Therefore, we will create new Argo CD applications to deploy it. We will create an Argo CD application for **istio-base**, **istiod**, and **ingress**. The following YAML describes `istio-base`:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: istio-base
  namespace: argo
spec:
  project: default
  source:
    chart: base
    repoURL: https://istio-release.storage.googleapis.com/charts
    targetRevision: 1.19.1
    helm:
      releaseName: istio-base
  destination:
    server: "https://kubernetes.default.svc"
    namespace: istio-system
  syncPolicy:
    syncOptions:
    - CreateNamespace=true
    automated:
      selfHeal: true
```

As we can see, it will deploy `v1.19.1` of the `istio-base` helm chart from `https://istio-release.storage.googleapis.com/charts` to the `istio-system` namespace of the Kubernetes cluster. Similarly, we will deploy `istiod` to the `istio-system` namespace using the following config:

```
...
  source:
    chart: istiod
    repoURL: https://istio-release.storage.googleapis.com/charts
    targetRevision: 1.19.1
    helm:
      releaseName: istiod
  destination:
    server: "https://kubernetes.default.svc"
    namespace: istio-system
...
```

Finally, we will install the `istio-ingress` component on the `istio-ingress` namespace using the following config:

```
...
  source:
    chart: gateway
    repoURL: https://istio-release.storage.googleapis.com/charts
    targetRevision: 1.19.1
    helm:
```

```
      releaseName: istio-ingress
  destination:
    server: "https://kubernetes.default.svc"
    namespace: istio-ingress
...
```

We will also define the configuration on Terraform so that we can use push-based GitOps to create our application automatically. So, we will append the following to the `app.tf` file:

```
data "kubectl_file_documents" "istio" {
    content = file("../manifests/argocd/istio.yaml")
}
resource "kubectl_manifest" "istio" {
  depends_on = [
    kubectl_manifest.gcpsm-secrets,
  ]
  for_each  = data.kubectl_file_documents.istio.manifests
  yaml_body = each.value
  override_namespace = "argocd"
}
```

Now, we can commit and push these files to our remote repository and wait for Argo CD to reconcile the changes using the following commands:

```
$ cd ~
$ cp -a ~/modern-devops/ch15/install-istio/app.tf \
 ~/mdo-environments/terraform/app.tf
$ cp -a ~/modern-devops/ch15/install-istio/istio.yaml \
 ~/mdo-environments/manifests/argocd/istio.yaml
$ git add --all
$ git commit -m "Install istio"
$ git push
```

As soon as we push the code, we'll see that the GitHub Actions workflow has been triggered. To access the workflow, go to `https://github.com/<your_github_user>/mdo-environments/actions`. Soon, the workflow will apply the configuration and create the Kubernetes cluster, deploy Argo CD, external secrets, our Blog App, and Istio.

Once the workflow succeeds, we must access the Argo Web UI. To do that, we need to authenticate with the GKE cluster. To do so, run the following command:

```
$ gcloud container clusters get-credentials \
 mdo-cluster-dev --zone us-central1-a --project $PROJECT_ID
```

To utilize the Argo CD Web UI, you will require the external IP address of the `argo-server` service. To get that, run the following command:

```
$ kubectl get svc argocd-server -n argocd
NAME          TYPE        EXTERNAL-IP  PORTS          AGE
argocd-server LoadBalaner 34.122.51.25 80/TCP,443/TCP 6m15s
```

Now, we know that Argo CD can be accessed at `https://34.122.51.25/`.

Next, we will run the following commands to reset the admin password:

```
$ kubectl patch secret argocd-secret -n argocd \
-p '{"data": {"admin.password": null, "admin.passwordMtime": null}}'
$ kubectl scale deployment argocd-server --replicas 0 -n argocd
$ kubectl scale deployment argocd-server --replicas 1 -n argocd
```

Now, allow 2 minutes for the new credentials to be generated. After that, execute the following command to retrieve the password:

```
$ kubectl -n argocd get secret argocd-initial-admin-secret \
 -o jsonpath="{.data.password}" | base64 -d && echo
```

Now that we have the credentials, we can log in. We will see the following page:

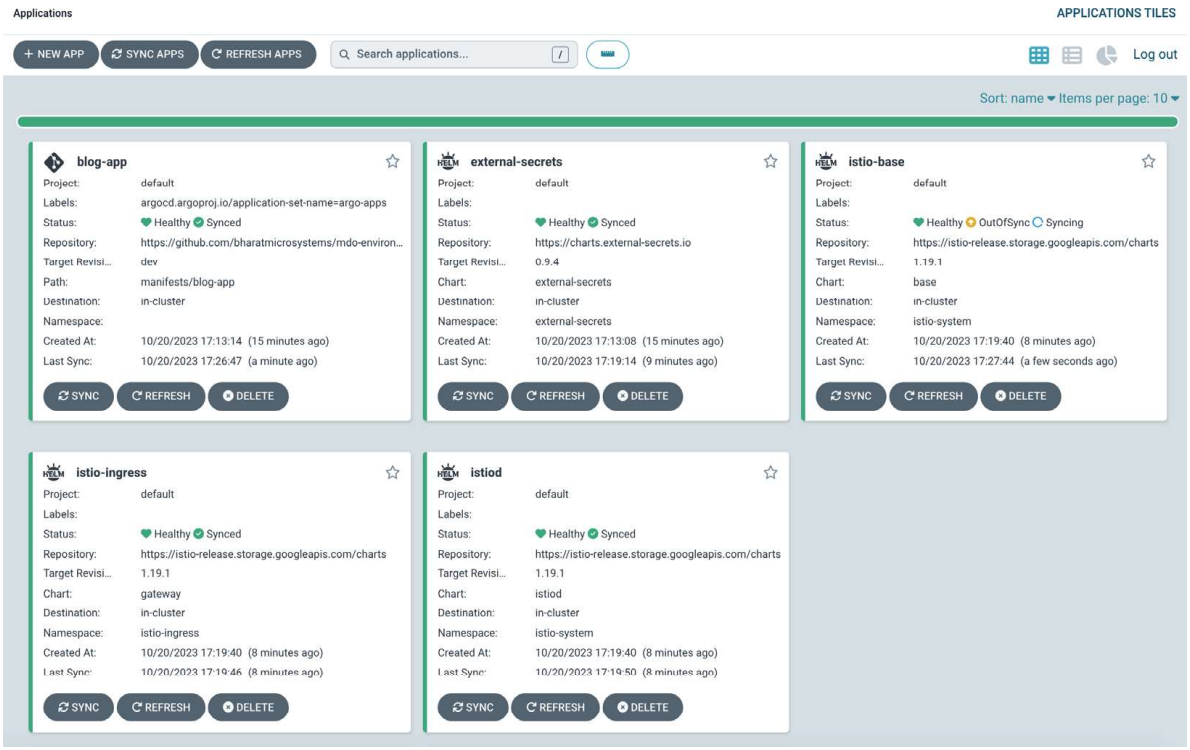

Figure 15.4 – Argo CD Web UI – home page

As we can see, the Istio applications are up and running. Though Istio is installed and running, the sidecars won't be injected unless we ask Istio to do so. We'll look at this next.

Enabling automatic sidecar injection

Since envoy sidecars are the key technology behind Istio's capabilities, they must be added to your existing pods to enable Istio to manage them. Updating each pod's configuration to include these sidecars can be challenging. To address this challenge, Istio offers a solution by enabling the automatic injection of these sidecars. To allow automatic sidecar injection on a namespace, we must add a label – that is, `istio-injection: enabled`. To do so, we will modify the `blog-app.yaml` file and add the label to the namespace resource:

```
apiVersion: v1
 kind: Namespace
 metadata:
   name: blog-app
   labels:
     istio-injection: enabled
...
```

Now, we can commit this resource to Git and push the changes remotely using the following commands:

```
$ cd ~
$ cp -a ~/modern-devops/ch15/install-istio/blog-app.yaml \
 ~/mdo-environments/manifests/blog-app/blog-app.yaml
$ git add --all
$ git commit -m "Enable sidecar injection"
$ git push
```

In the next Argo CD sync, we will soon find the label attached to the namespace. As soon as the label is applied, we need to restart our deployments and stateful sets, at which point new pods will come up with the injected sidecars. Use the following commands to do so:

```
$ kubectl -n blog-app rollout restart deploy frontend
$ kubectl -n blog-app rollout restart deploy posts
$ kubectl -n blog-app rollout restart deploy users
$ kubectl -n blog-app rollout restart deploy reviews
$ kubectl -n blog-app rollout restart deploy ratings
$ kubectl -n blog-app rollout restart statefulset mongodb
```

Now, let's list the pods in the `blog-app` namespace using the following command:

