaciones nuevas o existentes. Como de costumbre, la respuesta a qué y cómo ejecutar una pieza de tecnología es con frecuencia “depende”. Existe una superposición significativa en cuanto a la funcionalidad en las ofertas de mensajería de open source y Azure que analizamos en este capítulo. Asegurarse de comprender los requisitos de durabilidad, escala y seguridad antes de elegir un servicio de mensajería le permitirá tener éxito cuando implemente una infraestructura de mensajería. Ahora pasaremos a la parte final de la infraestructura: la informática sin servidor.



# Informática sin servidor

Hasta aquí, nos hemos centrado en gran medida en tomar conceptos locales y analizar e implementar sus equivalentes basados en la nube. En este capítulo, profundizaremos en un nuevo tipo de arquitectura informática conocida como “sin servidor”, que se basa en la informática en la nube. En concreto, examinaremos plataformas sin servidor comunes, incluidas Azure Function Apps, Knative, KEDA y OpenFaaS.

## Introducción a la informática sin servidor

Al igual que con los contenedores, hay mucha exageración en torno a la informática sin servidor y lo que se puede hacer con ella. En este capítulo, se examinarán las características y los beneficios de la informática sin servidor.

### ¿Qué es la informática sin servidor?

A diferencia de los contenedores, que en gran medida aún contienen una serie de conceptos basados en host, la informática sin servidor abstrae todo eso hasta el punto en que a usted solo le preocupa poder ejecutar una función de software.

La informática sin servidor ha adquirido mucha popularidad debido a la capacidad de crear aplicaciones rápidamente y ejecutarlas sin preocuparse por la administración de la infraestructura. Además, esta considera el escalado automático como una característica de primera clase, lo que significa que no necesita preocuparse por nada más que escribir código e implementarlo a través de mecanismos de implementación de marco estandarizado.

El beneficio de la informática sin servidor es que la configuración y el mantenimiento del servidor no son una preocupación. Además, si desea ejecutar una función de software con poca frecuencia, las funciones sin servidor pueden ser una opción mucho más económica, puesto que paga por la solicitud y no por la reserva de los recursos informáticos (es decir, el proceso).

El panorama de la informática sin servidor, como se mencionó antes, es un espacio razonablemente nuevo que se ha innovado con rapidez. Además de los grandes proveedores de nube que ofrecen una plataforma de funciones, otras empresas de Internet como Cloudflare, Netlify y Twilio también ofrecen funcionalidades sin servidor.

## ¿Qué es una función sin servidor?

Una función sin servidor es un pequeño programa de corta duración que se ejecuta cuando se activa. Las funciones sin servidor se pueden desencadenar a través de una solicitud HTTP(s), a través de un temporizador o en Azure Event Grid mediante el desencadenamiento de una función basada en un evento de Cosmos DB o Service Bus.

Por lo general, una función tendrá un único propósito autónomo que se ejecuta rápidamente y tiene dependencias limitadas. Además, por lo general, no es un punto de conexión de qps (consultas por segundo) altas, puesto que el rendimiento es una preocupación secundaria de las funciones sin servidor.

Es posible que haya oído hablar de utilidades como <https://icanhazip.com>, que es un servicio que solo devuelve la dirección IP pública desde la que se está conectando. Esto se puede escribir e implementar fácilmente como una función sin servidor:

```
async function handleRequest(request) {
  const clientIP = request.headers.get("CF-Connecting-IP")
  return new Response(clientIP)
}

addEventListener("fetch", event => {
  event.respondWith(handleRequest(event.request))
})
```

En el ejemplo anterior, se crea una escucha de eventos para un tipo de evento “fetch” (que es HTTP GET) y, cuando se desencadena ese evento, se llama a la función `handleRequest`. La función analiza un encabezado de solicitud HTTP, `CF-Connecting-IP`, que es la IP que realiza la solicitud a la función sin servidor. A continuación, la función responde con la dirección IP de solicitud.

## El entorno sin servidor

Como se mencionó antes, a pesar de ser un nuevo espacio, el entorno de la informática sin servidor ha experimentado un rápido desarrollo del sistema. El entorno sin servidor de CNCF (que se muestra en la [Figura 11-1](#)) consta de varias secciones:

### Herramientas

Marcos de software que ayudan a compilar software sin servidor

### Seguridad

Software que está diseñado específicamente para proteger la infraestructura sin servidor

### Marcos

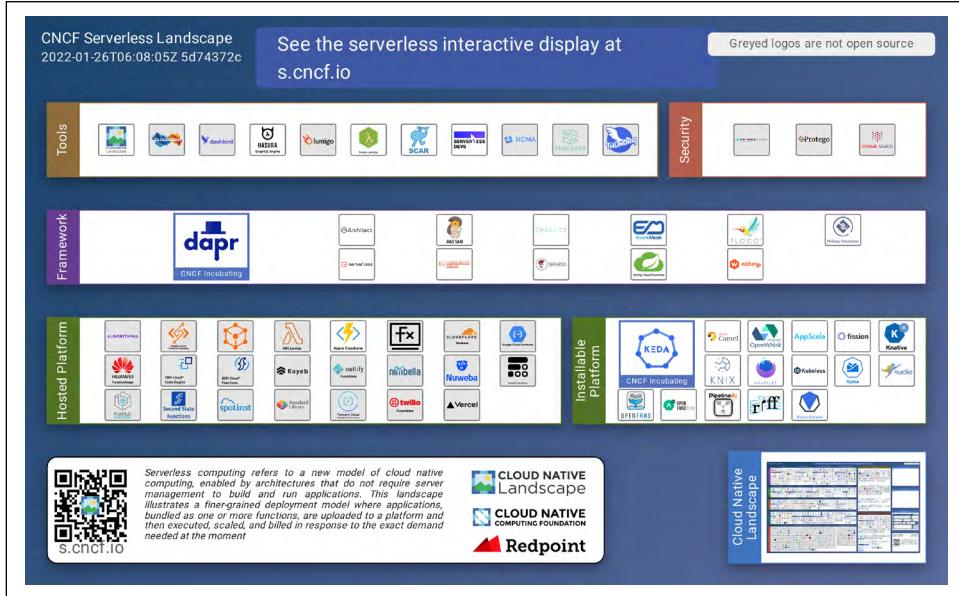
Marcos de software para crear funciones sin servidor

## *Plataforma hospedada*

Proveedores públicos que ofrecen funciones sin servidor

## *Plataforma instalable*

Software que puede instalar para ejecutar funciones sin servidor



*Figura 11-1. El entorno sin servidor de CNCF*

Aunque no lo abordaremos en detalle en este libro, más allá de los marcos y los proveedores de nube primarios que proporcionan ofertas sin servidor, los proveedores de red perimetral (o CDN) también ofrecen funcionalidad sin servidor o de “proceso perimetral”. Esto se ha vuelto extremadamente popular en empresas como Cloudflare.

# Beneficios de la informática sin servidor

Cuando se usa correctamente, la informática sin servidor puede aportarle al usuario beneficios importantes:

### *Costos más bajos*

Se le factura por la cantidad de veces o la cantidad de tiempo en que se ejecuta la función, lo que significa que no tiene que pagar por recursos de nube inactivos.

*Se elimina la sobrecarga operativa*

Debido a la simplicidad del modelo de función y las dependencias limitadas, la necesidad de aprovisionar o administrar activamente la función se reduce mucho.

#### *Escalado automático*

La función escala automáticamente y no se requiere ningún esfuerzo adicional para escalar o reducir verticalmente el número de trabajadores de función.

#### *Ciclos de desarrollo más rápidos*

Unir todos los puntos anteriores ayuda a generar un ciclo de ingeniería más rápido, lo que puede conducir a una mayor productividad del desarrollador y del operador.

## **Posibles desventajas de la informática sin servidor**

La informática sin servidor también puede tener desventajas potenciales que también deben considerarse:

#### *Capacidad de depuración*

Cuando todo el proceso es totalmente efímero y de corta duración, la depuración de la función es muy compleja.

#### *Una gran cantidad de funciones*

Si bien el escalado automático se integra de forma inherente en todas las funciones sin servidor, hay una serie de efectos secundarios que siempre se escalan automáticamente. En primer lugar, puede afectar de forma negativa al rendimiento y, en segundo lugar, administrar la agrupación de conexiones en los niveles más bajos se vuelve más complicado (en especial, con las bases de datos).

#### *Bloqueo de proveedores*

Una desventaja de todas las plataformas sin servidor es que el código de su función está directamente vinculado a la plataforma de software en la que está ejecutando sus funciones. Esto limita mucho su portabilidad con el tiempo.

Ahora que explicamos qué es la informática sin servidor y cuáles son algunos de los beneficios de usar la tecnología, analizaremos varias plataformas en las que ejecutar funciones sin servidor.

## **Azure Function Apps**

Azure Function Apps es la oferta de aplicaciones sin servidor de la plataforma de Azure. Las aplicaciones de funciones proporcionan la capacidad de ejecutar código o contenedores en entornos Linux o Windows compatibles con varios lenguajes de programación, entre los que se incluyen los siguientes:

- .NET Platform (C#, F#)
- JavaScript y TypeScript
- Python
- Java
- PowerShell Core

Una aplicación de funciones de Azure tendrá un plan asociado que dicta la forma en que se escalará la aplicación, las características disponibles y el precio que paga. En el momento de la redacción de este libro, existen tres opciones:

#### *Consumo*

Este es un plan básico en el que la función se escala automáticamente y solo se paga cuando se ejecutan las funciones.

#### *Funciones premium*

Esto se escala automáticamente según demanda mediante trabajadores preparados. Se ejecuta con instancias más poderosas y permite una funcionalidad de red avanzada.

#### *Plan de App Service*

Esto es para escenarios de larga duración en los que también se necesita escalado y facturación predictivos.

Puede obtener más información sobre cada plan en la página de la documentación de [opciones de hospedaje de Azure Functions](#).

## Arquitectura de la aplicación de funciones

La arquitectura de aplicaciones de Azure (consulte la [Figura 11-2](#)) se ajusta al conjunto más amplio de ofertas de Azure, que incluyen Cosmos DB, cuentas de almacenamiento y Azure Pipelines. La integración nativa con Cosmos DB y las cuentas de almacenamiento hace que sea una oferta atractiva para la integración sencilla con otras ofertas de Azure. Aunque no lo trataremos en este libro, las funciones de Azure también se vinculan con la oferta de integración continua/implementación continua (CI/CD) en Azure Pipelines y Azure Monitor puede supervisarla.

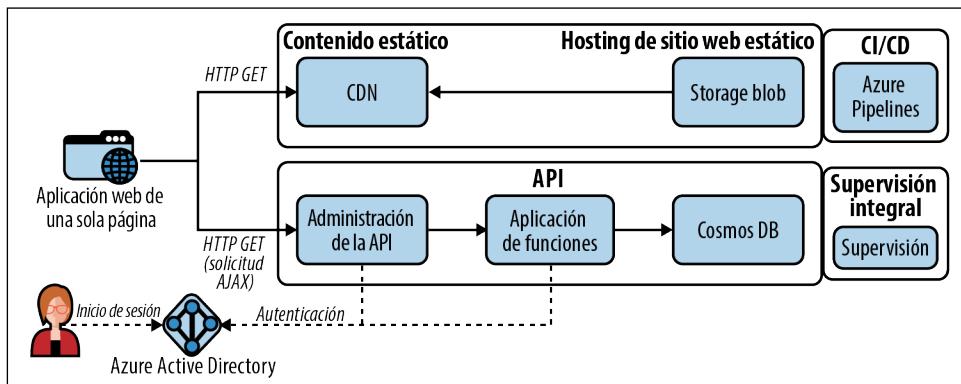


Figura 11-2. Arquitectura de referencia de la función sin servidor de Azure

Cuando crea una aplicación de funciones, también debe vincular una cuenta de almacenamiento. Esta se puede utilizar para almacenamiento temporal. Cada aplicación se aprovisionará con

un punto de conexión HTTP(s) público en la creación para que la función sea accesible. Es posible proteger la aplicación de funciones mediante un punto de conexión privado.

## Creación de una aplicación de funciones

En esta sección, crearemos una aplicación de funciones con Azure Portal:

1. Comenzaremos por abrir Azure Portal y hacer clic en Crear en Aplicación de funciones.
2. Ahora completaremos los parámetros básicos de la aplicación de funciones:
  - a. Crearemos un nuevo grupo de recursos llamado `serverless-test`.
  - b. El nombre DNS de nuestra aplicación será `ourtestfunctionapp.azurewebsites.net`.
  - c. Ejecutaremos código (una función), en lugar de un contenedor.
  - d. El lenguaje será Python v3.9.
  - e. La aplicación de funciones se implementará en la región este de EE. UU.
3. En la pestaña Hospedaje:
  - a. Crearemos una nueva cuenta de almacenamiento llamada `functionstorageaccount`. Como los nombres de las cuentas de almacenamiento son globalmente únicos, tendrá que elegir su propio nombre.
  - b. Debido a que estamos ejecutando Python, debemos ejecutar Linux.
  - c. Por último, utilizaremos el plan de consumo para fines de este ejemplo.
4. En la pestaña Supervisión:
  - a. Desactivaremos Application Insights.
  - b. Luego, haremos clic en Revisar y crear.

Ahora que creamos nuestra aplicación de funciones, necesitamos crear una función. Visual Studio Code tiene una extensión, Azure Functions, que facilita la creación, prueba y publicación de una función:

1. Si no ha instalado Visual Studio Code puede descargarlo en la [página de descarga de Visual Studio Code](#).
2. Una vez que haya instalado Visual Studio Code, tendrá que instalar la extensión [Azure Functions](#).
3. Haga clic en el ícono de Azure en la barra de actividad en el lado izquierdo de la pantalla, que abrirá la ventana de Azure Functions. Haga clic en “Crear proyecto nuevo”.
4. Aparecerá un cuadro de diálogo donde deberá elegir un directorio local en el que guardar el código.