```
$ kubectl get pod -n blog-app
NAME                       READY   STATUS    RESTARTS   AGE
frontend-759f58f579-gqkp9  2/2     Running   0          109s
mongodb-0                  2/2     Running   0          98s
posts-5cdcb5cdf6-6wjrr     2/2     Running   0          108s
ratings-9888d6fb5-j2712    2/2     Running   0          105s
reviews-55ccb7fbd9-vw72m   2/2     Running   0          106s
users-5dbd56c4c5-stgjp     2/2     Running   0          107s
```

As we can see, the pods now show two containers instead of one. The extra container is the envoy sidecar. Istio's installation and setup are complete.

Now that our application has the Istio sidecar injected, we can use Istio ingress to allow traffic to our application, which is currently exposed via a load balancer service.

Using Istio ingress to allow traffic

We need to create a Blog App ingress gateway to associate our application with the Istio ingress gateway. It is necessary for configuring our application to route traffic through the Istio ingress gateway as we want to leverage Istio's traffic management and security features.

Istio deploys the Istio ingress gateway as a part of the installation process, and it's exposed on a load balancer by default. To determine the load balancer's IP address and ports, you can run the following commands:

```
$ kubectl get svc istio-ingress -n istio-ingress
NAME           EXTERNAL-IP    PORT(S)
istio-ingress  34.30.247.164  80:30950/TCP,443:32100/TCP
```

As we can see, Istio exposes various ports on your load balancer, and as our application needs to run on port 80, we can access it using `http://<IngressLoadBalancerExternalIP>:80`.

The next step would be to use this ingress gateway and expose our application. For that, we need to create **Gateway** and **VirtualService** resources.

Istio gateway is a **custom resource definition** (**CRD**) that helps you define how incoming external traffic can access services in your mesh. It acts as an entry point to your service and a load balancer for incoming traffic. When external traffic arrives at a gateway, it determines how to route it to the appropriate services based on the specified routing rules.

When we define an Istio gateway, we also need to define a `VirtualService` resource that uses the gateway and describes the routing rules for the traffic. Without a `VirtualService` resource, the Istio gateway will not know where and how to route the traffic it receives. A `VirtualService` resource is not only used for routing traffic from gateways but also for routing traffic within different services of the mesh. It allows you to define sophisticated routing rules, including traffic splitting, retries, timeouts, and more. Virtual services are often associated with specific services or workloads and determine how traffic should be routed to them. You can use virtual services to control how traffic is distributed among different versions of a service, enabling practices such as A/B testing, canary deployments, and Blue/Green deployments. Virtual services can also route traffic based on HTTP headers, paths, or other request attributes. In the current context, we will use the `VirtualService` resource to filter traffic based on paths and route them all to the **frontend** microservice.

Let's look at the definition of the `Gateway` resource first:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: blog-app-gateway
```

```
  namespace: blog-app
spec:
  selector:
    istio: ingress
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

As we can see, we define a `Gateway` resource that uses the Istio ingress gateway (defined by the `istio: ingress` selector) and listens on HTTP port `80`. It allows connection to all hosts as we've set that to `"*"`. For gateways to work correctly, we need to define a `VirtualService` resource. Let's look at that next:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: blog-app
  namespace: blog-app
spec:
  hosts:
  - "*"
  gateways:
  - blog-app-gateway
  http:
  - match:
    - uri:
        exact: /
    - uri:
        prefix: /static
    - uri:
        prefix: /posts
    - uri:
        exact: /login
    - uri:
        exact: /logout
    - uri:
        exact: /register
    - uri:
        exact: /updateprofile
    route:
    - destination:
        host: frontend
        port:
          number: 80
```

The `VirtualService` resource listens on all hosts and applies to `blog-app-gateway` as specified. It allows `/static`, and `/posts` as a `prefix` match. This means all requests with a URI that begins with them would be routed. The `/login`, `/logout`, `/register`, `/updateprofile`, and `/` paths have an `exact` match, which means that the exact URI is matched and allowed. These are routed to the `frontend` service on port `80`.

We must also modify the `frontend` service within the `blog-app.yaml` file to change the service type to `ClusterIP`. This will remove the attached load balancer from the service, and all requests will be routed via the ingress gateway.

Now, let's go ahead and apply these changes using the following commands:

```
$ cd ~/mdo-environments
$ cp ~/modern-devops/ch15/istio-ingressgateway/gateway.yaml \
manifests/blog-app/gateway.yaml
$ cp ~/modern-devops/ch15/istio-ingressgateway/blog-app.yaml \
manifests/blog-app/blog-app.yaml
$ git add --all
$ git commit -m "Added gateway"
$ git push
```

We will wait 5 minutes for the sync to work, after which we can go to `http://<Ingress LoadBalancerExternalIP>` to access our Blog App. You should see the following page. This shows that the application is working correctly:

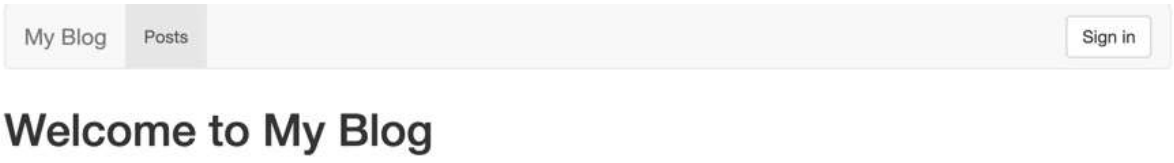

Figure 15.5 – Blog App – home page

You can play around with the application by registering, logging in, creating a post, and writing a review. Try updating the post and reviews to see whether all aspects of the application are working. Now, let's look at the security aspects of our microservices.

Securing your microservices using Istio

Running microservices in production offers numerous advantages, such as independent scalability, enhanced agility, reduced scope of change, frequent deployments, and reusability. However, they also introduce unique challenges, particularly in terms of security.

In a monolithic architecture, the security focus revolves around safeguarding a single application. However, in a typical enterprise-grade microservices application, hundreds of microservices may

need to interact securely with each other. Kubernetes serves as an excellent platform for hosting and orchestrating microservices. Nevertheless, the default communication between microservices is insecure, as they typically use plaintext HTTP. This may not meet your security requirements. To apply the same security principles to microservices as you would to a traditional enterprise monolith, you must ensure the following:

- **Encrypted communications**: All interactions between microservices must be encrypted to prevent potential man-in-the-middle attacks
- **Access control**: Access control mechanisms need to be in place to ensure that only authorized microservices can interface with each other
- **Telemetry and audit logging**: Capturing, logging, and auditing telemetry data is crucial to understanding traffic behavior and proactively detecting intrusions

Istio simplifies addressing these security concerns and provides these essential security features out of the box. With Istio, you can enforce strong **identity and access management**, mutual **TLS** and **encryption**, **authentication** and **authorization**, and comprehensive **audit logging** – all within a unified control plane. This means you can establish robust security practices for your microservices, promoting the safety and reliability of your applications in a dynamic and distributed environment.

In the context of Istio, you should be aware that it automatically injects sidecar proxies into your pods and modifies the IP tables of your Kubernetes cluster to ensure that all connections occur through these proxies. This setup is designed to enforce TLS encryption by default, enhancing the security of your microservices without requiring specific configurations. The communication between these envoy proxies within the service mesh is automatically secured through TLS.

While the default setup offers a foundational level of security and effectively prevents man-in-the-middle attacks, it's advisable to further bolster the security of your microservices by applying specific policies. Before delving into the detailed features, having a high-level understanding of how security functions in Istio is beneficial.

Istio incorporates the following key components for enforcing security:

- **Certificate authority** (**CA**): This component manages keys and certificates, ensuring secure and authenticated communication within the service mesh.
- **Configuration API Server**: The Configuration API Server distributes authentication policies, authorization policies, and secure naming information to the envoy proxies. These policies define how services can authenticate and authorize each other and manage secure communication.
- **Sidecar proxies**: Sidecar proxies, deployed alongside your microservices, are crucial in enforcing security policies. They act as policy enforcement points, implementing the policies supplied to them.
- **Envoy proxy extensions**: These extensions enable the collection of telemetry data and auditing, providing insights into traffic behavior and helping to identify and mitigate security issues.

With these components working in concert, Istio ensures a robust security framework for your microservices, which can be further fine-tuned by defining and enforcing specific security policies tailored to your application's needs.

As our application currently runs on HTTP, it would be a great idea to implement TLS in our Blog App and expose it over HTTPS. Let's start by creating a secure ingress gateway for this.

Creating secure ingress gateways

Secure ingress gateways are nothing but **TLS-enabled ingress gateways**. To enable TLS on an ingress gateway, we must provide it with a **private key** and a **certificate** chain. We will use a self-signed certificate chain for this exercise, but you must use a proper CA certificate chain in production. A CA certificate is a digital certificate that's granted by a reputable CA, such as Verisign or Entrust, within a **public key infrastructure** (**PKI**). It plays a pivotal role in guaranteeing the security and reliability of digital interactions and transactions.

Let's start by creating a **root certificate** and **private key** to sign certificates for our application by using the following command:

```
$ openssl req -x509 -sha256 -nodes -days 365 \
-newkey rsa:2048 -subj '/O=example Inc./CN=example.com' \
-keyout example.com.key -out example.com.crt
```

Using the generated root certificate, we can now generate the **server certificate** and the key using the following commands:

```
$ openssl req -out blogapp.example.com.csr \
-newkey rsa:2048 -nodes -keyout blogapp.example.com.key \
-subj "/CN=blogapp.example.com/O=blogapp organization"
$ openssl x509 -req -sha256 -days 365 \
-CA example.com.crt -CAkey example.com.key -set_serial 1 \
-in blogapp.example.com.csr -out blogapp.example.com.crt
```

The next step is to generate a Kubernetes TLS secret within the `istio-ingress` namespace for our ingress gateway to read it. However, as we don't want to store the TLS key and certificate in our Git repository, we will use **Google Secrets Manager** instead. Therefore, let's run the following command to do so:

```
$ echo -ne "{\"MONGO_INITDB_ROOT_USERNAME\": \"root\", \
\"MONGO_INITDB_ROOT_PASSWORD\": \"itsasecret\", \
\"blogapptlskey\": \"$(base64 blogapp.example.com.key -w 0)\", \
\"blogapptlscert\": \"$(base64 blogapp.example.com.crt -w 0)\"}" | \
gcloud secrets versions add external-secrets --data-file=-
Created version [2] of the secret [external-secrets].
```

Now, we must create an external secret manifest to fetch the keys and certificates from Secrets Manager and generate a TLS secret. The following manifest will help us achieve that:

```
apiVersion: external-secrets.io/v1alpha1
kind: ExternalSecret
metadata:
  name: blogapp-tls-credentials
  namespace: istio-ingress
spec:
  secretStoreRef:
    kind: ClusterSecretStore
    name: gcp-backend
  target:
    template:
      type: kubernetes.io/tls
      data:
        tls.crt: "{{ .blogapptlscert | base64decode | toString }}"
        tls.key: "{{ .blogapptlskey | base64decode | toString }}"
    name: blogapp-tls-credentials
  data:
  - secretKey: blogapptlskey
    remoteRef:
      key: external-secrets
      property:  blogapptlskey
  - secretKey: blogapptlscert
    remoteRef:
      key: external-secrets
      property: blogapptlscert
```

Now, let's create a directory within our Environment Repository and copy the external secret manifest there. Use the following commands for that:

```
$ mkdir ~/mdo-environments/manifests/istio-ingress
$ cp ~/modern-devops/ch15/security/blogapp-tls-credentials.yaml \
~/mdo-environments/manifests/istio-ingress
```

Next, we need to modify the ingress gateway resource to configure TLS. To do so, we must modify the `Gateway` resource to the following:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: blog-app-gateway
  namespace: blog-app
spec:
  selector:
    istio: ingress
  servers:
  - port:
      number: 443
```

```
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE
      credentialName: blogapp-tls-credentials
    hosts:
    - "*"
```

The gateway configuration is similar to the previous one, but instead of `port 80`, we're using `port 443` for `HTTPS`. We also have a `tls` section with a `SIMPLE` mode, which means it is a standard TLS connection. We've specified `credentialName`, pointing to the secret we created using the TLS key and certificate. Since all the setup is now ready, let's commit and push the code using the following commands:

```
$ cp ~/modern-devops/ch15/security/gateway.yaml \
~/mdo-environments/manifests/blog-app/
$ cp ~/modern-devops/ch15/security/run-tests.yml \
~/mdo-environments/.github/workflows/
$ git add --all
$ git commit -m "Enabled frontend TLS"
$ git push
```

Wait for `blog-app` to sync. Once we've done this, we can access our application at `https://<IngressLoadBalancerExternalIP>`. With that, the connection coming into our application has been encrypted.

Though we've secured connection coming into our mesh, securing all internal service interactions with services using TLS within your service mesh would be good as an additional security layer. We'll implement that next.

Enforcing TLS within your service mesh

As we know by now, by default, Istio provides TLS encryption for communication between workloads that have sidecar proxies injected. However, it's important to note that this default setting operates in compatibility mode. In this mode, traffic between two services with sidecar proxies injected is encrypted. However, workloads without sidecar proxies can still communicate with backend microservices over plaintext HTTP. This design choice is made to simplify the adoption of Istio, as teams newly introducing Istio don't need to immediately address the issue of making all source traffic TLS-enabled.

Let's create and get a shell to a pod in the `default` namespace. The backend traffic will be plaintext because the namespace does not have automatic sidecar injection. We will then `curl` the `frontend` microservice from there and see whether we get a response. Run the following command to do so:

```
$ kubectl run -it --rm --image=curlimages/curl curly -- curl -v http://frontend.blog-app
*   Trying 10.71.246.145:80…
* Connected to frontend (10.71.246.145) port 80
> GET / HTTP/1.1
```

```
> Host: frontend
> User-Agent: curl/8.4.0
> Accept: */*
< HTTP/1.1 200 OK
< server: envoy
< date: Sat, 21 Oct 2023 07:19:18 GMT
< content-type: text/html; charset=utf-8
< content-length: 5950
< x-envoy-upstream-service-time: 32
<!doctype html>
<html la"g="en">
...
```

As we can see, we get an `HTTP 200` response back.

This approach balances security and compatibility, allowing a gradual transition to a fully encrypted communication model. Over time, as more services have sidecar proxies injected, the overall security posture of the microservices application improves. However, as we are starting fresh, enforcing strict TLS for our Blog App would make sense. So, let's do that.

To enable strict TLS on a workload, namespace, or the entire cluster, Istio provides peer authentication policies using the `PeerAuthentication` resource. As we only need to implement strict TLS on the Blog App, enabling it at the namespace level would make sense. To do that, we will use the following `PeerAuthentication` resource:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
  namespace: blog-app
spec:
  mtls:
    mode: STRICT
```

Now, let's apply this using the following commands:

```
$ cp ~/modern-devops/ch15/security/strict-mtls.yaml \
 ~/mdo-environments/manifests/blog-app/
$ git add --all
$ git commit -m "Enable strict TLS"
$ git push
```

Argo CD should pick up the new configuration and apply the strict TLS policy as soon as we push the changes. Wait for the Argo CD sync to be in a clean state, and run the following commands to check whether strict TLS is working:

```
$ kubectl run -it --rm --image=curlimages/curl  curly -- curl -v http://frontend.blog-app
*   Trying 10.71.246.145:80...
* Connected to frontend.blog-app (10.71.246.145) port 80
> GET / HTTP/1.1
> Host: frontend.blog-app
```

```
> User-Agent: curl/8.4.0
> Accept: */*
* Recv failure: Connection reset by peer
* Closing connection
curl: (56) Recv failure: Connection reset by peer
```

As we can see, the request has now been rejected as it is a plaintext request, and the backend will only allow TLS. This shows that strict TLS is working fine. Now, let's move on and secure our services even better.

From our design, we know how services interact with each other:

- The `frontend` microservice can only connect to the `posts`, `reviews`, and `users` microservices
- Only the `reviews` microservice can connect to the `ratings` microservice.
- Only the `posts`, `reviews`, `users`, and `ratings` microservices can connect to the `mongodb` database

Therefore, we can define these interactions and only allow these connections explicitly. Therefore, the `frontend` microservice will not be able to connect with the `mongodb` database directly, even if it tries to.

Istio provides the `AuthorizationPolicy` resource to manage this. Let's implement the preceding scenario using that.

Let's start with the `posts` microservice:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: posts
  namespace: blog-app
spec:
  selector:
    matchLabels:
      app: posts
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/blog-app/sa/frontend"]
```

The `AuthorizationPolicy` has multiple sections. It starts with `name` and `namespace`, which are `posts` and `blog-app`, respectively. The `spec` section contains `selector`, where we specify that we need to apply this policy to all pods with the `app: posts` label. We use an `ALLOW` action for this. Note that Istio has an implicit `deny-all` policy for all pods that match the selector, and any `ALLOW` rules will be applied on top of that. Any traffic that does not match the `ALLOW` rules will be denied by default. We have rules to define what traffic to allow; here, we're using the `from` >

`source` > `principals` and setting the `frontend` service account on this. So, in summary, this rule will apply to the `posts` microservice and only allow traffic from the `frontend` microservice.

Similarly, we will apply the same policy to the `reviews` microservice, as follows:

```
...
  name: reviews
...
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/blog-app/sa/frontend"]
```

The `users` microservice also only needs to accept traffic from the `frontend` microservice:

```
...
  name: users
...
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/blog-app/sa/frontend"]
```

The `ratings` microservice should accept traffic only from the `reviews` microservice, so we will make a slight change to the principals, as follows:

```
...
  name: ratings
...
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/blog-app/sa/reviews"]
```

Finally, the `mongodb` service needs a connection from all microservices apart from `frontend`, so we must specify multiple entries in the principal section:

```
...
  name: mongodb
...
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/blog-app/sa/posts", "cluster.local/ns/blog-app/sa/
reviews", "cluster.local/ns/blog-app/sa/ratings", "cluster.local/ns/blog-app/sa/users"]
```

Since we've used service accounts to understand where the requests are coming from, we must also create and assign service accounts to respective services. So, we will modify the `blog-app.yaml` file and add service accounts for each service, something like the following:

```
apiVersion: v1
kind: ServiceAccount
```

```
metadata:
  name: mongodb
  namespace: blog-app
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
...
spec:
...
  template:
...
    spec:
      serviceAccountName: mongodb
      containers:
...
```

I've already replicated the same in the new `blog-app.yaml` file. Let's commit the changes and push them to GitHub so that we can apply them to our cluster:

```
$ cp ~/modern-devops/ch15/security/authorization-policies.yaml \
 ~/mdo-environments/manifests/blog-app/
$ cp ~/modern-devops/ch15/security/blog-app.yaml \
 ~/mdo-environments/manifests/blog-app/
$ git add --all
$ git commit -m "Added auth policies"
$ git push
```

Now, we must wait for the sync to complete and then verify the setup. First, we'll get a shell to the `frontend` pod and try to use `wget` to connect with the backend microservices. We will try to connect with each microservice and see what we get. If we get `HTTP 200` or `404`, this means the backend is allowing connections, while if we get `HTTP 403` or `Error`, this signifies the backend is blocking connections. Run the following commands to do so:

```
$ kubectl -n blog-app exec -it $(kubectl get pod -n blog-app | \
grep frontend | awk {'print $1'}) -- /bin/sh
/ # wget posts:5000
Connecting to posts:5000 (10.71.255.204:5000)
wget: server returned error: HTTP/1.1 404 Not Found
/ # wget reviews:5000
Connecting to reviews:5000 (10.71.244.177:5000)
wget: server returned error: HTTP/1.1 404 Not Found
/ # wget ratings:5000
Connecting to ratings:5000 (10.71.242.178:5000)
wget: server returned error: HTTP/1.1 403 Forbidden
/ # wget users:5000
Connecting to users:5000 (10.71.241.255:5000)
wget: server returned error: HTTP/1.1 404 Not Found
/ # wget mongodb:27017
Connecting to mongodb:27017 (10.68.0.18:27017)
```

```
wget: error getting response: Resource temporarily unavailable
/ # exit
command terminated with exit code 1
```

As we can see, we get an `HTTP 404` response from the `posts`, `reviews`, and `users` microservices. The `ratings` microservice returns a `403 Forbidden` response, and the `mongodb` service reports that the resource is unavailable. This means that our setup is working correctly.

Let's try the same with the `posts` microservice:

```
$ kubectl -n blog-app exec -it $(kubectl get pod -n blog-app | \
grep posts | awk {'print $1'}) -- /bin/sh
/ # wget mongodb:27017
Connecting to mongodb:27017 (10.68.0.18:27017)
saving to 'index.html'
index.html           100% |************|    85  0:00:00 ETA
'index.html' saved
/ # wget ratings:5000
Connecting to ratings:5000 (10.71.242.178:5000)
wget: server returned error: HTTP/1.1 403 Forbidden
/ # wget reviews:5000
Connecting to reviews:5000 (10.71.244.177:5000)
wget: server returned error: HTTP/1.1 403 Forbidden
/ # wget users:5000
Connecting to users:5000 (10.71.241.255:5000)
wget: server returned error: HTTP/1.1 403 Forbidden
/ # exit
command terminated with exit code 1
```

As we can see, the `posts` microservice can communicate successfully with `mongodb`, but the rest of the microservices return `403 Forbidden`. This is what we were expecting. Now, let's do the same with the `reviews` microservice:

```
$ kubectl -n blog-app exec -it $(kubectl get pod -n blog-app | \
grep reviews | awk {'print $1'}) -- /bin/sh
/ # wget ratings:5000
Connecting to ratings:5000 (10.71.242.178:5000)
wget: server returned error: HTTP/1.1 404 Not Found
/ # wget mongodb:27017
Connecting to mongodb:27017 (10.68.0.18:27017)
saving to 'index.html'
index.html           100% |**********|    85  0:00:00 ETA
'index.html' saved
/ # wget users:5000
Connecting to users:5000 (10.71.241.255:5000)
wget: server returned error: HTTP/1.1 403 Forbidden
/ # exit
command terminated with exit code 1
```

As we can see, the `reviews` microservice can successfully connect with the `ratings` microservice and `mongodb`, while getting a `403` response from other microservices. This is what we expected. Now, let's check the `ratings` microservice:

```
$ kubectl -n blog-app exec -it $(kubectl get pod -n blog-app \
   | grep ratings | awk {'print $1'}) -- /bin/sh
/ # wget mongodb:27017
Connecting to mongodb:27017 (10.68.0.18:27017)
saving to 'index.html'
index.html           100% |************|    85  0:00:00 ETA
'index.html' saved
/ # wget ratings:5000
Connecting to ratings:5000 (10.71.242.178:5000)
wget: server returned error: HTTP/1.1 403 Forbidden
/ # exit
command terminated with exit code 1
```

As we can see, the `ratings` microservice can only connect successfully with the `mongdb` database and gets a `403` response for other services.

Now that we've tested all the services, the setup is working fine. We've secured our microservices to a great extent! Now, let's look at another aspect of managing microservices with Istio – traffic management.

Managing traffic with Istio

Istio offers robust traffic management capabilities that form a core part of its functionality. When leveraging Istio for microservice management within your Kubernetes environment, you gain precise control over how these services communicate with each other. This empowers you to define the traffic path within your service mesh meticulously.

Some of the traffic management features at your disposal are as follows:

- Request routing
- Fault injection
- Traffic shifting
- TCP traffic shifting
- Request timeouts
- Circuit breaking
- Mirroring

The previous section employed an ingress gateway to enable traffic entry into our mesh and used a virtual service to distribute traffic to the services. With virtual services, the traffic distribution happens in a round-robin fashion by default. However, we can change that using destination rules. These rules provide us with an intricate level of control over the behavior of our mesh, allowing for a more granular management of traffic within the Istio ecosystem.

Before we delve into that, we need to update our Blog App so that it includes a new version of the `ratings` service deployed as `ratings-v2`, which will return black stars instead of orange stars. I've already updated the manifest for that in the repository. Therefore, we just need to copy that to the `mdo-environments` repository, commit it, and push it remotely using the following commands:

```
$ cd ~/mdo-environments/manifests/blog-app/
$ cp ~/modern-devops/ch15/traffic-management/blog-app.yaml .
$ git add --all
$ git commit -m "Added ratings-v2 service"
$ git push
```

Wait for the application to sync. After this, we need to do a few things:

1. Go to the Blog App home page > **Sign In** > **Not a User? Create an account** and create a new account.
2. Click on the **Actions** tab > **Add a Post**, add a new post with a title and content of your choice, and click **Submit**.
3. Use the **Add a Review** text field to add a review, provide a rating, and click **Submit**.
4. Click on **Posts** again and access the post that we had created.

Now, keep refreshing the page. We will see that we get orange stars half the time and black stars for the rest. Traffic is splitting equally across `v1` and `v2` (that is, the orange and black stars):

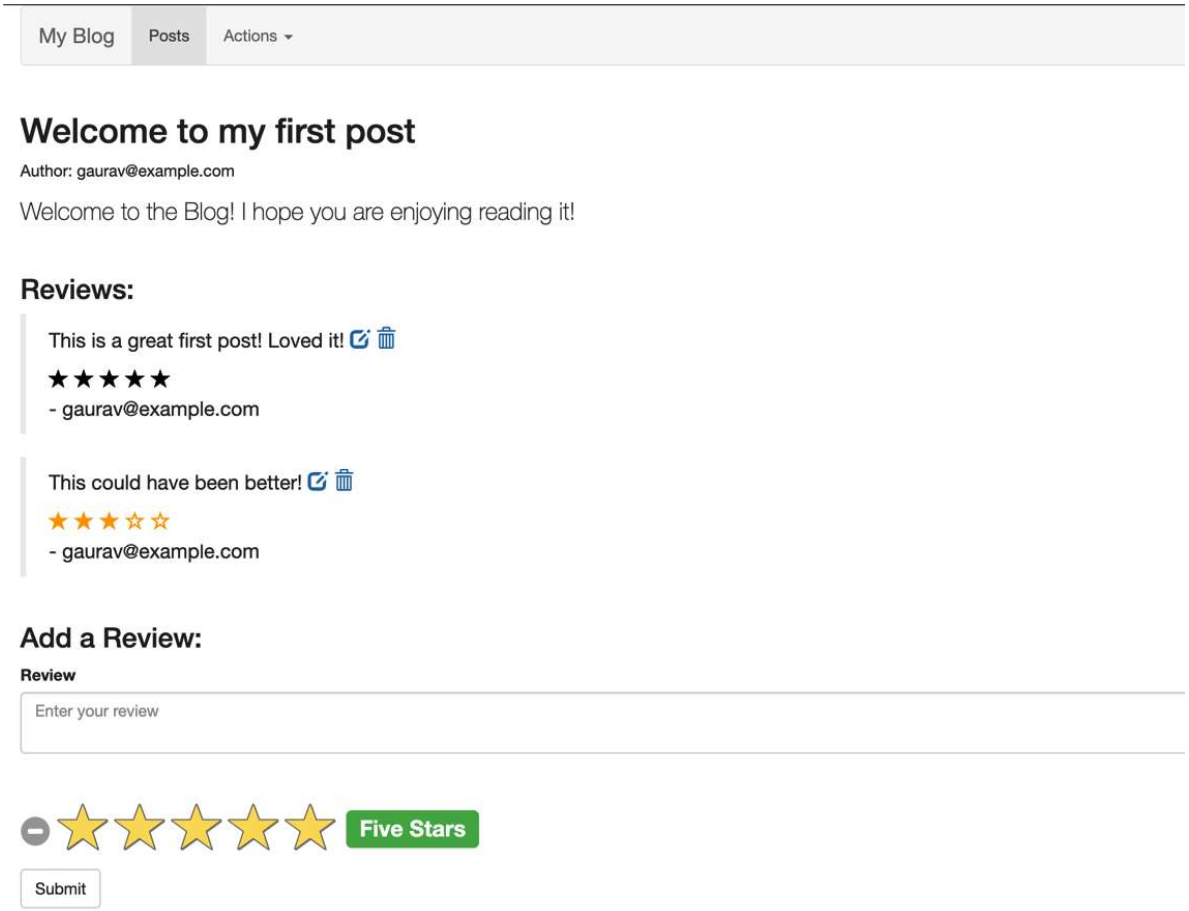

Figure 15.6 – Round robin routing

This occurs due to the absence of destination rules, which leaves Istio unaware of the distinctions between `v1` and `v2`. Let's define destination rules for our microservices to rectify this, clearly informing Istio of these versions. In our case, we have one version for each microservice, except for the `ratings` microservice, so we'll define the following destination rules accordingly.

Let's start by defining the destination rule of the `frontend` microservice:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: frontend
  namespace: blog-app
spec:
  host: frontend
  subsets:
  - name: v1
    labels:
      version: v1
```

The provided YAML manifest introduces a `DestinationRule` resource named `frontend` within the `blog-app` namespace. This resource is associated with the host named `frontend`. Subsequently, we define subsets labeled as `v1`, targeting pods with the `version: v1` label. Consequently, configuring our virtual service to direct traffic to the `v1` destination will route requests to pods bearing the `version: v1` label.

This same configuration approach can be replicated for the `posts`, `users`, and `reviews` microservices. However, the `ratings` microservice requires a slightly different configuration due to the deployment of two versions, as follows:

```
...
spec:
  host: ratings
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

The YAML manifest for the `ratings` microservice closely resembles that of the other microservices, with one notable distinction: it features a second subset labeled as `v2`, corresponding to pods bearing the `version: v2` label.

Consequently, requests routed to the `v1` destination target all pods with the `version: v1` label, while requests routed to the `v2` destination are directed to pods labeled `version: v2`.

To illustrate this in a practical context, we will proceed to define virtual services for each microservice. Our starting point will be defining the virtual service for the `frontend` microservice, as illustrated in the following manifest:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: frontend
  namespace: blog-app
spec:
  hosts:
  - frontend
  http:
  - route:
    - destination:
        host: frontend
        subset: v1
```

The provided YAML manifest outlines a `VirtualService` resource named `frontend` within the `blog-app` namespace. This resource configures the host `frontend` with an HTTP route destination, directing all traffic to the `frontend` host and specifying the `v1` subset. Consequently, all requests targeting the `frontend` host will be routed to the `v1` destination that we previously defined.

We will replicate this configuration approach for the `posts`, `reviews`, and `users` microservices, creating corresponding `VirtualService` resources. In the case of the `ratings` microservice, the decision is made to route all traffic to the `v1` (orange stars) version. Therefore, we apply a similar `VirtualService` resource for the `ratings` microservice as well.

Now, let's copy the manifests to the `mdo-environments` repository and commit and push the code to the remote repository using the following commands:

```
$ cd ~/mdo-environments/manifests/blog-app
$ cp ~/modern-devops/ch15/traffic-management/destination-rules.yaml .
$ cp ~/modern-devops/ch15/traffic-management/virtual-services-v1.yaml .
$ git add --all
$ git commit -m "Route to v1"
$ git push
```

Wait for Argo CD to sync the changes. Now, all requests will route to `v1`. Therefore, you will only see orange stars in the reviews, as shown in the following screenshot:

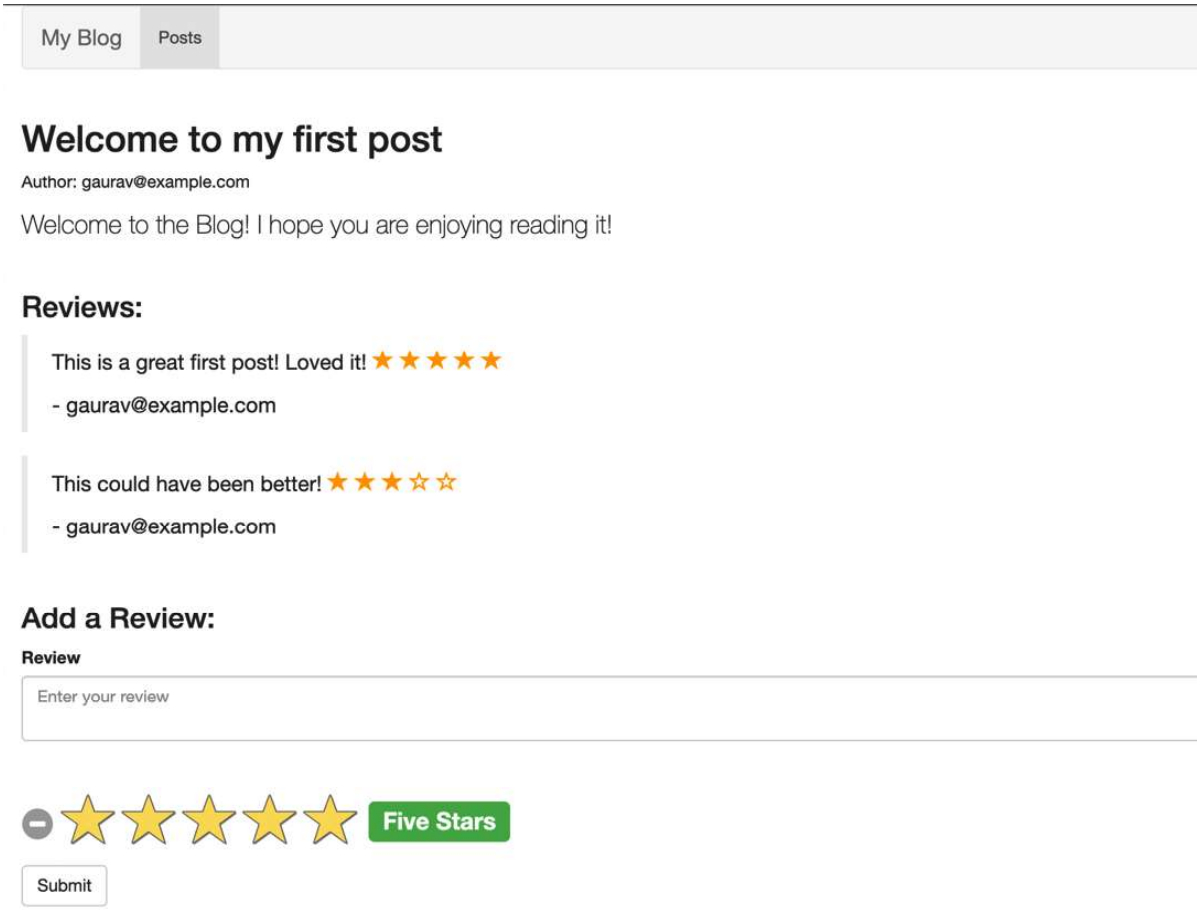

Figure 15.7 – Route to v1

Now, let's try to roll out `v2` using a canary rollout approach.

Traffic shifting and canary rollouts

Consider a scenario where you've developed a new version of your microservice and are eager to introduce it to your user base. However, you're understandably cautious about the potential impact on the entire service. In such cases, you may opt for a deployment strategy known as **canary rollouts**, also known as a **Blue/Green deployment**.