5. A continuación, se le pedirá información del proyecto en las ventanas emergentes de Visual Studio Code independientes:
  - a. Seleccione un lenguaje para el proyecto de función: elija Python.
  - b. Seleccione un alias de Python para crear un entorno virtual: elija la ubicación de su intérprete de Python.
  - c. Si no se muestra la ubicación, escriba la ruta completa a su archivo binario de Python.
  - d. Seleccione una plantilla para la primera función de su proyecto: elija “Desencadenador de HTTP”.
  - e. Proporcione un nombre de función: escriba `HttpExample`.
  - f. Elija un nivel de autorización: seleccione Anónimo, que permite a cualquier persona llamar el punto de conexión de su función. Para obtener información sobre los niveles de autorización, consulte la documentación de [las claves de autorización](#).
  - g. Seleccione cómo le gustaría abrir su proyecto: elija “Aregar al espacio de trabajo”.
  - h. Con esta información, Visual Studio Code genera un proyecto de Azure Functions con un desencadenador HTTP. Puede ver los archivos de proyecto locales en el explorador. Para obtener más información sobre los archivos que se crean, consulte la documentación de [archivos de proyecto generados](#).
6. En este momento, tendrá la estructura de un proyecto creada. Agregue el siguiente código al archivo `HttpExample`, que le devolverá la IP del cliente que realiza una solicitud a la función (similar a [icanhazip.com](http://icanhazip.com)):

```
import azure.functions as func

def main(req: func.HttpRequest) -> func.HttpResponse:
    response = False
    if (req.headers.get('X-Forwarded-For')):
        response = req.headers.get('X-Forwarded-For').split(':')[0]
    else:
        response = "127.0.0.1"
    return func.HttpResponse(response)
```

7. Presione F5 en su teclado para probar su función y, luego, vaya a <http://localhost:7071/api/HttpExample>.
8. Para implementar en Azure, deberá hacer clic en el botón “Iniciar sesión en Azure” en la ventana de Azure Functions.
9. Haga clic en el botón Cargar, que le permitirá cargar su función en la aplicación que creó antes.

Ahora que examinamos las ofertas nativas de Azure, analicemos las aplicaciones de funciones implementadas con Kubernetes.

# Knative

Knative es una plataforma sin servidor basada en Kubernetes y fue una de las primeras favoritas de la industria, dado el entusiasmo en torno a Kubernetes en el momento de su lanzamiento. Una de las ventajas de Knative es que las funciones se empaquetan y ejecutan como contenedores Docker, lo que hace que el esfuerzo de integración en un entorno de nube existente sea mínimo.

## Arquitectura de Knative

Knative tiene dos modos de operación:

- **Knative Serving** utiliza Kubernetes para implementar funciones o contenedores sin servidor y tiene soporte completo para el escalado automático (incluida la reducción de los pods a cero). También tiene una ventaja adicional de apoyar las soluciones de red en la nube, como Gloo e Istio.
- **Knative Eventing** permite a los desarrolladores desarrollar aplicaciones creadas en torno a una arquitectura basada en eventos. En una implementación de eventos, puede especificar los productores de eventos (origenes) a los que responder.

Veamos cómo instalar Knative en Kubernetes.

## Instalación y ejecución de Knative Serving en Kubernetes

En esta sección, ejecutaremos un sistema de servicio básico de Knative:

1. Empiece por instalar los recursos personalizados:

```
$ kubectl apply -f https://github.com/knative/net-kourier/releases/download/v0.26.0/ \
  kourier.yaml
```

2. A continuación, instale los componentes de servicio:

```
$ kubectl apply -f https://github.com/knative/serving/releases/download/v0.26.0/ \
  serving-core.yaml
```

Si desea utilizar cualquiera de las integraciones de red (Kourier, Ambassador, Contour, Istio), puede [seguir las instrucciones de la capa de redes](#).

3. Ahora debe crear su aplicación. Usaremos la biblioteca de Flask de Python para crear una aplicación sencilla:

```
Import logging
import os

from flask import Flask

app = Flask(__name__)
Log =
    @app.route('/', methods=['POST'])
def hello_world():
    return f'Hello world. Data {request.data}\n'
```

```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=int(os.environ.get('PORT', 8080)))
```

4. Ahora necesita una imagen de Docker que contenga su aplicación sin servidor:

```
FROM python:3.7-slim

RUN pip install Flask gunicorn

WORKDIR /app
COPY . .

CMD exec gunicorn --bind :$PORT --workers 1 --threads 8 app:app
```

5. Compile e inserte su nueva imagen sin servidor con el siguiente código:

```
$ docker push https://myharborinstallation.com/helloworld:v1
```

6. Ahora defina su servicio sin servidor en un archivo llamado *hello-world.yaml*:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld
  namespace: default
spec:
  template:
    metadata:
      name: helloworld
    spec:
      containers:
        - image: docker.io/test/helloworld:v1
```

7. Implemente la función mediante el siguiente código:

```
$ kubectl apply -f hello-world.yaml
```

8. En este momento, se implementará su función sin servidor. Para encontrar la URL de la función, ejecute el código:

```
$ kubectl get ksvc hello-world --output=custom-columns=NAME:.metadata.name, \
  URL:.status.url
```

9. Pruebe la función llamando a la URL:

```
$ curl -X POST <URL found from above> -d "My Data"
Hello World: My Data
```

Ahora que se está ejecutando una función de servicio básica, examinemos Knative Eventing.

# Instalación y ejecución de Knative Eventing en Kubernetes

En esta sección, instalaremos componentes de Knative Eventing y escribiremos un sistema de eventos simple:

1. Empiece por instalar los recursos personalizados:

```
$ kubectl apply -f https://github.com/knative/eventing/releases/download/v0.26.0/ \
eventing-crds.yaml
```

2. Ahora, instale los componentes de servicio:

```
$ kubectl apply -f https://github.com/knative/eventing/releases/download/v0.26.0/ \
eventing-core.yaml
```

En este momento, tendrá instalados los componentes de eventos. Reutilizaremos nuestra imagen de Docker de la sección anterior. Sin embargo, configuraremos Knative para que podamos enviar un mensaje HTTP a un bus de mensajes dentro de Knative, en lugar de publicarlo en un servidor de aplicaciones HTTP. Cuando el mensaje llegue al bus de mensajes, desencadenará nuestro servidor de aplicaciones de Python. En un ejemplo más complicado, puede establecer un tema de Kafka como origen de eventos.

3. Cree un archivo *knative-eventing.yaml*:

```
# Namespace for sample application with eventing enabled
apiVersion: v1
kind: Namespace
metadata:
  name: knative-samples
  labels:
    eventing.knative.dev/injection: enabled
---
# A default broker
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: default
  namespace: knative-samples
  annotations:
    # Note: you can set the eventing.knative.dev/broker.class annotation to change
    # the class of the broker.
    # The default broker class is MTChannelBasedBroker, but Knative also supports
    # use of the other class.
    eventing.knative.dev/broker.class: MTChannelBasedBroker
spec: {}
---
# hello-world app deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
  namespace: knative-samples
spec:
  replicas: 1
  selector:
    matchLabels: &labels
      app: hello-world
```

```

template:
  metadata:
    labels: *labels
  spec:
    containers:
      - name: hello-world
        image: https://myharborinstallation.com/helloworld:v1
        imagePullPolicy: IfNotPresent
    ...
# Service that exposes helloworld-python app.
# This will be the subscriber for the Trigger
apiVersion: v1
kind: Service
metadata:
  name: hello-world
  namespace: knative-samples
spec:
  selector:
    app: hello-world
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  ...
# Knative Eventing Trigger to trigger the helloworld-python service
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: hello-world
  namespace: knative-samples
spec:
  broker: default
  filter:
    attributes:
      type: dev.knative.samples.hello-world
      source: dev.knative.samples/helloworldsource
  subscriber:
    ref:
      apiVersion: v1
      kind: Service
      name: hello-world

```

4. Aplique la configuración *knative-eventing.yaml* en Kubernetes:

```
$ kubectl apply -f knative-eventing.yaml
```

Esto creará un nuevo espacio de nombres de Kubernetes llamado **knative-samples** que permite los eventos, y un agente de mensajes que pasará los mensajes a nuestro servidor de aplicaciones de Python (en eventos).

5. Ahora enviaremos un evento a nuestro agente de mensajes de Knative. Aquí está la respuesta:

```
$ curl -v "broker-ingress.knative-eventing.svc.cluster.local/knative-samples/default" \
-X POST \
-d 'My data'

Event received. Context: Context Attributes,
specversion: 0.3
type: dev.knative.samples.hello-world
source: dev.knative.samples/helloworldsource
```

```

id: 536808d3-88be-4077-9d7a-6a3f162705f79
time: 2021-11-09T06:21:26.09471798Z
datacontenttype: application/json
Extensions,
  Knativearrivaltime: 2021-11-09T06:21:26Z
  knativehistory: default-kn2-trigger-kn-channel.knative-samples.svc.cluster.local
  traceparent: 00-971d4644229653483d38c46e92a959c7-92c66312e4bb39be-00

Hello World Message "My data"
Responded with event Validation: valid
Context Attributes,
  specversion: 0.2
  type: dev.knative.samples.hifromknative
  source: knative/eventing/samples/hello-world
  id: 37458d77-01f5-411e-a243-a459bbf79682
Data,
  My data

```

## KEDA

KEDA es la sigla de *Kubernetes Event-Driven Autoscaling*. KEDA le permite escalar cualquier contenedor en Kubernetes en función de la cantidad de eventos entrantes. KEDA es un complemento de Kubernetes y utiliza el Escalador automático horizontal de pod (HPA) para escalar en función de una variedad de orígenes de eventos, incluidos Apache Kafka, Azure Event Hubs y streaming de NATS.

## Arquitectura de KEDA

En la [Figura 11-3](#), se muestra una arquitectura típica de KEDA, donde KEDA interactúa directamente con el servidor de API de Kubernetes, HPA, y el origen de desencadenador configurado.

El componente de KEDA cumple los siguientes roles:

### *Controlador y escalador*

El controlador y escalador escalarán o reducirán verticalmente la cantidad de pods en función de la señal del adaptador de métricas.

### *Adaptador de métricas*

El adaptador de métricas actúa como un servidor de métricas de Kubernetes que proporciona información de longitud de cola o de desfase de transmisión al agente. El escalador usa esta información para escalar y reducir verticalmente los contenedores.

Puede ver todos los escaladores de KEDA en la [documentación de escaladores](#).

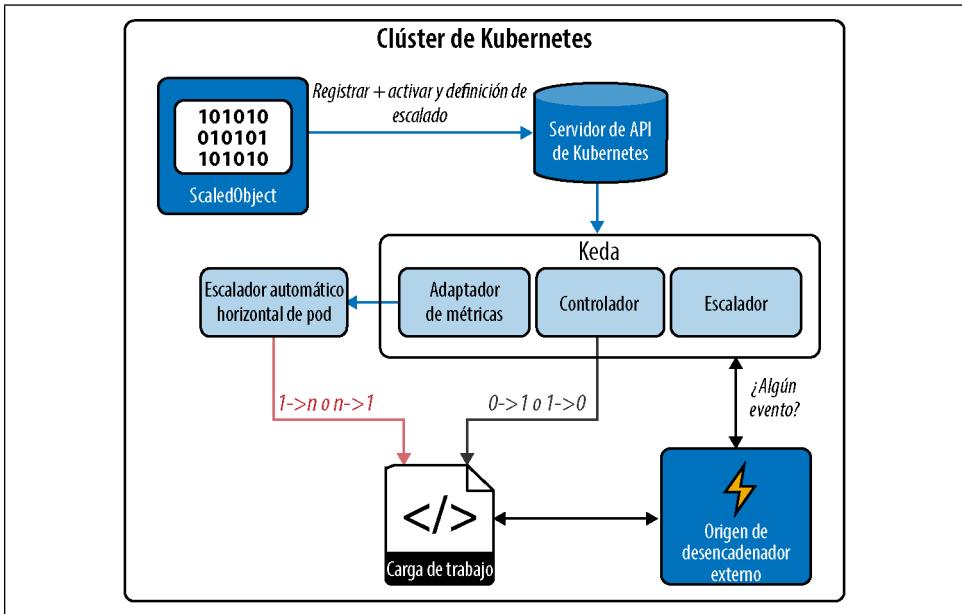


Figura 11-3. Arquitectura de referencia de KEDA

## Instalación de KEDA en Kubernetes

KEDA se instala mejor con el operador de Helm en Kubernetes. Revisaremos esto ahora:

1. Agregue el repositorio de Helm a KEDA:

```
$ helm repo add kedacore https://kedacore.github.io/charts
```

2. Instale el gráfico de Helm:

```
$ kubectl create namespace keda
$ helm install keda kedacore/keda --namespace keda
```

Después de esto, el clúster de Kubernetes estará listo para la implementación de la función.

En el siguiente ejemplo, compilaremos un sistema de eventos de prueba de concepto con una cola de Azure Service Bus. Produciremos mensajes para la cola de Azure Service Bus mediante un script de Python y, luego, usaremos KEDA para escalar un trabajo de consumo de Python:

1. Tome la cadena de conexión de Service Bus con el método manual que se describe en la [documentación de Azure](#) o mediante la ejecución del siguiente código:

```
$ az servicebus namespace authorization-rule keys list \
-n RootManageSharedAccessKey -g <group> --namespace-name <namespace> \
--query primaryConnectionString -o tsv
```

- Necesitará obtener los valores codificados en Base64 de la cadena de conexión y el nombre de la cola mediante la ejecución del siguiente código:

```
$ echo -n "<connection string>" | base64
$ echo -n "<queue name>" | base64
```

- Ahora cree un archivo con el nombre *secret.yaml*. Reemplace los valores **servicebus-queue** y **servicebus-connectionstring** con las cadenas Base64 generadas en el paso anterior:

```
apiVersion: v1
kind: Secret
metadata:
  name: sample-job-secret
  namespace: default
data:
  servicebus-queue: <base64_encoded_servicebus_queue_name>
  servicebus-connectionstring: <base64_encoded_servicebus_connection_string>
```

- Cree un segundo archivo, llamado *keda.yaml*, que creará su ScaledJob. Ejecutará una imagen de Docker ([docker.pkg.github.com/prabdeb/sample-python-keda-service-bus-scaler/consumer:latest](https://docker.pkg.github.com/prabdeb/sample-python-keda-service-bus-scaler/consumer:latest)) cuando se desencadene:

```
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: auth-service-bus-sample-job
spec:
  secretTargetRef:
    - parameter: connection
      name: sample-job-secret
      key: servicebus-connectionstring
---
apiVersion: keda.sh/v1alpha1
kind: ScaledJob
metadata:
  name: servicebus-queue-so-sample-job
  namespace: default
spec:
  jobTargetRef:
    parallelism: 1 # max number of desired pods
    completions: 1 # desired number of successfully finished pods
    activeDeadlineSeconds: 600 # Specifies the duration in seconds relative to the
      startTime that the job may be active before the system tries to terminate it;
      value must be positive integer
    backoffLimit: 6 # Specifies the number of retries before marking this job failed.
      Defaults to 6
    template:
      metadata:
        labels:
          app: sample-keda-job
      spec:
        containers:
          - name: sample-keda-job
            image: docker.pkg.github.com/prabdeb/sample-python-keda-service-bus-scaler/
              consumer:latest
        env:
          - name: SERVICE_BUS_CONNECTION_STR
```

```

valueFrom:
  secretKeyRef:
    name: sample-job-secret
    key: servicebus-connectionstring
  - name: SERVICE_BUS_QUEUE_NAME
    valueFrom:
      secretKeyRef:
        name: sample-job-secret
        key: servicebus-queue
    restartPolicy: Never
  pollingInterval: 30          # Optional. Default: 30 seconds
  successfulJobsHistoryLimit: 100 # Optional. Default: 100.
                                 # How many completed jobs should be kept.
  failedJobsHistoryLimit: 100   # Optional. Default: 100.
                                 # How many failed jobs should be kept.
  #envSourceContainerName: {container-name} # Optional. Default:
                                             # .spec.JobTargetRef.template.spec
                                             # .containers[0]
  maxReplicaCount: 100         # Optional. Default: 100
  scalingStrategy:
    strategy: "custom"        # Optional. Default: default.
                               # Which Scaling Strategy to use.
    customScalingQueueLengthDeduction: 1 # Optional. A parameter
                                         # to optimize custom ScalingStrategy.
    customScalingRunningJobPercentage: "0.5" # Optional. A parameter
                                              # to optimize custom ScalingStrategy.
  triggers:
  - type: azure-servicebus
    metadata:
      # Required: queueName OR topicName and subscriptionName
      queueName: <servicebus_queue_name>
      # Required: Define what Azure Service Bus
      # to authenticate to with Managed Identity
      namespace: <servicebus_namespace>
      messageCount: "10" # default 5
    authenticationRef:
      name: auth-service-bus-sample-job # authenticationRef would need either
                                         podIdentity or define a connection parameter

```

5. Ahora puede aplicar los dos archivos a Kubernetes mediante este código:

```
$ kubectl apply -f keda-config/jobs/secret.yaml
$ kubectl apply -f keda-config/jobs/keda.yaml
```

6. Para comprobar que su ScaledJob está listo, ejecute:

```
$ kubectl get ScaledJob
$ kubectl get TriggerAuthentication
```

7. Puede probar la implementación de KEDA mediante la ejecución de un script de prueba. En primer lugar, debe configurar el entorno de Python:

```
$ python3 -m venv .
$ source bin/activate
$ pip3 install logger
$ pip3 install azure-servicebus
```

8. Ahora necesita crear su programa que envía mensajes a la cola de Azure Service Bus. Deberá modificar `connection_string` y `queue_name`:

```

import os
import sys
import time
import yaml
from logger import logger
from azure.servicebus import ServiceBusClient, ServiceBusMessage


def send_a_list_of_messages(sender):
    messages = [ServiceBusMessage(f"Message in list {i}") for i in range(1000)]
    sender.send_messages(messages)
    logger.info("Sent a list of 1000 messages")

def main():
    with open("azure-service-bus.yaml", 'r') as stream:
        config = yaml.safe_load(stream)

    logger.info("Start sending messages")
    connection_string = <FIXME>
    queue_name = <FIXME>

    queue = ServiceBusClient.from_connection_string(conn_str=connection_string, \
        queue_name=queue_name)

    with queue:
        sender = queue.get_queue_sender(queue_name=queue_name)
        with sender:
            send_a_list_of_messages(sender)

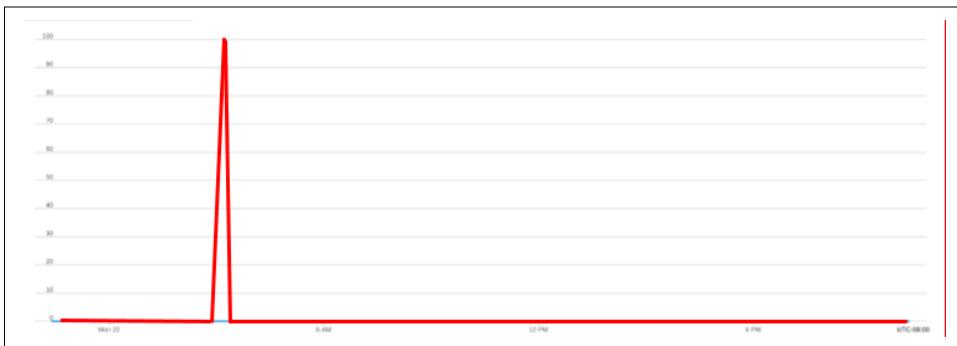
    logger.info("Done sending messages")
    logger.info("-----")

if __name__ == "__main__":
    main()

```

- Ejecute `python3 send_message_queue.py`. Esto creará 1000 mensajes y los enviará a la cola de Azure Service Bus.

Puede ver los mensajes que se ingieren y, luego, procesan en la cola de Azure Service Bus (**Figura 11-4**).



*Figura 11-4. Cola de Azure Service Bus*

En la siguiente sección, analizaremos OpenFaaS, que ofrece funciones sencillas en sistemas independientes y clústeres de Kubernetes.

## OpenFaaS

OpenFaaS es otro servicio de función basado en eventos para Kubernetes que le permite escribir funciones en cualquier lenguaje y, luego, empaquetarlas en contenedores compatibles con OCI o Docker.

### Arquitectura de OpenFaaS

La arquitectura de referencia de OpenFaaS (como se muestra en la [Figura 11-5](#)) es bastante similar a KEDA. Sin embargo, OpenFaaS se basa en las métricas de Prometheus para actuar como una señal de escalado automático.

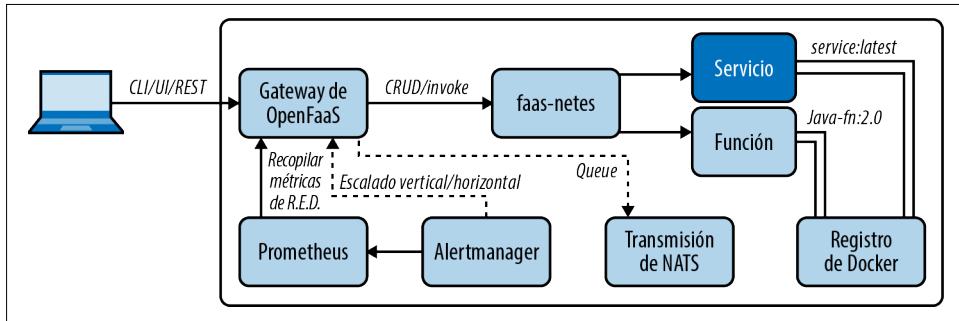


Figura 11-5. La arquitectura de referencia de OpenFaaS

### Instalación de OpenFaaS

OpenFaaS tiene tres métodos de instalación recomendados:

- Uso de `arkade`
- Uso de `helm`
- Instalación con Flux o ArgoCD

Para mantener la coherencia con el resto del libro, implementaremos OpenFaaS con Helm:

1. Cree los espacios de nombres (uno para los servicios principales de OpenFaaS y uno para las funciones):

```
$ kubectl apply -f https://raw.githubusercontent.com/openfaas/faas-netes/master/ \
namespaces.yaml
```

2. Agregue el gráfico Helm para OpenFaaS:

```
$ helm repo add openfaas https://openfaas.github.io/faas-netes/
```

3. Por último, implemente OpenFaaS:

```
$ helm repo update \
&& helm upgrade openfaas --install openfaas/openfaas \
--namespace openfaas \
--set functionNamespace=openfaas-fn \
--set generateBasicAuth=true
```

4. Podrá recuperar la contraseña de administrador con el siguiente comando. Este se utilizará para acceder al panel de OpenFaaS:

```
$ PASSWORD=$(kubectl -n openfaas get secret basic-auth -o \
jsonpath="{.data.basic-auth-password}" | base64 --decode) && \
echo "OpenFaaS admin password: $PASSWORD"
```

Ahora que instalamos OpenFaaS, escribiremos una función de OpenFaaS.

## Cómo escribir su primera función de OpenFaaS

En esta sección, utilizaremos faas-cli para crear e implementar rápidamente una función sin servidor:

1. Descargue faas-cli mediante la ejecución de brew `install faas-cli` en OSX o de curl `-sSL https://cli.openfaas.com | sudo sh` en Linux.
2. Cree una carpeta para colocar la nueva función:

```
$ mkdir -p ~/functions && cd ~/functions
```

3. Use faas-cli para crear un nuevo proyecto de función de Python:

```
$ faas-cli new --lang python helloworld
```

Esto creará tres archivos nuevos:

```
$ ls
helloworld/handler.py
helloworld/requirements.txt
helloworld.yml
```

El archivo `handler.py` será donde escriba su función. El archivo `requirements.txt` es el mismo que un archivo `requirements.txt` estándar de Python. El archivo `helloworld.yml` define algo de código reutilizable de la función.

4. En el archivo `handler.py`, definiremos nuestra función. Esta función imprimirá "Hello World! You said: " más los datos publicados en el punto de conexión de la función:

```
def handle(req):
    print("Hello World! You said: " + req)
```

5. Por último, en su archivo `helloworld.yml`, definirá los metadatos sobre su función. `http://127.0.0.1:31112` es la dirección predeterminada del servidor de gateway/API de OpenFaaS:

```
provider:
  name: openfaas
  gateway: http://127.0.0.1:31112
```

```
functions:  
  hello-python:  
    lang: python  
    handler: ./hello-world  
    image: hello-world
```

6. Compile la imagen sin servidor e insértela en el registro Harbor que creó en el [Capítulo 2](#):

```
$ faas-cli build -f ./hello-world.yml  
$ faas-cli push -f ./hello-world.yml
```

7. Impleméntela en el clúster de Kubernetes:

```
$ faas-cli deploy -f ./hello-world.yml  
Deploying: hello-python.  
No existing service to remove  
Deployed.  
200 OK  
URL: http://127.0.0.1:31112/function/hello-python
```

8. El paso anterior genera una URL que puede probar. Utilice `curl` para probarla:

```
$ curl -X POST 127.0.0.1:31112/function/hello-world -d "My name is Michael"  
Hello world! My name is Michael
```

De forma predeterminada, OpenFaaS utiliza solicitudes por segundo (RPS) como un mecanismo para realizar el escalado automático. Puede obtener más información sobre la configuración de escalado automático de OpenFaaS en la [página de documentación de escalado automático](#).

En este momento, tendrá una función OpenFaaS en ejecución en Kubernetes. También puede ejecutar OpenFaaS fuera de Kubernetes con `faasd`. Hay más información en la [página de implementación de OpenFaaS](#).

Por último, OpenFaaS se basa en la HPA de Kubernetes. Puede obtener más información sobre cómo realizar pruebas de carga y escalar automáticamente la función OpenFaaS en la [página del tutorial de OpenFaaS Kubernetes](#).

## Resumen

Aunque la informática sin servidor definitivamente no es para todos, cuando se utiliza en las situaciones correctas, puede proporcionar ventajas significativas. El espacio sin servidor aún está evolucionando, pero la barrera de entrada es extremadamente baja, lo que hace que sea una tecnología emocionante. En este capítulo, exploramos cuatro formas diferentes en las que puede ejecutar funciones sin servidor en Azure. Primero examinamos Azure Function Apps y, luego, revisamos tres proyectos principales de CNCF: Knative, KEDA y OpenFaaS. En este punto, hemos delineado los principales pilares de la infraestructura nativa de la nube. En el [Capítulo 12](#), resumiremos lo que aprendió en este recorrido.



## CAPÍTULO 12

---

# Conclusión

Hemos llegado al final del libro. Así que resumamos nuestro recorrido. A lo largo de este libro, nuestro objetivo fue proporcionar un único recurso que pueda utilizar para crear y ejecutar una infraestructura nativa de la nube en Azure. Aunque nos hemos enfocado en Azure, también puede aplicar el mismo conocimiento para crear y operar su propio entorno de nube nativa en diferentes plataformas de proveedores de nube. Revisemos lo que ha aprendido hasta ahora.

Comenzamos este libro recordando la forma en que el mundo de la infraestructura y los servicios evolucionó con el tiempo con la introducción de la nube. A fin de aprovechar al máximo lo que la nube nos ofrece, necesitamos desarrollar soluciones que sean nativas de la nube y no solo adaptarlas a ella de manera retroactiva. Entendimos cómo lo nativo de la nube es el camino a seguir a fin de utilizar completamente su poder.

En el [Capítulo 2](#), presentamos los pasos clave para crear un entorno nativo de nube moderna con Azure. Configuramos una cuenta de Azure y comenzamos a usar Ansible, Terraform y Packer. Al emplearlos como nuestras herramientas clave, podemos habilitar los servicios listos para la producción de forma automatizada. Usamos Microsoft Azure como nuestra opción de proveedor de nube, y a lo largo del libro usamos los servicios nativos de la nube y de Azure en conjunto para mostrar cómo se construye y brinda soporte a la infraestructura moderna.

En el [Capítulo 3](#), le presentamos el contenedor y el tiempo de ejecución del contenedor, que constituyen los fundamentos del mundo nativo de la nube moderna. Analizamos las capas de abstracción que componen el ecosistema de contenedores.

Luego, pasamos al poderoso orquestador, Kubernetes, en el [Capítulo 4](#) y el [Capítulo 5](#), y mostramos los aspectos esenciales de cómo esta importante herramienta se puede usar en la infraestructura de nube moderna para facilitar la plataforma de aplicaciones nativas de la nube. Kubernetes condujo a la creación y la adaptación de la Cloud Native Computing Foundation (CNCF) y fue el primer proyecto que se graduó de ella. Esto ha llevado a una

gran cantidad de optimización del mundo de la contenedorización y a una mayor adaptación del ecosistema nativo de la nube. Analizamos los diversos componentes y conceptos que hacen que este sistema opere a gran escala, y aprendimos a crear el clúster de Kubernetes de forma manual y con Azure AKS como un servicio administrado.

En este punto, ya habíamos cubierto los aspectos básicos de cómo se ve una infraestructura nativa de nube mínima. El siguiente paso fue agregar una forma de obtener información sobre la infraestructura y las aplicaciones. Así, en el [Capítulo 6](#), presentamos el concepto de los tres pilares de la observabilidad, para darle información sobre esta enorme infraestructura y pila de aplicaciones. Mediante la creación de sistemas observables, puede tener un mayor control sobre los sistemas distribuidos complejos que interactúan sobre los límites. En este capítulo, también probamos la observabilidad y su creciente necesidad en el entorno nativo de la nube. Con los tres pilares de la observabilidad y una introducción a los métodos preferidos de realizar el registro, la supervisión y el seguimiento de los sistemas distribuidos modernos en Azure, concluimos el capítulo con una revisión breve de la forma en que Azure ofrece sus soluciones integradas para realizar algunas tareas similares con Azure Monitor.

Cuando hablamos de límites y fronteras, nos enfrentamos al problema de la comunicación de servicio a servicio en este entorno que cambia de forma dinámica. Para resolver esto, presentamos la detección de servicios y la malla de servicios en el [Capítulo 7](#) como un medio para encontrar servicios y comunicarse eficazmente con ellos. También mostramos minuciosamente la necesidad de que las mallas de servicios y los proxies controlen de forma eficaz el entorno general en el que residen los microservicios. Presentamos CoreDNS como el DNS predeterminado, que reemplaza a kube-dns en el entorno de Kubernetes. También presentamos mallas de servicios que usan Istio, que utiliza Envoy como su plano de datos para obtener información útil sobre los servicios y proporciona el conjunto completo de características de la malla de servicios de Istio. Hacia el final del capítulo, presentamos Kiali, que proporciona una consola de administración completa en una malla de servicios basada en Istio.

En el [Capítulo 8](#), pasamos al problema de la administración de redes y políticas en entornos nativos de la nube. Le presentamos dos herramientas principales: Calico y Flannel, que sirven como la interfaz de red de los contenedores. También aclaramos cómo puede aplicar políticas en toda la pila mediante la administración de políticas abiertas. Revisamos el estándar CNI y de qué manera es la base para plataformas como Flannel y Cilium. Examinamos cómo se pueden usar esos sistemas para proporcionar conectividad y seguridad de red a su infraestructura de nube. También analizamos los mecanismos de políticas que no son de red para administrar la infraestructura de nube a través de Azure Policy, además de la política para aplicaciones a través de Open Policy Agent.

En el [Capítulo 9](#), analizamos cómo se percibe y diseña el almacenamiento en los entornos nativos de la nube mediante el uso de Vitess como una base de datos distribuida. También hablamos de Rook, que es un extenso orquestador de almacenamiento que puede ayudarlo a reducir la carga de administrar un sistema de almacenamiento distribuido simplemente realizando acciones como el autoescalado y la recuperación automática. En este capítulo, revisamos por qué es conveniente usar sistemas de datos

de nube nativa en la nube de Azure. Abarcamos cuatro sistemas: Vitess (relacional), Rook (blob), TiKV (clave-valor) y etcd (configuración clave-valor/detección de servicios). Estos sistemas son la piedra angular de una arquitectura nativa de la nube, que almacena y suministra datos en línea y ofrece una gran ventaja sobre la utilización de componentes PaaS.

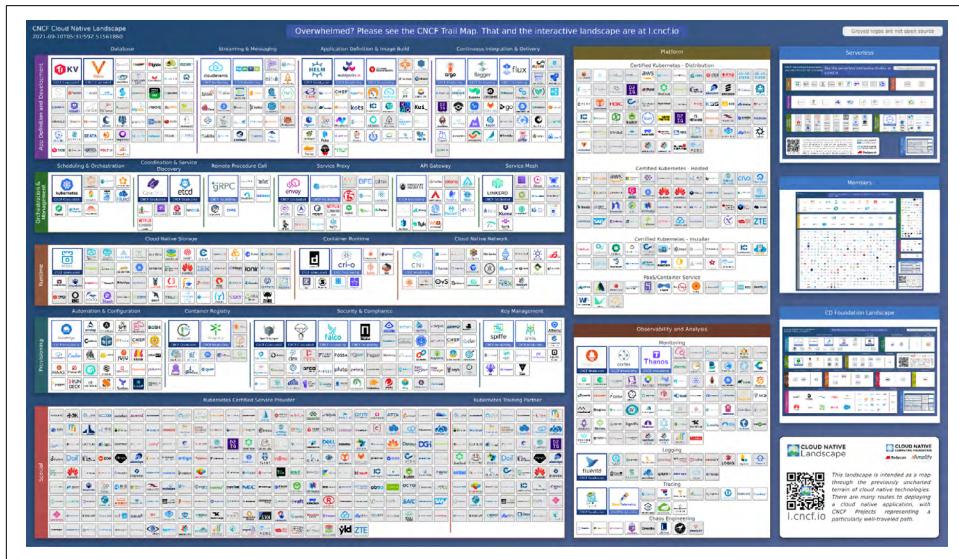
En el Capítulo 10, hablamos sobre cómo los servicios de mensajería y streaming se crean e implementan en entornos nativos de la nube mediante tecnologías tradicionales, como NATS, gRPC, Apache Kafka y RabbitMQ.

Por último, en el [Capítulo 11](#), exploramos cuatro formas diferentes en las que puede ejecutar funciones sin servidor en Azure. Primero examinamos Azure Function Apps y, luego, revisamos tres proyectos principales de CNCF: Knative, KEDA y OpenFaaS.

Esperamos que haya podido comprender bien cómo crear y usar la infraestructura nativa de la nube. Ahora tiene los conocimientos para crear y operar su propia infraestructura nativa de nube en Azure.

# ¿Qué sigue?

El panorama de **CNCF** es bastante grande y contiene una gran cantidad de proyectos que están siendo constantemente creados y apoyados por muchas grandes empresas (**Figura 12-1**). Estos proyectos se crean desde cero, teniendo en cuenta la escala y la confiabilidad.



*Figura 12-1. El panorama de CNCF*

Le recomendamos encarecidamente que explore otras tecnologías en el panorama de CNCF para crear su entorno de nube nativa de acuerdo con su caso de uso, y también lo animamos a explorar el entorno sin servidor de CNCF ([Figura 12-2](#)).

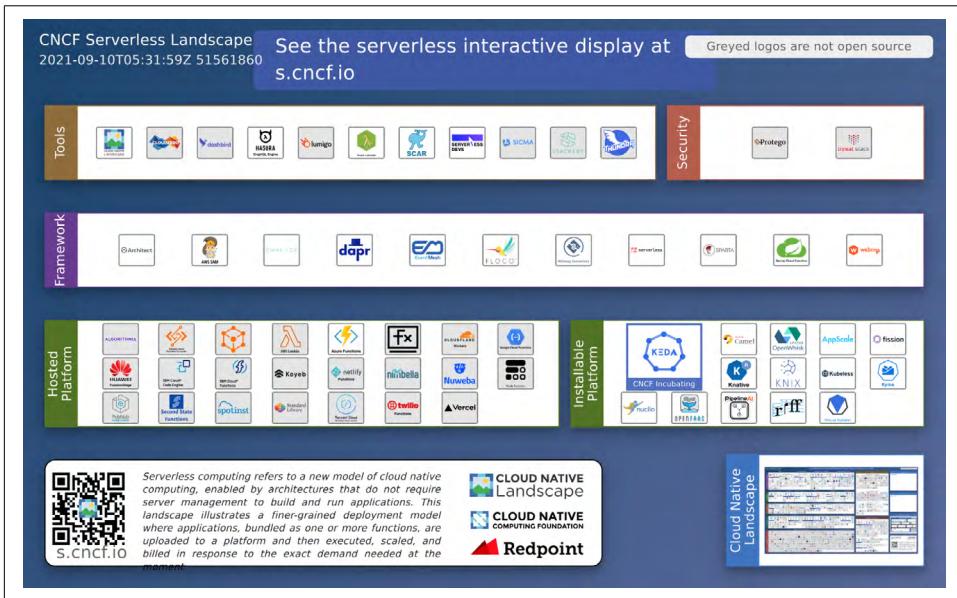


Figura 12-2. El entorno sin servidor de CNCF

El entorno sin servidor de **CNCF** resalta la informática sin servidor y el modelo de programación asociado, así como los formatos de mensajes. Aunque la informática sin servidor ha estado ampliamente disponible, queda mucho por hacer en el desarrollo de este enfoque. Puede explorar la informática sin servidor si su carga de trabajo es asincrónica, esporádica, sin estado y muy dinámica en términos de cambios en los requisitos empresariales. También puede usar varios servicios de Azure y ver cómo se pueden comparar con las tecnologías de CNCF.

La mayoría de estos proyectos son de open source y son las iniciativas de muchos ingenieros que contribuyen enérgicamente a la comunidad. Open source también aporta una gran cantidad de control de calidad al diseño general de estos proyectos. Esto significa que las tecnologías nativas de la nube se admiten bien en todo el mundo y se pueden adaptar fácilmente. También es importante tener en cuenta que muchas de estas tecnologías nativas de la nube son el resultado de problemas difíciles que debían resolverse de manera eficiente. Por lo tanto, es igualmente importante aventurarse, a veces, y hacer preguntas difíciles, con el fin de obtener una comprensión profunda de sus sistemas que, con el tiempo, lo llevarán a encontrar las soluciones correctas para problemas complejos.

El panorama nativo de la nube sigue creciendo y lo animamos a explorarlo aún más en su verdadero potencial. Le deseamos la mejor de las suertes.

---

# Índice

## A

- AAD (Azure Active Directory), 11  
acceso remoto, generación de archivos kubeconfig para, 110  
ACI (ver Azure Container Instances)  
actualizar un lanzamiento de Helm, 117  
acuerdos de nivel de servicio (SLA), 7  
administración de directivas, 189, 286  
    Azure Policy, 210-213  
    Open Policy Agent (OPA), 214-218  
    solución de directivas y seguridad de red de Calico, 193-200  
administración de pods, 78-85  
    comprobaciones de estado, 80  
    límites de recursos, 81  
    volúmenes, 82  
Administrador de controladores Kube, 70  
administrador de paquetes para Kubernetes (ver Helm)  
agentes de mesos, 42  
agentes de transferencia de mensajes (ver MTA)  
agentes, 234  
    y agrupación en clústeres, 240  
aislamiento proporcionado por los contenedores, 36  
AKS (ver Azure Kubernetes Service)  
Alertmanager (Prometheus), 126  
almacén de clave-valor transaccional (ver TiKV)  
almacén de datos (Calico), 194  
almacenamiento de blobs, 15, 23  
    creación de almacenamiento de blobs de Azure, 24  
almacenamiento de Hadoop Distributed File System (HDFS), 220  
almacenamiento de objetos, 236  
almacenamiento, 219-232, 286  
    almacén de clave-valor etcd, 71  
    bases de datos distribuidas, 219  
    (ver también bases de datos distribuidas)  
claves, 250  
creación de back-end de cuenta de almacenamiento para el clúster de Kubernetes en Azure, 101  
etcd, 229-231  
mensajes, 241  
objetos de almacenamiento en Kubernetes, 84  
opciones de Azure para, 220  
orquestador de almacenamiento de Rook para Kubernetes, 224-226  
TiKV (Titanium Key-Value), 226-229  
almacenes de clave-valor  
    etcd, almacén de clave-valor distribuido, 229  
    TiKV, 226  
alteración de la configuración, 10  
Amazon SQS, 234  
AMQP, 253  
    AMQP 1.0, Azure Service Bus, 254  
anotaciones (Kubernetes), 74  
Ansible, 31, 285  
    generar el archivo kubeconfig para el acceso remoto y la validación del clúster, 110  
    uso para buscar hosts para Prometheus, 134  
    uso para implementar y configurar los nodos de trabajo de Kubernetes, 109  
    uso para implementar y configurar los nodos del controlador de Kubernetes, 106-109  
Apache Kafka (ver Kafka)  
Apache Mesos (ver Mesos)  
API  
    Azure, entidad de servicio con la que interactuar, 202  
    habilitar el mantenimiento de entornos en la nube, 7  
LXC, orquestación de, 50  
    suministrado por OPA, 214

- TiKV, sin procesar y transaccional, 226
- API de REST
- agentes de Mesos impulsados por, 42
  - Kubernetes, 136
  - proporcionado por el demonio de LXD, 50
  - proporcionado por OPA, 214
- API sin procesar (TiKV), 226
- API transaccional (TiKV), 226
- aplicaciones contenedezadas, 35-66, 285
- beneficios del uso de contenedores, 35-37
    - aislamiento, 36
    - empaquetado e implementación, 37
    - seguridad, 36
  - componentes de la ejecución de un contenedor, 40-43
    - orquestradores de contenedores, 41
    - plataformas de software de contenedores, 41
  - Docker, 46-49
    - ejecución de Docker en Azure, 60-66
    - especificación OCI, 43-46
    - otras plataformas de contenedores, 49-50
    - primitivas de contenedores básicos, 37-40
  - aplicaciones desacopladas de la infraestructura, 10
  - aplicaciones nativas de la nube, 2
    - desacoplamiento de la infraestructura, 10
  - aprovisionadores, 29
  - archivo .dockernignore, 49
  - archivo Chart.yaml, 119
  - archivo custom-resources.yaml, 196
  - archivo de inventario (Ansible), 32
  - archivo kubeconfig, 78
    - descarga para clúster de AKS, 112
    - generar para el acceso remoto y la validación del clúster, 110
  - arquitectura de aplicación monolítica, 237
  - arquitectura de implementación sidecar para seguimiento distribuido, 159
  - autenticación
    - Nkey, 250
    - TLS, uso en NATS, 250
  - autenticación Nkey, 250
  - Azure
    - características de escalado automático en productos, 231
    - CNI con, 191
    - crear una cuenta, 12
    - etcd en, 230
    - opciones de bases de datos y almacenamiento, 220
    - servicios de mensajería, 234
  - Azure Cloud Shell (ver Cloud Shell)
  - Azure Container Instances (ACI), 60-64
- implementación, 61-64
- salvedades en la ejecución, 60
- Azure Container Registry, 55-59
- Azure Disk, 82
- Azure DNS, 171
- Azure Event Grid, 234, 261-263
- activación de funciones sin servidor, 266
  - biblioteca de cliente para Python, 262
  - criterios al elegir, 253
  - ecosistema de Event Grid, 261
  - implementación y uso de cuadrículas de eventos, 262
  - terminología diferente de otras plataformas, 261
- Azure Event Hubs, 234, 258-261
- administrar con Terraform, 259
  - arquitectura, 258
  - conceptos, 259
  - criterios al elegir, 253
  - Kafka y, 243
- Azure Files, 82
- Azure Function Apps, 268-271
- arquitectura, 269
  - creación de una aplicación de funciones, 270
  - lenguajes de programación compatibles, 268
  - planes, 269
- Azure Functions, crear una función, 270
- Azure Key Vault, 250
- Azure Kubernetes Service (AKS), 286
- compatibilidad con complementos de Azure VNet, 191
  - configuración de red que incluye Calico, 197
  - controlador de ingreso de Application Gateway, 92
  - crear y administrar un clúster de Kubernetes, 111
  - ejecución de cargas de trabajo con estado en, 220
  - implementación de Calico a través de la instalación de AKS, 196
  - instalación de Cilium en, 202
  - instalación de Istio en, 175
- Azure Monitor, 159, 286
- Azure Pipeline, 33
- Azure Policy, 189, 210-213
- crear su propia política, 212-213
  - funciones clave, 210
  - inicio rápido: 210-212
    - directivas integradas, 210
    - iniciativas, 211
    - modos de aplicación de políticas o efectos, 211
  - límites, 213
  - para Kubernetes, 213

- Azure Portal  
    búsqueda de Azure Container Registry, 55  
    CLI de Azure interactuando con, 13  
    creación de cuentas de Azure a través de, 12  
    creación de inquilinos de AAD con, 11  
    creación de un clúster de AKS, 111  
    crear una instancia de contenedor de Azure, 61  
    ícono de Cloud Shell en su cuenta, 18  
    usar para crear una aplicación de función, 270  
    verificar la creación de la imagen de la  
        máquina de Packer en, 30  
    verificar la creación del grupo de recursos de  
        Terraform, 22  
    visualización de implementación de Service  
        Bus, 257  
        visualización de la red virtual que se está  
            creando, 26
- Azure Service Bus, 234, 253-258  
    administrar con Terraform, 255-258  
    cola, 277-281  
    conceptos, 254-255  
        colas, temas y suscripciones, 254  
        namespaces, 254  
    creación del sistema de eventos con Service  
        Bus  
        criterios al elegir, 253  
    enviar/recibir mensajes en la cola en  
        Python, 258
- Azure Storage Explorer, 220
- Azure Virtual Network (VNet), complementos  
    que conectan contenedores con, 191
- azure\_sd\_config, 135
- AzureDisk, 84
- AzureFile, 83
- B**
- backend, 15
- back-ends de transporte (Flannel), 208
- bases de datos distribuidas, 219-224, 286  
    MySQL distribuida y con particiones, con  
        Vtess, 221-224  
    necesidad de arquitectura nativa de la nube,  
        219
- opciones de base de datos de azure, 220
- Bases de datos NoSQL  
    Cassandra, 224  
        operar bases de datos SQL como, 221  
    bases de datos SQL, operar de forma NoSQL, 221
- Bash, 18
- bibliotecas de cliente  
    para el seguimiento distribuido, 153  
    Prometheus, 126
- BIRD (Calico), 194
- bloques de servidores en CoreDNS, 167
- bucle de conciliación (administrador de  
    controladores Kube), 70
- C**
- Calico, 193-200  
    arquitectura básica, 194  
    beneficios de usar, 193  
    implementación de la política de seguridad,  
        198-200  
    implementación, 195-197  
        Instalación de calicoctl, 196  
        instalación manual de Calico en el clúster  
        de Kubernetes, 196  
    uso, 197  
        habilitar el plan de datos eBPF, 197
- calicoctl, 194  
    habilitar eBPF con, 198
- canalizaciones de CI/CD  
    Azure Function Apps y, 269  
    Azure Pipelines, 33
- capacidad de depuración, falta de con sin  
    servidor, 268
- capacidades de proceso, control de Linux de, 40
- capacidades, control de Linux para proceso o  
Cassandra, 224  
    implementación del clúster de Cassandra con  
        Rook, 225-226
- certificados  
    crear certificado autofirmado, 180  
    generación de certificados TLS para nodos de  
        controlador de Kubernetes con cfssl, 107  
    proteger el ingreso de Kubernetes con certificado  
        y clave TLS codificado Base64, 94  
    TLS, 250
- Certificados de SSL, 52
- cgroup, 40
- cgroups (grupos de control), 38  
    control de capacidades de Linux, 40
- CIDR (Enrutamiento de interdominios sin clases)  
    CIDR de la red de Flannel y pod, 208  
    seguridad de Cilium para, 200
- cifrado de datos del clúster de Kubernetes en  
    reposo, 107
- Cilium, 190, 200-207  
    implementación en clúster de  
        Kubernetes, 201-204  
        instalación de Cilium autoadministrada, 201  
        Instalación de Cilium de AKS, 202, 202  
        instalación de Hubble, 203  
    integración con su nube, 204-207  
        firewall del host, 204-206  
        observabilidad, 206

Citadel (Istiod), 174  
clases de almacenamiento, 82, 83  
Cloud Native Computing Foundation (CNCF), 4, 285  
CloudEvents, 244  
entorno de transmisión y mensajería, 238  
Entorno sin servidor, 266, 287  
orquestadores de contenedores, 41  
Panorama, 287  
proyectos de bases de datos, 219  
Cloud Shell, 17, 18  
    crear una entidad de servicio para su uso con Terraform, 18  
CloudEvents (CNCF), 244  
clústeres  
    agentes y, 240  
    de varios servidores NATS, 245  
clústeres (Kubernetes)  
    crear desde cero en Azure, 99-111  
        configuración del enrutamiento y la red del pod, 109  
        creación de direcciones IP públicas para el equilibrador de cargas, 102-104  
        creación de imágenes de máquinas para máquinas de trabajadores y controladores, 100  
    creación de instancias de trabajo y controlador, 104-106  
    creación del back-end de la cuenta de almacenamiento, 101  
    creación del grupo de recursos, 100  
    crear una red virtual de Azure, 102  
    generar el archivo kubeconfig para el acceso remoto y la validación del clúster, 110  
    implementación y configuración de nodos de controlador con Ansible, 106-109  
    implementación y configuración de nodos de trabajo con Ansible, 109  
    crear en Azure con AKS, 111  
    implementación de aplicaciones y servicios con Helm, 113-119  
        administración de lanzamientos de Helm, 117  
        componentes de Helm, 114  
        creación de gráficos para sus aplicaciones, 118  
        instalación y administración de Helm, 114-117  
Clusterrole, 208  
ClusterRoleBinding, 208  
CNCF (ver Cloud Native Computing Foundation)

CNI (interfaz de red de contenedores), 190-193, 286  
beneficios de usar, 191  
Cilium implementa su propio 200  
cómo funciona con Azure, 191  
complemento de CNI de Calico, 194  
ejemplo de configuración, 190  
primitivos administrados por, 190  
proyectos, 192  
cola de archivos de registro, Fluentd lee eventos de, 141  
cola de pub/sub basada en memoria, 237  
cola duradera en mensajería, 242  
    ejemplo de caso de uso de mensajería, ingestión y análisis de registro, 235-237  
    generación 1: sin colas, 235  
    generación 2: con colas de la nube y almacenamiento de objetos, 236  
    patrón de cola simple en mensajería, 242, 253  
colas, 234  
    en Azure Service Bus, 254, 255  
    envío/recepción de mensajes en la cola en Python, 258  
comando apply (kubectl), 80, 87  
    aplicación del manifiesto de trabajo, 97  
comando apply (Terraform), 14, 16, 24, 26  
comando autoscale (kubectl), 89  
comando cluster-info (kubectl), 77  
comando curl, 145  
comando delete pod (kubectl), 84  
comando describe (kubectl), 77  
    comprobación del proxy sidecar que se ejecuta en un pod, 178  
    describe deployment, 88  
    describir el estado de un trabajo, 97  
comando destroy (Terraform), 16, 27  
comando dig, 168, 170  
comando docker build, 48  
    uso de opciones, 49  
comando edit (kubectl), 87, 228  
comando get (kubectl), 77  
    get hpa, 89  
    get svc, 91  
comando get nodes (kubectl), 76  
    flag -o wide, 76  
comando init (Terraform), 16  
comando logs (kubectl), 78, 146  
comando plan (Terraform), 24  
comando run (kubectl), 79  
compilaciones de varias etapas de imágenes de contenedor, 49  
complejidad operativa, 3

complemento de caché (CoreDNS), 168  
complemento de errores (CoreDNS), 168  
complemento file (Fluentd), 141  
complemento filter\_geoiip (Fluentd), 143  
complemento filter\_record\_transformer (Fluentd), 142  
complemento filter\_stdout (Fluentd), 143  
complemento forward (Fluentd), 140, 141  
cómo escribir su primera función de OpenFaaS, 282-283  
funciones (sin servidor), 266  
implementación de la función de Knative Serving en Kubernetes, 273  
una gran cantidad de, 268  
complemento http (Fluentd), 140  
complemento in\_exec (Fluentd), 141  
complemento in\_syslog (Fluentd), 140  
complemento in\_tail (Fluentd), 141  
complemento in\_unix (Fluentd), 141  
complemento log (CoreDNS), 168  
complemento out\_copy (Fluentd), 142  
complemento out\_elasticsearch (Fluentd), 141  
complemento stdout (Fluentd), 142  
complementos de entrada (Fluentd), 140  
recuperación de registros de eventos de orígenes externos, 140  
complementos IPAM, CNI, 192  
complementos meta (CNI), 192  
complementos principales (CNI), 192  
comprobaciones de estado en Kubernetes, 80  
sondeos de inicio, 81  
sondeos de preparación, 81  
sondeos de vitalidad, 80  
comprobaciones de validación para el clúster de Kubernetes en Azure, 110  
comunicación entre procesos, 233  
confd (Calico), 194  
config (Helm), 114  
ConfigMap, 208  
configuración ipPools, 196  
consumidores (mensajería), 239  
contadores (Prometheus), 131  
containerd, 42  
contaminaciones (característica de programación de Kubernetes), 95  
contenedor sidecar, 72  
Contenedores Kata, 49  
contenedores, 43  
Azure Monitor, 159  
hogar de microservicios, 67  
contexto (en Kubernetes), 78  
control de acceso basado en roles (RBAC)  
roles en Flannel, 208  
uso en etcd, 231  
controlador de ingreso, 74  
Gateway de aplicación de Azure Kubernetes Service, configuración, 92  
controladores  
creación de instancias de controladores para el clúster de Kubernetes en Azure, 104-106  
implementación y configuración de nodos de controlador de Kubernetes con Ansible, 106-109  
KEDA, 276  
controladores de admisión (Kubernetes), 177  
copy on write (CoW), 40  
CoreDNS, 286  
complemento específico de Azure, 171  
instalación y configuración, 167-169  
introducción a, 165  
proceso de consulta, 166  
Corefile, 167  
corrección automática, etcd, 230  
creadores, 29  
crear registros en el mundo nativo de la nube, 138-149  
ejecución de Fluentd en Kubernetes, 146-149  
uso de Fluentd, 138-146  
CRI-O, 43  
cuenta de almacenamiento y blob, creación desde Terraform, 24  
cuentas de almacenamiento, 220  
cuotas de recursos, 74  
DaemonSets, 75  
CNI de Flannel, 208  
Fluentd, 147  
implementación en clúster de Kubernetes, 148  
uso en producción, 94  
versus ReplicaSets, 94  
desarrollo  