The essence of a canary rollout lies in its incremental approach. Instead of an abrupt transition, you methodically shift traffic from the previous version (referred to as **Blue**) to the new version (**Green**). This gradual migration allows you to thoroughly test the functionality and reliability of the new release with a limited subset of users before implementing it across the entire user base. This approach minimizes the risks associated with deploying new features or updates and ensures a more controlled and secure release process. The following figure illustrates this process beautifully:

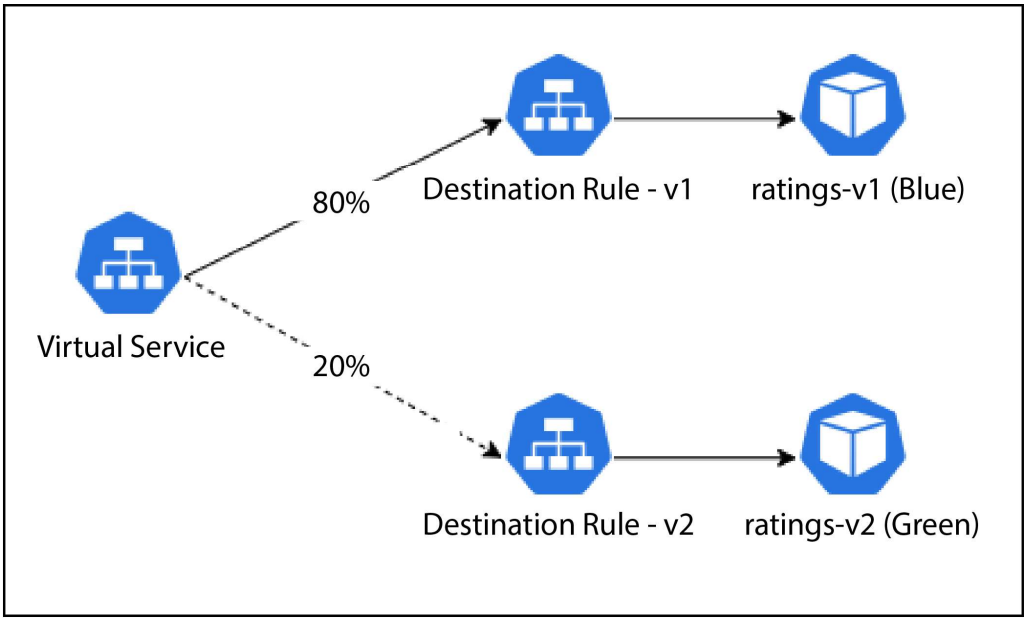

Figure 15.8 – Canary rollouts

Here's how a *canary rollout* strategy works:

1. **Initial release**: The existing version (referred to as the *baseline* or *current version*) continues to serve the majority of users.
2. **Early access**: A small group of users or systems, typically selected as a representative sample, is identified as the *canary group*. They receive the new version.
3. **Monitoring and evaluation**: The software's performance and behavior in the canary group are closely monitored. Metrics, logs, and user feedback are collected to identify issues or anomalies.

4. **Gradual expansion**: If the new version proves stable and performs as expected in the canary group, its exposure is incrementally expanded to a broader user base. This expansion can occur in stages, with a small percentage of users being "promoted" to the new version at each stage.
5. **Continuous monitoring**: Throughout the rollout, continuous monitoring and analysis are critical to identify and address any emerging issues promptly. If problems are detected, the rollout can be halted or reversed to protect the majority of users.
6. **Full deployment**: Once the new version has been successfully validated through the canary rollout stages, it is eventually made available to the entire user base.

So, let's roll out the `ratings-v2` service to `20%` of our users. For that, we'll use the following `VirtualService` resource:

```
...
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
      weight: 80
    - destination:
        host: ratings
        subset: v2
      weight: 20
```

As we can see, we've modified the `ratings` virtual service to introduce a second destination pointing to the `v2` subset. A noteworthy addition in this configuration is the introduction of the `weight` attribute. For the `v1` destination, we have assigned a weight of `80`, while the `v2` destination carries a weight of `20`. This means that `20%` of the traffic will be directed to the `v2` version of the `ratings` microservice, providing a controlled and adjustable distribution of traffic between the two versions.

Let's copy the manifest and then commit and push the changes to the remote repository using the following commands:

```
$ cd ~/mdo-environments/manifests/blog-app
$ cp ~/modern-devops/ch15/traffic-management/virtual-services-canary.yaml \
virtual-services.yaml
$ git add --all
$ git commit -m "Canary rollout"
$ git push
```

Following the completion of the Argo CD sync, if you refresh the page 10 times, you'll observe that black stars appear twice out of those 10 times. This is a prime example of a canary rollout in action. You can continue monitoring the application and gradually adjust the weights to shift traffic toward `v2`. Canary rollouts effectively mitigate risks during production rollouts, providing a method to address the fear of the unknown, especially when implementing significant changes.

However, another approach exists to test your code in a production environment that involves using live traffic without exposing your application to end users. This method is known as traffic mirroring. We'll delve into it in the following discussion.

Traffic mirroring

Traffic mirroring, also called shadowing, is a concept that has recently gained traction. It is a powerful approach that allows you to assess your releases in a production environment without posing any risk to your end users.

Traditionally, many enterprises maintained a staging environment that closely mimicked the production setup. The Ops team deployed new releases to the staging environment in this setup while testers generated synthetic traffic to simulate real-world usage. This approach provided a means for teams to evaluate how the code would perform in the production environment, assessing its functional and non-functional aspects before promoting it to production. The staging environment served as the ground for performance, volumetric, and operational acceptance testing. While this approach had its merits, it was not without its challenges. Maintaining static test environments, which involved substantial costs and resources, was one of them. Creating and sustaining a replica of the production environment required a team of engineers, leading to high overhead.

Moreover, synthetic traffic often deviated from real live traffic since the former relied on historical data, while the latter reflected current user interactions. This discrepancy occasionally led to overlooked scenarios.

On the other hand, traffic mirroring offers a solution that similarly enables operational acceptance testing while going a step further. It allows you to conduct this testing using live, real-time traffic without any impact on end users.

Here's how traffic mirroring operates:

1. Deploy a new version of the application and activate traffic mirroring.
2. The old version continues to respond to requests as usual but concurrently sends an asynchronous copy of the traffic to the new version.
3. The new version processes the mirrored traffic but refrains from responding to end users.
4. The ops team monitors the behavior of the new version and reports any issues to the development team.

This process is depicted in the following figure:

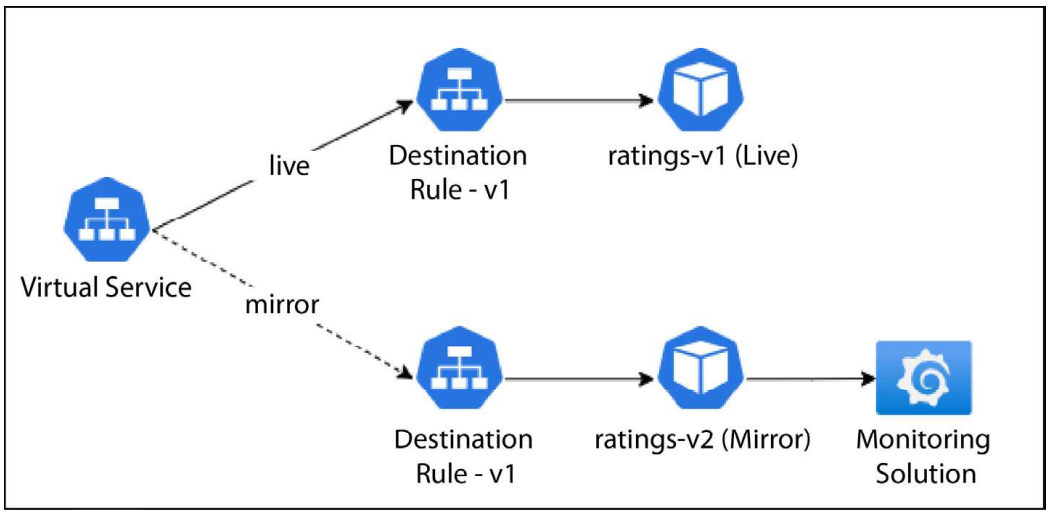

Figure 15.9 – Traffic mirroring

Traffic mirroring revolutionizes the testing process by enabling teams to uncover issues that might remain hidden in a traditional staging environment. Additionally, you can utilize monitoring tools such as Prometheus and Grafana to record and monitor the outcomes of your testing efforts, enhancing the overall quality and reliability of your releases.

Now, without further ado, let's configure traffic mirroring for our `ratings` service. Traffic mirroring is managed through the `VirtualService` resource, so let's modify the `ratings` virtual service to the following:

```
...
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
      weight: 100
    mirror:
      host: ratings
      subset: v2
    mirror_percent: 100
```

In this configuration, we set up a single `destination` targeting `v1` with a `weight` value of `100`. Additionally, we defined a `mirror` section that directs traffic to `ratings:v2` with a `mirror_percent` value of 100. This signifies that all traffic initially routed to `ratings:v1` is mirrored and simultaneously sent to `v2`.