ciclos de desarrollo más rápidos sin servidor, 268  
impulsado por la observabilidad, 124

## D

desarrollos futuros, 287  
desinstalar un lanzamiento de Helm, 118  
detección de servicios, 68, 89, 134, 164-172, 286  
acerca de, 164  
Azure DNS, 171  
configuraciones de detección de servicios de Azure, 135  
etcd, 229  
en Kubernetes, con CoreDNS, 169-171  
instalación y configuración de CoreDNS, 167-169

- introducción a CoreDNS, 165
- DevOps, 8
- Azure DevOps y la infraestructura como código, 33
- direcciones IP
- CIDR de la red de pods, 208
  - creación de IP públicas para el equilibrador de carga en el clúster de Kubernetes en Azure, 102-104
  - recuperación con funciones sin servidor, 266
  - servicios en Kubernetes, 90
    - IP del clúster, puerto de nodo y equilibrador de carga en un nodo, 91
- directiva @include (Fluentd), 144
- directiva EXPOSE (Dockerfile), 47
- directiva filter (Fluentd), 142
  - complementos adicionales, 142
- directiva label (Fluentd), 143
- directiva match (Fluentd), 141
- directiva source (Fluentd), 140
  - parámetro @label, 144
- directiva system (Fluentd), 143
- discos administrados, 219
- discos de almacenamiento en Azure, 82
- disponibilidad
- de servicios PaaS, 221
  - etcd, 231
- DNS
- administración de DNS del clúster (ver CoreDNS)
  - asignación del servicio de Kubernetes al nombre DNS, 91
  - Azure DNS, 171
  - direcciones IP y nombres en un clúster, 170
  - limitar el DNS de egreso saliente a los servidores DNS públicos de Google con Cilium, 205
  - observabilidad del tráfico DNS con Cilium y Hubble, 206
- Docker Swarm, 41
- Docker, 42, 46-49
- almacenamiento de imágenes de Docker en un registro de contenedores, 59
  - creación de su primera imagen de Docker, 46-48
  - ejecución de todos los componentes de Jaeger en una sola imagen de contenedor, 155
  - ejecución en Azure, 60-66
    - ejecución de un motor de contenedores de Docker, 65-66
- uso de Azure Container Instances, 60-64
- imagen que contiene la aplicación sin servidor, 273
- motor de tiempo de ejecución, 43
- procedimientos recomendados para el uso, 48
- Dockerfiles, 46
  - crear, 47
  - definido, 48
- dominio de error en Vitess, 224
- dominio del clúster (Kubernetes), 170
- dominios, 170
  - (ver también CoreDNS; DNS)
    - en Azure Event Grid, 261
    - ingreso con varias rutas para, 93
- durabilidad de los mensajes, 241
- patrón de cola duradera, 242
- E**
- eBPF (Berkeley Packet Filter extendido)
- compatibilidad con Calico, 193
  - habilitación como plano de datos de Calico, 197
  - uso con versión del kernel posterior a 4.16, 198
  - uso por el proyecto Cilium, 200
- efectos (Azure Policy), 211
- ejemplo de bloque de recursos (Terraform), 15
- Elasticsearch
- escribir registros en el punto de conexión del clúster con el complemento Fluentd, 141
  - implementación con Helm en Kubernetes, 148
- empaquetado en contenedores, 36-38
- empaquetado, beneficios de los contenedores para, 37
- enlace, 70
- enrutamiento
- Calico, 194
  - CIDR, 208
  - CNI, 190
  - configuración de pods en el clúster de Kubernetes en Azure, 109
  - en ingreso de Kubernetes, 93
  - en Istio, 174, 174
  - en los espacios de nombres de Linux, 39
  - en mallas de servicios, 172, 173
  - Fluentd, 140, 143
  - Kiali en Istio, 180, 185
- entidad de servicio, crear para Terraform, 18
- entrega (mensaje), 241
- entrega de mensaje exactamente solo una vez, 241, 249
- entrega de mensajes al menos una vez, 241

- proporcionado por los servicios de mensajería de Azure, 253
- entrega de mensajes como máximo una vez, 241
- Envoy (plano de datos), 174, 286
- inserción automática de proxy sidecar, 177
- equilibradores de carga
- creación de IP públicas para el equilibrador de carga de Kubernetes en Azure, 102-104
  - IP del clúster, puerto de nodo y equilibrador de carga en un nodo de Kubernetes, 91
- equilibrio de carga
- automatización por orquestadores de contenedores, 68
  - controlador de ingreso en Kubernetes, 74 en Kubernetes, 68
  - HTTP, realizado por el objeto Ingress de Kubernetes, 92
- equilibrio de carga de Capa 7 en Kubernetes, 92
- equilibrio de carga HTTP en Kubernetes, 92
- Escalabilidad automática controlada por eventos y basada en Kubernetes (ver KEDA)
- escalado automático
- Azure, 231
  - enfoques de máquina virtual a, 234
  - etcd y, 230
  - KEDA, 276-281
  - OpenFaaS, 281, 283 sin servidor, 265, 268
- escalado horizontal con pods, 73
- escalado, 234
- (ver también escalado automático)
  - escalado horizontal de las colas de mensajes, 237
- Escalador automático horizontal de pod (HPA), 88, 230
- escaladores (KEDA), 276
- Especificación de OCI (Open Container Initiative), 43-46
- especificación de imagen, 44
  - especificación de tiempo de ejecución, 45
- especificación de tiempo de ejecución (OCI), 45
- estado
- definición de un contenedor, especificación de OCI, 45
  - determinar para el clúster de Kubernetes, 71 en Terraform, 15
  - estado de la red virtual cargado por Terraform en la cuenta de almacenamiento, 27
- etcd, 71, 229-231
- arquitectura operativa, 229
  - disponibilidad y seguridad, 231
  - escalado automático y corrección automática, 230
- plataforma de hardware, 230
- etiquetas
- configuración de herramienta de seguimiento Jaeger, 156
  - crear para imágenes de Docker, 48
- etiquetas (Kubernetes), 74
- etiquetado de pods, 87
  - istio-injection, 177
  - obtener información sobre kubectl, 78
- etiquetas (Prometheus), 126
- filtrar métricas con, 130
  - reetiquetado, 136
- exportadores (Prometheus), 126
- ExternalName, 91
- ## F
- faas-cli, uso para crear e implementar una función faasd, 283
- failureThreshold, 81
- Felix (Calico), 194
- Flannel, 207-210
- arquitectura de ejemplo, 210
  - implementación, 207
  - uso, 208
    - back-ends de transporte compatibles, 208
- Flask, 46, 66
- uso para crear una aplicación de Knative Serving, 272
- Fluentd, 138-146
- actuar como capa de registro, 138
  - directiva @include, 144
  - directiva filter, 142
  - directiva label, 143
  - directiva match, 141
  - directiva source, 140
  - directiva system, 143
  - implementación con DaemonSet, 95
  - instalación de td-agent, 139
- función irate, 130
- funcionalidad de informática perimetral, 267
- ## G
- Galley (Istiod), 174
- gateway (Kiali), 180
- GlobalNetworkPolicy (Calico), 199
- Grafana Surveyor, 252
- gráficos (Helm), 114
- cambiar los valores predeterminados del gráfico, 116
  - creación para sus aplicaciones, 118
  - editar archivos en el directorio de gráficos, 119
  - elementos en el directorio de gráficos, 118

- instalación en Kubernetes, 115  
revertir un gráfico implementado, 118
- grupos de administración, 12  
directiva de Azure Policy aplicada en, 211  
directiva de configuración de Azure Policy para, 189, 210
- grupos de consumidores 259, 260
- grupos de recursos, 12  
Azure Container Registry, 56  
crear con Terraform, 22  
crear, 100  
en Azure DNS, 171
- ## H
- Harbor, 51-54  
almacenamiento de imágenes de Docker en, 59  
configuración para la instalación, 52  
creación de una imagen de Packer para implementar Harbor, 53  
instalación, 51  
comprobación del instalador, 52
- HDInsight, 243
- Helm, 113-119
- herramienta cfssl, 107
- herramienta de agregación de registros Loki, 149
- herramienta de seguimiento Jaeger, 154  
instrumentar código utilizando normas de seguimiento y Jaeger, 154-159
- herramienta de seguimiento Zipkin, 154
- herramientas de IaC destacadas, 13  
Ansible, 31  
Packer, 29-31  
Terraform, 14-28
- histogramas (Prometheus), 133
- hosts  
hallazgos extraídos por Prometheus, 134-136  
instalación y configuración de Prometheus en, 127
- HTTPS  
exponer puntos de conexión a los servicios de Kubernetes en pods, 74  
uso con Harbor, 52, 54
- Hubble  
instalación, 203  
observabilidad con Cilium, 206
- ## I
- idempotencia  
infraestructura como código, 9  
Plantillas ARM, 28
- imagen de contenedor, 37  
(ver también imágenes (contenedor))
- especificación de imagen de OCI, 44
- imágenes (contenedor)  
almacenamiento de imágenes de Docker en un registro, 59  
creación de su primera imagen de Docker, 46-48  
especificación de imagen de OCI, 44
- imágenes de máquina  
compilar con Packer, 29-31  
creación de máquinas para máquinas de controladores y trabajadores en Kubernetes, 100
- implementación  
implementación de mensajería de NATS, 234  
análisis de la mensajería en la nube con, 244-253  
Arquitectura del protocolo NATS, 244-248  
implementación de NATS en Kubernetes, 251  
Persistencia de NATS con JetStream, 249  
Seguridad NATS, 249  
ventajas de NATS, 244  
capacidades de streaming, 238
- implementaciones  
automático, con Kubernetes, 68  
beneficios de los contenedores para, 37  
Objeto implementado en Kubernetes, 73, 84  
uso en entornos de producción, 86
- implementaciones sin tiempo de inactividad, 86
- IMS, 253
- informática sin servidor, 265-283, 287  
acerca de, 265  
Azure Function Apps, 268-271  
arquitectura de referencia de la función sin servidor, 269  
creación de una aplicación de funciones, 270  
desventajas potenciales, 268  
entorno sin servidor CNCF, 266, 287  
funciones sin servidor, 266  
KEDA, 276-281  
arquitectura, 276  
instalación en Kubernetes, 277-281
- OpenFaaS, 281-283  
arquitectura, 281  
escribir una función de OpenFaaS, 282-283  
instalación en Kubernetes, 281
- plataforma Knative, 272-276  
Arquitectura de Knative, 272  
instalación y ejecución de Knative Eventing en Kubernetes, 274-276  
instalación y ejecución de Knative Serving en Kubernetes, 272

ventajas de, 267  
infraestructura antifragilidad, 9  
infraestructura como código (IaC), 7-33, 285  
Azure DevOps y, 33  
beneficios de, 8, 9  
introducción a Azure y configuración del entorno, 11-13  
creación de cuenta de Azure, 12  
fundamentos de Azure y preparación del entorno, 11  
instalación de la CLI de Azure, 13  
infraestructura inmutable, 10, 29  
infraestructura nativa de la nube, 2  
adoptar con Azure, 4  
crear y ejecutar con Azure, 285  
filosofía, 10  
futuras implementaciones en, 287  
obtener información sobre las aplicaciones y la infraestructura, 286  
ingreso, 74  
con varias rutas para un dominio, 93  
objeto Ingress de Kubernetes, usando en producción, 92  
iniciativas (Azure Policy), 211  
initialDelaySeconds, 80  
inquilinos, 11  
instancia del controlador de ubicación (PD), 228  
instancias (en Prometheus), 127  
interfaz de la línea de comandos (CLI)  
    azure-cli, 255  
    calicoctl en Calico, 194  
    instalación de la CLI de Azure, 13  
    ver el grupo de recursos de Terraform desde, 23  
Internet de las Cosas (IOT), uso de la mensajería, 234  
intervalos, 151  
en seguimientos del servicio de reserva de películas, 157  
intervalo de solicitudes entrantes y salientes, 154  
Istio, 159, 286  
    administración de mallas de servicios con Kiali, 179-186  
    arquitectura, 175  
    introducción a, 174  
Istiod (plano de control), 174

**J**

JetStream (NATS), 244, 244  
persistencia con, 249

JSON  
    definiciones de Azure Policy, 210  
    Prometheus leyendo hosts del archivo inventory.json, 134

uso con Terraform, 14

**K**

Kafka, 234  
capacidades de streaming, 238  
descripción general, 243  
KEDA (escalabilidad automática controlada por eventos y basada en Kubernetes), 276-281  
arquitectura, 276  
instalación de KEDA en Kubernetes, 277-281  
Kiali (consola de administración para Istio), 179-186, 286  
explorar el panel, 183  
exponer a Internet, 180  
gateway, servicio virtual y regla de destino, 180  
instalación de Kiali en el clúster de Kubernetes, 179  
panel que muestra el tráfico entrante, 182  
panel que muestra la inserción automática de un proxy sidecar en el espacio de nombres predeterminado, 182  
registros, métricas y seguimientos proporcionados, 186  
UI, acceso, 181  
Kibans, 149  
kit de herramientas de infraestructura de clave pública/seguridad de la capa de transporte (PKI/TLS), 107  
Knative, 272-276  
arquitectura, 272  
instalación y ejecución de Knative Eventing en Kubernetes, 274-276  
instalación y ejecución de Knative Serving en Kubernetes, 272  
kube-apiserver, 70  
kubectl, 76-98  
comprobación de recursos creados para el gráfico de Tomcat en Helm, 116  
descargar e instalar Calico, 196  
información general y comandos del clúster, 76  
instalación de archivo binario en OS X y Linux, 108  
instalación de Flannel, 207  
instalación mediante Hubble, 203  
kube-dns versus CoreDNS, 165  
kubelets, 42, 71  
extracción de nodos kubelet con Prometheus, 137  
kube-proxy, 71  
desactivar la implementación de, 198  
Kubernetes en producción, 85-98  
    DaemonSets, 94  
    Escalador automático horizontal de pod

- (HPA), 88  
objeto Ingress, 92  
objetos Deployment, 86  
ReplicaSets, 85  
trabajos, 96
- Kubernetes, 4, 41, 67-98, 285  
Azure Policy para, 213  
características proporcionadas de inmediato, 68  
CNI de Flannel, 207-210  
componentes, 69-72  
    nodos de trabajo, 71  
    plano de control, 70  
controladores de admisión, 177  
CoreDNS reemplaza kube-dns, 165  
creación de un clúster en Azure, 99-120  
    creación de un clúster desde cero, 99-111  
    implementación de aplicaciones y servicios con Helm, 113-119  
    uso de Azure Kubernetes Service, 111  
detección de servicios con CoreDNS, 169-171  
etcd utilizado como back-end de detección de servicios en, 229  
Fluentd en, 146-149  
implementación de Calico a través de la instalación de AKS, 196  
implementación de NATS en, 251  
implementación de Open Policy Agent en, 215  
implementación de TiKV en, 228-229  
implementación de Vitess en, 223-224  
instalación de Istio en AKS, 175  
instalación de Kiali en el clúster, 179  
instalación de OpenFaaS en, 281  
instalación manual de Calico en el clúster de Kubernetes, 196  
instalación y ejecución de Knative Eventing en, 274-276  
instalación y ejecución de Knative Serving en, 272  
instalar KEDA en, 277-281  
Interfaz de tiempo de ejecución de contenedores, implementación de CRI-O de, 43  
numerónimo K8s, 68  
objetos del servidor API, 72-76  
observar, operar y administrar clústeres con kubectl, 76-98  
    Kubernetes en producción, 85-98  
orquestador de almacenamiento Rook, 224-226  
plataforma Knative, 272  
Prometheus en, roles de detección de servicios, 136-138  
recursos para el aprendizaje adicional, 75  
servicio OpenFaaS de funciones basada en eventos para, 281
- L**
- la Capa 3 y la capa 4 permiten reglas (Cilium), 204  
    habilitación de la visibilidad, 206
- lanzamientos (Helm), 114  
    administración, 117  
        actualizar un lanzamiento, 117  
        comprobación de un lanzamiento, 117  
        desinstalar un lanzamiento, 118  
        revertir un lanzamiento, 118
- lenguaje CRuby (Fluentd), 138  
lenguaje de Hashicorp (HCL), 14  
lenguaje de programación Erlang/OTP, 243  
lenguaje de programación Ruby, implementación C de, 138  
lenguaje específico del dominio, HCL, 14  
lenguaje Rego, 214  
limitación de velocidad, 153  
límites de recursos (Kubernetes), 81
- Linux
- Etiquetas de SELinux, 37  
    imagen de la máquina, compilación en Azure con Packer, 29  
    primitivos de contenedores, 37-40  
    reducir el acceso del contenedor a las API del kernel, 36  
    soporte de Calico para, 193
- LXC (Contenedores Linux), 49
- LXD (software de administración de contenedores), 50
- malla de servicios, 172-187, 286  
    arquitectura general, 172  
    definido, 164  
    inserción automática de proxy sidecar, 177  
    instalación de Istio en AKS, 175  
    introducción a Istio, 174  
    Istio, administración con Kiali, 179-186  
    plano de control, 173  
    plano de datos, 173
- M**
- ManagedImageId, 101  
manifestos (pod), 79  
    configurar sondeos de preparación y vitalidad, 81  
    límites de recursos en, 81
- manuales (Ansible), 31
- máquinas virtuales
- Azure Monitor para, 159  
    Azure, soporte para complementos VNet de Azure, 191
- MariaDB, Azure Database para, 220
- medidores (Prometheus), 132

- memoria
- l límite máximo en manifiesto de pod, 81
  - t técnica de administración, copy on write, 40
- mensajería, 233-263, 287
- c conceptos básicos de las plataformas de mensajería, 238
  - c conceptos básicos, 238-242
  - a agentes y clústeres, 240
  - d durabilidad y persistencia, 241
  - e entrega de mensajes, 241
  - p productores y consumidores, 239
  - s seguridad, 242
- d descripción general de las plataformas populares de mensajería nativa de la nube
- A Apache Kafka, 243
  - C CNCF CloudEvents, 244
  - p plataformas, 243-244
  - R RabbitMQ, 243
- e ejemplo de caso de uso, ingestión y análisis de registro, 235-237
- g generación 1: sin colas, 235
  - g generación 2: con colas de la nube y almacenamiento de objetos, 236
  - g generación 3: con cola pub/sub basada en la memoria, 237
- m mensajería en la nube con NATS, 244-253
- n necesidad de, 233-235
- r razones para implementar la mensajería, 234
  - p patrones comunes, 242-243
    - c cola duradera, 242
    - c cola simple, 242
    - p publicar y suscribir, 242
- S Servicios de mensajería de Azure, 253
- A Azure Event Grid, 261-263
  - A Azure Event Hubs, 258-261
  - A Azure Service Bus, 253-258
  - e elegir entre, 253
- M Mesos, 41
- m métricas, 122
- a adaptador de métricas en KEDA, 276
  - A Azure Monitor, 160
  - d desde el punto de conexión de supervisión de NATS, 248
  - m métricas de Cilium en el tráfico de DNS y HTTP, 206
  - P Prometheus con la malla de servicios de Istio, 182
  - s supervisión con Prometheus en la nube
    - C Componentes y arquitectura de Prometheus, 125-127
    - e encontrar hosts, 134-136
    - i instalación y configuración de
- P Prometheus, 127
- i instrumentación de aplicaciones, 130-134
  - m mundo nativo, 125-138
  - n node\_exporter, 129
  - P Prometheus en Kubernetes, 136-138
- microservicios 67
- c cambiar de arquitectura monolítica a, 10
  - c comunicación de manejo de malla de servicio
  - c comunicaciones entre, enruteamiento y en ingesta y análisis de registro, ejemplo de caso de uso de mensajería, 237
  - e entre, 172
  - c malla de servicios de Istio simplifica la comunicación entre, 174
  - s seguridad, 163
  - v ventajas de, 163
- m modos de aplicación de políticas de red (Cilium), 204
- módulos (Terraform), 15
- MTAs (agentes de transferencia de mensajes), 239
- a agrupación en clústeres, 240
- muestreo sensible al contexto, 153
- muestreo, 153
- MutatingAdmissionController, 177
- MySQL
- A Azure Database for, 220
  - d distribuida y con particiones, Vitess, 221-224
- namespaces, 39
- c componentes de Istio en AKS, 176
  - e en Azure Service Bus, 254, 255
  - e en Kubernetes, 74
  - o obtener información con kubectl, 77
- N
- n navegador de expresiones (Prometheus), 128
- NetworkPolicy (Calico), 199
- NFS (Network File System), 224
- nodo de control, 32
- nodos
- c creación de nodos de instancias de controladores y trabajo para el clúster de Kubernetes en Azure, 104-106
  - n node\_exporter (Prometheus), 129
  - r rol de nodo, 136
- nodos administrados, 32
- nodos de trabajo (Kubernetes), 71
- c creación de un clúster en Azure, 104-106
  - i implementación y configuración con Ansible, 109
- nombre de la métrica (Prometheus), 126
- nombre del registro (Azure Container Registry), 56

- nube  
desafíos en, 2  
recorrido a, 1  
observabilidad, 121-161, 286  
con Cilium, 200, 206  
crear registros en el mundo nativo de la nube, 138-149  
crear registros con Fluentd, 138-146  
Fluentd en Kubernetes, 146-149  
desarrollo impulsado por, 124  
introducción a, 121-124  
pilares de, registros, métricas y seguimientos, 122  
ir más allá de los tres pilares, 123  
métricas de supervisión con Prometheus en Componentes y arquitectura de Prometheus, 125-127  
encontrar hosts, 134-136  
instalación y configuración de Prometheus, 127  
instrumentación de aplicaciones, 130-134  
mundo nativo de la nube, 125-138  
node\_exporter, 129  
Prometheus en Kubernetes, 136-138  
seguimiento distribuido en el mundo nativo de la nube, 150-159  
arquitectura del sistema de seguimiento general y ensamblado de seguimiento, 153  
conceptos clave del seguimiento, 151-153  
normas de seguimiento, herramientas e instrumentación de código, 154-159  
superconjunto de supervisión, 123  
uso de Azure Monitor, 159  
uso de Kiali para mallas de servicios de Istio, 179  
Open Policy Agent (OPA), 214-218  
API clave transmitidas por, 214  
implementación de la directiva con, 216-218  
implementación en Kubernetes, 215  
integraciones, página del ecosistema de OPA, 215
- 0**
- OpenCensus, 154  
OpenFaaS, 281-283  
arquitectura, 281  
escriba su primera función, 282-283  
instalación en Kubernetes, 281  
OpenSSL, 250  
OpenTelemetry, 154  
instrumentar código con, 157
- OpenTracing, 154  
instrumentar código de la aplicación con, 155  
operación asíncrona (mensajería), 235  
operador (Kiali), 180  
organización de contenedores, 68  
orquestador de almacenamiento Rook, 224-226  
arquitectura, 224  
implementación en Kubernetes, 225-226  
orquestadores (contenedores), 41, 67  
(ver también Kubernetes)  
orquestadores de contenedores, 40  
(ver también Kubernetes; orquestadores)  
Packer, 29-31, 285  
creación de imágenes de instancias de controlador y trabajo, 100  
creación de una imagen de Linux en Azure, 29  
creación de una imagen para implementar Harbor, 53  
instalación de Docker CE a través de la imagen de Packer, 65  
instalación, 29
- P**
- parámetro storage, modificar para el clúster de TiKV, 228  
pares de claves, generar para la autenticación de Nkey, 250  
partición lógica, 74  
particionamiento  
características de particionamiento automático de TiKV, 226  
soporte de Vitess para, 221  
particiones (mensaje), 259  
patrones de entrega de unidifusión, multidifusión y difusión, 240  
patrones de publicación/suscripción  
(pub/sub), 234, 242  
cola de pub/sub basada en memoria, 237  
compatible con Azure Service Bus, 253  
en NATS, 245  
periodSeconds, 80  
PersistenceVolumeClaims (PVCs), 82, 84  
persistencia (mensajes), 241  
NATS JetStream, 246, 249  
protocolo pub/sub y, 244  
Pilot (Istiod), 174  
PKI/TLS toolkit, 107  
plan (Terraform), 14, 16  
plano de control (Kubernetes), 70  
plano de control (malla de servicios), 173  
Istiod, 174  
plano de datos (malla de servicios), 173

Envoy, uso por parte de Istio, 174  
plantillas  
    de los proveedores de nube para la implementación de IaC, 8  
    en Kubernetes ReplicaSets, 85  
    Packer, 29  
    Plantillas de Terraform y ARM, 28  
plantillas de Azure Resource Manager (ARM), 8  
    Terraform y, 28  
plataforma como servicio (PaaS), 2  
    Azure, 11  
    Docker, 46  
    servicios de almacenamiento y base de datos de Azure, 220  
plataforma de contenedor  
    Contenedores Kata, 49  
    Docker, 46-49  
    LXC y LXD, 49  
    plataformas comunes, 42  
plugins de salida (Fluentd), 141  
pods Nginx, 79  
Pods, 72  
    administración con kubectl, 78-85  
        creación de pods con el comando run, 79  
        creación de pods con la sintaxis declarativa en manifiestos de pod, 79  
    configuración de la red de pods y el enrutamiento para el clúster de Kubernetes en Azure, 109  
    disponibilidad durante las actualizaciones, 88  
    inserción automática de proxy sidecar, 178  
        sidecar proxy de Istio en, 176  
PostgreSQL, Azure Database for, 220  
PowerShell, 18  
productores y consumidores (mensajería), 239  
    patrón de cola simple, 242  
productores, 259  
Programador Kube, 70  
programadores, 67  
    características de programación en Kubernetes, 95  
    contaminaciones y tolerancias, 95  
    Programador Kube, 70  
programas de Berkeley Packet Filter (BPF), 40  
    (ver también eBPF)  
Prometheus, 125-138  
    componentes y arquitectura, 125-127  
        trabajos e instancias, 127  
    encontrar hosts, 134-136  
        uso de Ansible, 134  
        uso de azure\_sd\_config, 135  
        uso del archivo YAML o JSON, 134  
instalación con Istio, 182  
instalación y configuración, 127  
instrumentación de aplicaciones, 130-134  
    contadores, 131  
    histogramas, 133  
    medidores, 132  
    resumen, 133  
node\_exporter, 129  
    ejecución en Kubernetes, detección de servicios  
        rol de ingreso, 138  
        rol de nodo, 136  
        rol de pod, 137  
        rol de puntos de conexión, 137  
        rol de servicio, 137  
        roles, 136-138  
PromQL, 128  
propagación de contexto distribuido, 152  
propagación del contexto, 152  
protocolo de correo SMTP, ejemplo del sistema de colas, 240  
protocolo gRPC (Google Remote Procedure Call), 227  
protocolo Raft, 226, 227  
    uso por etcd, 229  
protocolos y patrones de mensajería, 253  
proveedor de recursos Microsoft.PolicyInsights, 211  
proveedores (Terraform), 14  
    ejemplo de bloque de proveedor, 15  
proxies sidecar (malla de servicios), 172, 172  
    inserción automática de proxy Envoy, 177  
proyectos open source (CNCF), 288  
puertos  
    servicios en Kubernetes, 90  
    mostrar la asignación de puertos, 91  
    servidor de zona DNS, 167  
punto de conexión TCP para Fluentd, 140  
puntos de conexión HTTP, definición y exposición de métricas mediante, 130  
puntos de conexión privados, 58  
pushgateway (Prometheus), 126  
Python, 32  
    biblioteca de Flask, 272  
    cola de Azure Service Bus, envío/recepción de mensajes, 258  
    configuración del entorno de Python para probar  
    Implementación KEDA, 279  
    paquete nats-py, 252  
    productor que usa el paquete de Python azure-eventgrid, 262

- RabbitMQ, 234  
capacidades de streaming, 238  
descripción general, 243  
recuperación automática, 68  
bucle de conciliación como fuerza impulsora en Kubernetes, 70  
recurso personalizado (Kiali), 180  
redes virtuales, 64  
CNI trabajando con Azure, 191  
creación de la red virtual de Azure a partir de Terraform, 25-27  
creación de red virtual de Azure para hospedar el clúster de Kubernetes, 102  
DNS en, 171  
redes, 189-218, 286  
CNI (interfaz de red de contenedores), 190-193  
beneficios de usar, 191  
cómo funciona con Azure, 191  
proyectos CNI, 192  
configuración de pods en el clúster de Kubernetes en Azure, 109  
creación de la red virtual de Azure a partir de Terraform, 25,102  
herramienta de redes de Cilium, 200-207  
implementación de Cilium, 201-204  
integraciones de red para Knative Serving, 272  
sistema de tejido de red de Capa 3 de OSI  
Flannel para Kubernetes, 207-210  
solución de directivas y seguridad de red de Calico, 193-200  
implementación de Calico, 195-197  
implementación de la directiva de seguridad de Calico, 198-200  
uso de Calico, 197-198
- R**
- redireccionamiento de puerto, 148  
redireccionamiento de puertos localmente con implementación de Kibana, 149  
Redis, 124, 157  
Azure Cache for, 220  
reetiquetado (en Prometheus), 136  
regiones, 12  
elegir región para Azure Container Registry, 56  
registro de servicios, 164  
registro en el nivel del clúster, 146  
registro, 286  
análisis de registro y consultas de registro con Azure Monitor, 160  
ejecución del contenedor de registros en Kubernetes, 95  
registros de contenedores, 50-59  
almacenamiento de imágenes de Docker en, 59  
almacenamiento seguro de imágenes con Harbor, 51-54  
productos nativos de la nube actual, 50  
Registros de servicios en un clúster, 170  
registros DNS SRV, 231  
registros, 122  
configuración de herramienta de seguimiento Jaeger, 156  
ingesta y análisis de registro, ejemplo de caso de uso de mensajería, 235-237  
ver para CoreDNS, 168  
regla de destino (kiali), 180  
reglas de visibilidad de Capa 7 en Cilium, 205  
rendimiento  
de servicios PaaS, 221  
funciones sin servidor y, 266  
mejora con la mensajería, 234  
rendimiento, mejora con la mensajería, 234  
replicación  
características de TiKV para, 226  
modificar el parámetro replicas para el clúster de TiKV, 228  
replicación geográfica en TiKV, 226  
réplicas, número para crear en ReplicaSets de Kubernetes, 85  
ReplicaSets, 73  
Daemonsets versus, 94  
Kubernetes en producción, 85  
manejo con objetos Deployment, 87  
repositorio de GitHub para este libro, 20  
repositorio de gráficos e interacción con el cliente, 113  
administración de versiones, 117  
componentes, 114  
creación de gráficos para sus aplicaciones, 118  
implementación de Elasticsearch en Azure, 148  
implementación de Kibana en Kubernetes, 149  
instalación de FaaS en Kubernetes, 281  
instalación de Istio en AKS, 175  
instalación de KEDA en Kubernetes, 277  
instalación de Kiali en Kubernetes, 179  
instalación de NATS en Kubernetes, 251  
instalación de TiKV en Kubernetes, 228  
instalación y administración, 114-117  
búsqueda de repositorios de Helm, 115  
cambiar los valores predeterminados del gráfico, 116  
instalación de gráfico de Helm en

Kubernetes, 115  
repositorio de PingCap, 228  
repositorios (Helm), 114  
agregar un repositorio de gráficos, 114  
búsqueda, 115  
resiliencia, aumentar con la mensajería, 234  
resumen (Prometheus), 133  
revertir un lanzamiento de Helm, 118  
 RocksDB, 227  
rol de ingreso, 138  
rol de pod, 137  
rol de puntos de conexión, 137  
rol de servicio, 137

**S**

Seccomp (SECure COMPuting), 40  
Seccomp-BPF, 40  
secretos  
especificación del soporte de TLS para el  
ingreso de Kubernetes con objeto secreto, 94  
parejas de claves almacenados como secreto de  
Kubernetes, 250  
seguimiento, distribuido, en el mundo nativo de  
la nube, 150-159, 286  
arquitectura del sistema de seguimiento  
general y ensamblado de seguimiento, 153  
conceptos clave del seguimiento, 151-153  
intervalos, 151  
muestreo, 153  
propagación del contexto, 152  
consumo de datos de seguimiento con Azure  
Monitor, 160  
normas de seguimiento, herramientas e  
instrumentación de código, 154-159  
seguimientos, 123  
definido, 152  
montaje, 154  
seguridad  
directiva de seguridad de Calico,  
implementación 198-200  
herramienta de redes de Cilium, 200  
mensajería, 242  
NATS, 249  
Autenticación basada en Nkey, 250  
autenticación TLS, 250  
proporcionado por contenedores, 36  
TLS y RBAC de etcd, 231  
selector de etiquetas, 74  
selectores (etiqueta), 74  
en ReplicaSets, 85  
serie temporal, datos de Prometheus almacenados  
como, 126

servicio virtual (Kiali), 180  
servicios  
cuenta de servicio de Kubernetes para Flannel,  
208  
objeto Service en Kubernetes, 73  
uso en entornos de producción, 89  
sin servidor, definición en un archivo, 273  
servidor API  
kube-apiserver, 70  
trabajar con objetos de Kubernetes, 72-76  
anotaciones, 74  
controlador de ingreso, 74  
DaemonSets, 75  
etiquetas y selectores, 74  
implementaciones, 73  
namespaces, 74  
Pods, 72  
ReplicaSets, 73  
servicios, 73  
StatefulSets, 75  
Trabajos, 75  
servidor Tomcat, implementación con Helm, 116  
servidor web nginx  
implementación de la nueva versión, 87  
uso con Azure Container Instances, 64  
servidores  
NATS  
agrupación en clústeres, 245  
supervisión, 246  
uso del servidor de Docker NATS, 246  
Servidor de Prometheus, 126  
shells  
Bash y PowerShell, 18  
Cloud Shell, 17  
sin servidor, 282  
sintaxis declarativa  
CLI de Azure para crear un script de forma  
declarativa  
en Kubernetes, 68, 73, 74  
infraestructura, 13  
lenguaje declarativo, Rego, 214  
Plantillas ARM, 28  
uso de los manifiestos de pod para crear pods  
en Kubernetes, 79  
uso por Terraform, 14  
sistema de almacenamiento de Ceph, 224  
sistemas de eventos, Knative Eventing, 274-276  
sistemas operativos, contenedores que se ejecutan  
en, 35, 43  
SKU  
Azure Container Registry, 57  
ejecución de etcd en SKU de Azure, 230

software como servicio (SaaS), 2, 2  
software de contenedor, 41  
Solicitudes HTTP GET, 266  
sondeos de inicio, 81  
sondeos de preparación, 81  
sondeos de vitalidad, 80  
sondeos, 80  
(ver también comprobaciones de estado en Kubernetes)  
soporte de protocolo MQTT, NATS Jetstream, 244  
SSH  
actualización de ruta para la ubicación de la clave SSH para máquinas de trabajo y controladores, 104  
generación de pares de claves SSH, 105  
suministro de acceso a las máquinas mediante OPA, 216-218  
state  
ejecución de cargas de trabajo con estado en AKS, 220  
StatefulSets, 75  
streaming, 287  
mensajería versus, 238  
servicio de streaming de NATS (Stan), instalación en Kubernetes, 251  
Streams (NATS), 244  
supervisión, 286  
de la infraestructura y las aplicaciones nativas de la nube modernas, 125  
(ver también Prometheus)  
de servidores NATS, 246  
observabilidad como superconjunto de, 123  
suscripciones, 12  
Azure Policy aplicada en, 211  
en Azure Service Bus, 255  
para Azure Container Registry, 56  
syscalls, filtrado, 40  
taller de rubber-docker, 46  
td-agent (Fluentd)  
cambiar configuración, 145  
instalación, 139  
temas  
en Azure Event Grid, 261  
en Azure Service Bus, 254, 255  
Terraform, 14-28, 285  
administración de Azure Event Hubs con, 259  
administración de Azure Service Bus con, 255-258  
comandos comunes, 17  
configuración básica de infraestructura y uso con, 20  
configuración de la red de pods y el enrutamiento para el clúster de Kubernetes en Azure, 109  
configuración del acceso a la cuenta de Azure, 18  
creación de back-end de cuenta de almacenamiento para el clúster de Kubernetes en Azure, 101  
creación de instancias de controladores y trabajo para el clúster de Kubernetes en Azure, 104-106  
creación de IP públicas para el equilibrador de carga de Kubernetes en Azure, 102-104  
creación de un clúster de AKS, 111  
creación del grupo de recursos para el clúster de Kubernetes en Azure, 100  
crear una red virtual de Azure, 102  
explorar la infraestructura de Azure con, 23  
creación de almacenamiento de blobs de Azure, 24  
creación de una red virtual e instancias de Azure, 25  
Plantillas ARM, 28  
flujo de trabajo, 16  
implementación de Azure Container Registry a través de, 58  
implementación de cuadrículas de eventos de Azure, 262  
instalación, 17  
terminología básica, 14

**T**

tiempos de ejecución (contenedor), 42  
tiempos de ejecución de contenedores, 42  
TiKV (Titanium Key-Value), 226-229  
arquitectura, 226  
implementación en Kubernetes, 228-229  
tipo NodePort, 90  
tipos de métricas quad-core, 126  
TLS (Seguridad de la capa de transporte)  
autenticación, uso en NATS, 250  
especificar mediante la creación de objeto Secret de Kubernetes, 94  
inspección de conexiones cifradas con TLS con Cilium, 207  
kit de herramientas de PKI/TLS para certificados de TLS, 107  
uso de cifrado TLS en etcd, 231  
tolerancias (característica de programación de Kubernetes), 95  
Trabajos (Kubernetes), 75  
objeto Job, creación y administración de pods, 96

- trabajos (Prometheus), 127  
tráfico de ICMP, permitir con Calico, 199  
Typha (Calico), 194
- U**  
unidades de estado sólido (SSD), 230  
utilidad mkcert, 250
- V**  
variable ARM\_ACCESS\_KEY, 101  
variables de entorno, configuración para Terraform en la máquina local, 19  
virtualización de infraestructura, 191  
virtualization, 1  
    CNI habilita la virtualización de infraestructura verdadera, 191  
vista de mapa de aplicaciones (Azure Monitor), 160  
vista diagnóstico de transacciones (Azure Monitor), 160  
Visual Studio Code, extensión de Azure Functions, 270  
Vitess, 221-224  
    arquitectura, 222  
    beneficios de usar, 221  
    implementación en Kubernetes, 223-224  
volúmenes (Kubernetes), 82  
    clase de almacenamiento, dinámico, 83  
    PersistentVolumeClaims (PVCs), 82  
    volúmenes persistentes, estáticos, 83  
volúmenes persistentes (PV), 82, 84  
    estático, 83  
vtcld, 222  
vtctl, 222  
VTGate, 222  
VTTablet, 222  
VXLAN  
    uso con Calico, 197
- uso con Flannel, 208
- W**  
Windows  
    redes, Azure CNI y, 191  
    soporte de Calico para el plano de datos de Windows HNS, 193
- X**  
XDP, 198
- Y**  
YAML, 32  
    Prometheus lee hosts desde el archivo static, 134
- Z**  
zonas (DNS), 165, 167  
    Azure DNS, 171

## Acerca de los autores

---

**Nishant Singh** es ingeniero senior de fiabilidad del sitio en LinkedIn, donde trabaja para mejorar la seguridad del sitio centrándose en reducir el tiempo medio de detección (MTTD) y el tiempo medio de respuesta (MTTR) a los incidentes. Antes de unirse a LinkedIn, trabajó en Paytm y Gemalto como ingeniero de DevOps, donde dedicaba su tiempo a la creación de soluciones personalizadas para los clientes y la administración y el mantenimiento de servicios a través de la nube pública. Nishant tiene un gran interés en la ingeniería de confiabilidad del sitio y en la creación de sistemas distribuidos.

**Michael Kehoe** es ingeniero sénior de seguridad del personal en Confluent. Antes de esto, trabajó en respuesta ante incidentes, recuperación ante desastres, ingeniería de visibilidad y principios de confiabilidad como ingeniero sénior de confiabilidad de sitios del personal en LinkedIn. Durante el tiempo que trabajó en LinkedIn, lideró los esfuerzos de la empresa por automatizar la migración a Microsoft Azure. Michael se especializa en la mantención de la infraestructura de sistemas grandes, como lo demuestra su trabajo en LinkedIn (aplicaciones, automatización e infraestructura) y en la Universidad de Queensland (redes). También ha dedicado tiempo a crear pequeños satélites en la NASA y a escribir software para entornos termales en Rio Tinto.

## Colofón

---

El animal en la portada de *Infraestructura nativa de la nube con Azure* es un arrendajo de Steller (*Cyanocitta stelleri*), un arrendajo crestado que se encuentra en Norteamérica y Centroamérica. El arrendajo de Steller suele habitar en los bosques de coníferas, pero también se puede encontrar en bosques caducifolios.

Nombrado en honor a Georg Wilhelm Steller, el naturalista alemán que hizo el primer registro de esta ave en 1741, este arrendajo azul es diferente del arrendajo azul de Norteamérica y su característica distintiva es una cresta negra espectacular.

Los arrendajos de Steller son omnívoros, su dieta consiste principalmente en nueces, semillas de pino, bellotas, bayas y otras frutas, que complementa con insectos, pequeños roedores y huevos.

Si bien el estado de conservación del arrendajo de Steller es de “Preocupación menor” (muchos de los animales que aparecen en las portadas de O'Reilly son especies en peligro de extinción), todos los animales son importantes para el mundo.

La ilustración de la portada es de Karen Montgomery y se basa en un antiguo grabado de línea del libro *Royal Natural History de Lydekker*. Las fuentes de la portada son Gilroy Semibold y Guardian Sans. La fuente del texto es Adobe Minion Pro, la fuente del encabezado es Adobe Myriad Condensed y la fuente del código es Dalton Maag's Ubuntu Mono.



O'REILLY®

## Aprenda de los expertos. Conviértase en uno.

Libros | Cursos en línea en vivo

Respuestas instantáneas | Eventos virtuales

Videos | Aprendizaje interactivo

Comience en [oreilly.com](https://oreilly.com).