Let's commit the changes and push them to the remote repository using the following commands:

```
$ cp ~/modern-devops/ch15/traffic-management/virtual-services-mirroring.yaml \
virtual-services.yaml
$ git add --all
$ git commit -m "Mirror traffic"
$ git push
```

Following the completion of the Argo CD synchronization process, we'll proceed to refresh the page five times. Subsequently, we can inspect the logs of the `ratings:v1` service using the following command:

```
$ kubectl logs $(kubectl get pod -n blog-app | \
grep "ratings-" | awk '{print $1}') -n blog-app
127.0.0.6 - - [22/Oct/2023 08:33:19] "GET /review/6534cba72485f5a51cbdcef0/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:19] "GET /review/6534cbb32485f5a51cbdcef1/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:23] "GET /review/6534cba72485f5a51cbdcef0/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:23] "GET /review/6534cbb32485f5a51cbdcef1/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:25] "GET /review/6534cba72485f5a51cbdcef0/rating
HTTP/1.1" 200 -
```

With traffic mirroring active, it's expected that the same set of logs observed in the `ratings:v1` service will also be mirrored in the `ratings:v2` service. To confirm this, we can list the logs for the `ratings:v2` service using the following command:

```
$ kubectl logs $(kubectl get pod -n blog-app | \
grep "ratings-v2" | awk '{print $1}') -n blog-app
127.0.0.6 - - [22/Oct/2023 08:33:19] "GET /review/6534cba72485f5a51cbdcef0/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:19] "GET /review/6534cbb32485f5a51cbdcef1/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:23] "GET /review/6534cba72485f5a51cbdcef0/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:23] "GET /review/6534cbb32485f5a51cbdcef1/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:25] "GET /review/6534cba72485f5a51cbdcef0/rating
HTTP/1.1" 200 -
```

Indeed, the logs and timestamps match precisely, providing clear evidence of concurrent log entries in `ratings:v1` and `ratings:v2`. This observation effectively demonstrates the mirroring functionality in operation, showcasing how traffic is duplicated for real-time monitoring and analysis in both versions.

Traffic mirroring is a highly effective method for identifying issues that often elude detection within traditional infrastructure setups. It is a potent approach for conducting operational acceptance testing of your software releases. This practice simplifies testing and safeguards against potential customer incidents and operational challenges.

There are other aspects of traffic management that Istio provides, but covering all of them is beyond the scope of this chapter. Please feel free to explore other aspects of it by visiting the Istio documentation: `https://istio.io/latest/docs/tasks/traffic-management/`.

As we already know, Istio leverages envoy proxies as sidecar components alongside your microservice containers. Given that these proxies play a central role in directing and managing the traffic within your service mesh, they also collect valuable telemetry data.

This telemetry data is subsequently transmitted to Prometheus, a monitoring and alerting tool, where it can be stored and effectively visualized. Tools such as Grafana are often employed in conjunction with Prometheus to provide insightful and accessible visualizations of this telemetry data, empowering you to monitor and manage your service mesh effectively. Therefore, we'll go ahead and explore the observability portion of Istio in the next section.

Observing traffic and alerting with Istio

Istio provides several tools to visualize traffic through our mesh through Istio add-ons. While **Prometheus** is the central telemetry data collection, storage, and query layer, **Grafana** and **Kiali** provide us with interactive graphical tools to interact with that data.

Let's start this section by installing the observability add-ons using the following commands:

```
$ cd ~
$ mkdir ~/mdo-environments/manifests/istio-system
$ cd ~/mdo-environments/manifests/istio-system/
$ cp ~/modern-devops/ch15/observability/*.yaml .
$ git add --all
$ git commit -m "Added observability"
$ git push
```

As soon as we push the code, Argo CD should create a new `istio-system` namespace and install the add-ons. Once they have been installed, we can start by accessing the Kiali dashboard.

Accessing the Kiali dashboard

Kiali is a powerful observability and visualization tool for microservices and service mesh management. It offers real-time insights into the behavior of your service mesh, helping you monitor and troubleshoot issues efficiently.

As the Kiali service is deployed on a cluster IP and hence not exposed externally, let's do a port forward to access the Kiali dashboard using the following command:

```
$ kubectl port-forward deploy/kiali -n istio-system 20001:20001
```

Once the port forward session has started, click on the web preview icon of Google Cloud Shell, choose **Change port to 20001**, and click **preview**. You will see the following dashboard. This dashboard provides valuable insights into the applications running across the mesh:

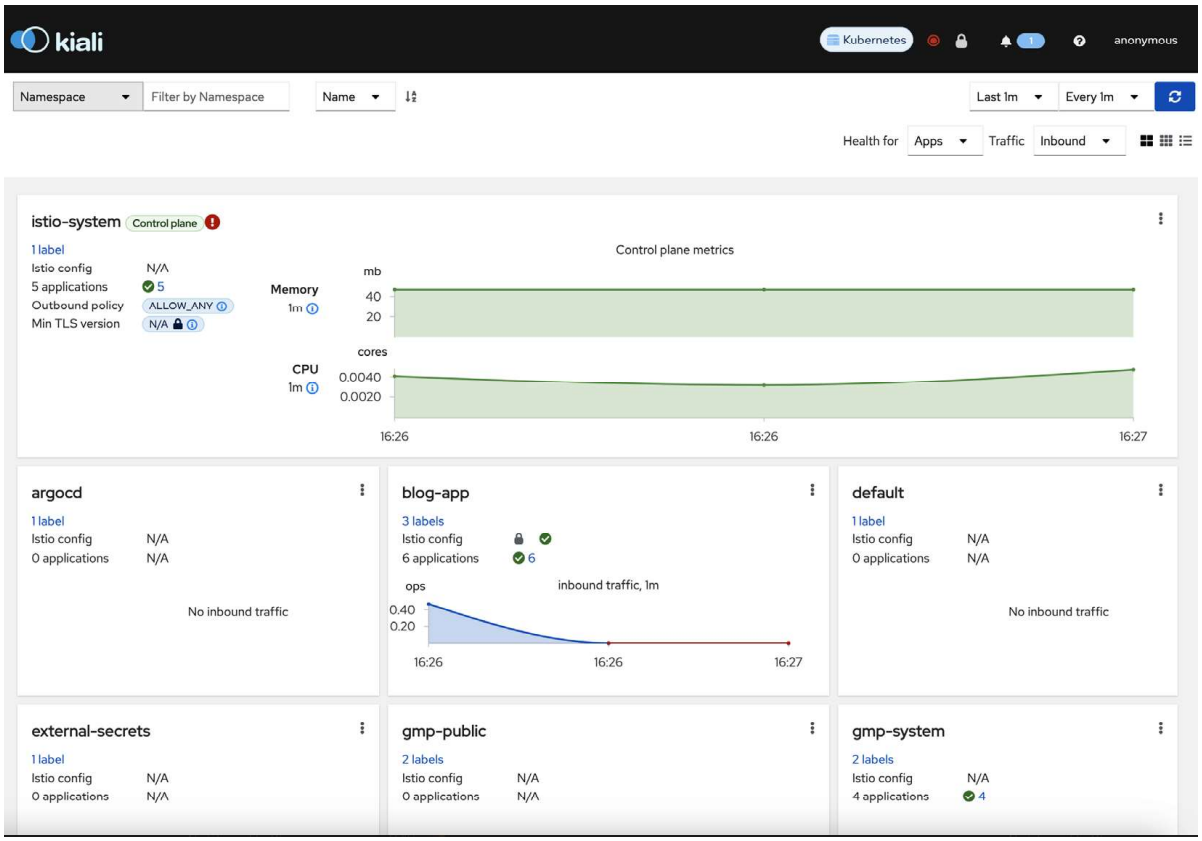

Figure 15.10 – Kiali dashboard

To visualize service interactions, we can switch to the graph view by clicking on the **Graph** tab and selecting the `blog-app` namespace. We will see the following dashboard, which provides an accurate view of how traffic flows, the percentage of successful traffic, and other metrics:

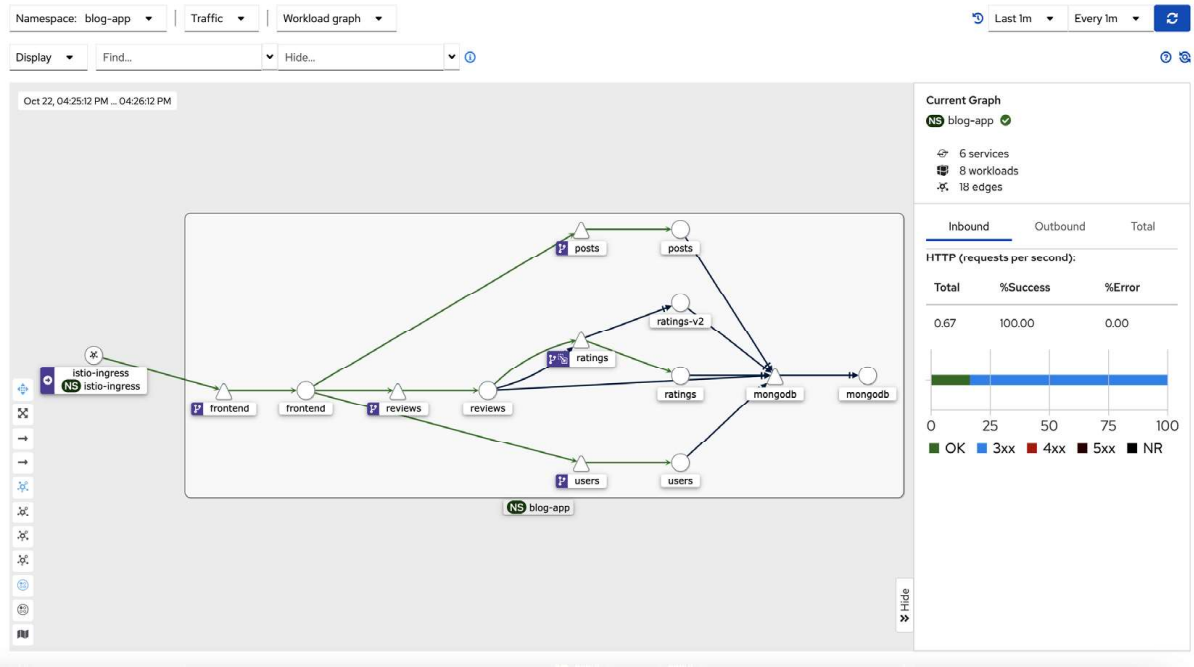

Figure 15.11 – Kiali service interaction graph

While Kiali dashboards provide valuable insights regarding our mesh and help us observe service interactions in real time, they lack the capability of providing us with advanced monitoring and alerting capabilities. For that, we can use Grafana.

Monitoring and alerting with Grafana

Grafana is a leading open source platform for observability and monitoring, offering dynamic dashboards and robust alerting capabilities. It enables users to visualize data from diverse sources while setting up alerts for proactive issue detection.

As we've already installed Grafana with the necessary add-ons, let's access it by opening a port forward session. Ensure you terminate the existing Kiali port-forwarding session or use a different port. Run the following command to do so:

```
$ kubectl port-forward deploy/grafana -n istio-system 20001:3000
```

Once the port forwarding session has started, access the Grafana page like we did for Kiali, and go to **Home** > **Dashboards** > **Istio** > **Istio Service Dashboard**. We should see a dashboard similar to the following:

Figure 15.12 – Istio service dashboard

This dashboard provides rich visualizations regarding some standard SLIs we may want to monitor, such as the request's *success rate*, *duration*, *size*, *volume*, and *latency*. It helps you observe your mesh meticulously, and you can also build additional visualizations based on your requirements by using the **Prometheus Query Language** (**PromQL**), which is simple to learn and apply.

However, monitoring and visualization must be complemented by alerting for complete reliability. So, let's delve into that.

Alerting with Grafana

To initiate the alerting process, it's crucial to establish clear criteria. Given the limited volume at hand, simulating an accurate SLO breach can be challenging. For simplicity, our alerting criteria will trigger when traffic volume surpasses one transaction per second.

The initial phase of this process involves crafting the query to retrieve the necessary metrics. We will employ the following query to achieve this objective:

```
round(sum(irate(istio_requests_total{connection_security_policy="mutual_tls",destination_
service=~"frontend.blog-app.svc.cluster.local",reporter=~"destination",source_
workload=~"istio-ingress",source_workload_namespace=~"istio-ingress"}[5m])) by (source_
workload, source_workload_namespace, response_code), 0.001)
```

The provided query determines the traffic rate for all transactions passing through the Istio ingress gateway to the `frontend` microservice.

The next step involves creating the alert rules with the query in place. To do this, navigate to **Home** > **Alerting** > **Alert rules**. Then, fill in the form, as illustrated in the following screenshot:

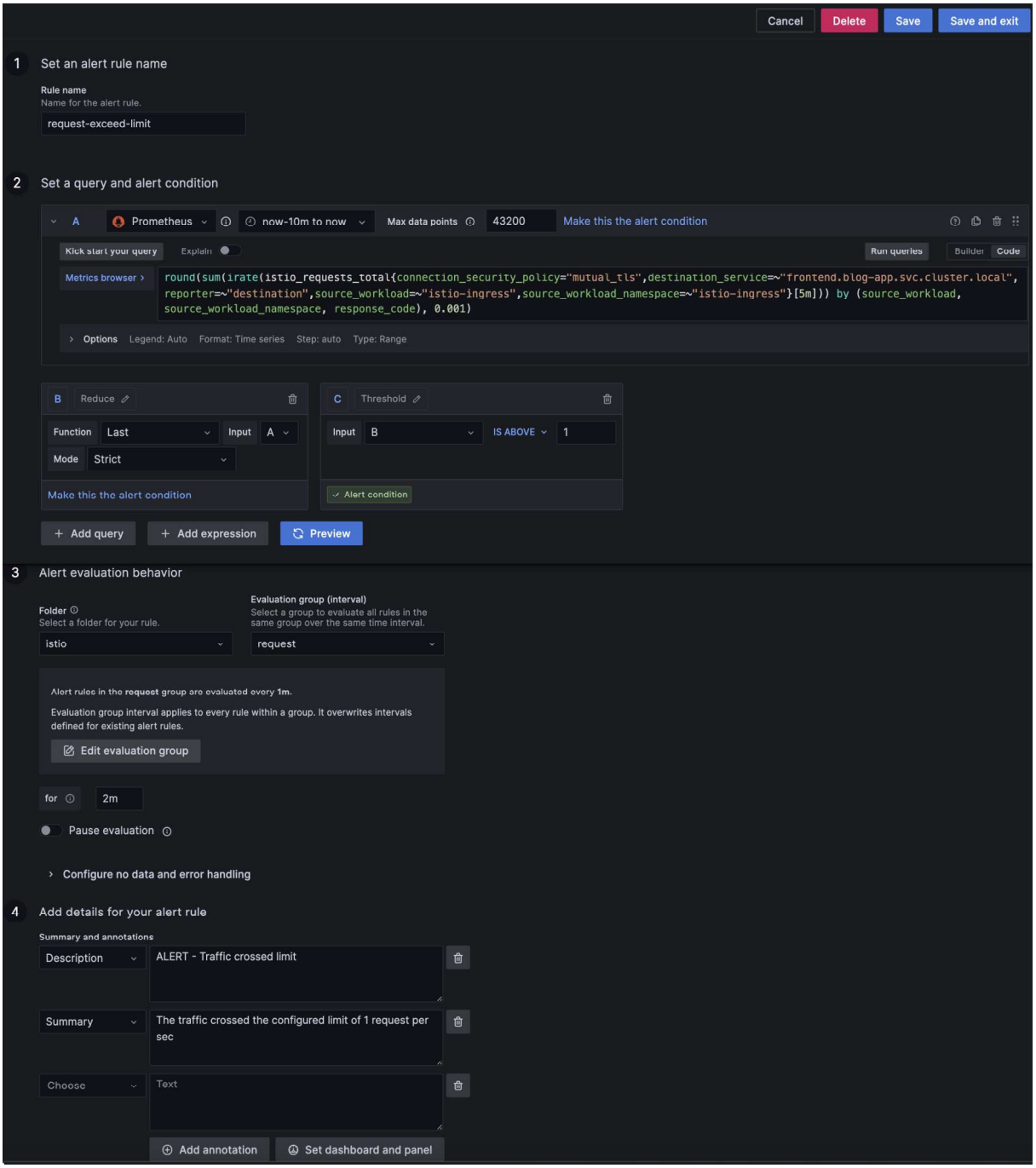

Figure 15.13 – Defining alert rules

The alert rule is configured to monitor for violations at a 1-minute interval for 2 consecutive minutes. Once the alert rule has been established, triggering the alert is as simple as refreshing the Blog App home page about 15–20 times rapidly every 1 to 2 minutes. This action should activate the alert. To observe this process, navigate to **Home** > **Alerting** > **Alert rules**. You will notice the alert in a **Pending** state in the first minute. This means it has detected a violation in one of its checks and will wait for another violation within the 2-minute duration before triggering the alert.

In a production environment, setting longer check intervals, typically around 5 minutes, with alerting intervals of 15 minutes is typical. This approach helps avoid excessive alerting for self-resolving transient issues, ensuring the SRE team is not inundated with false alerts. The goal is to maintain a balance and prevent the team from treating every alert as a potential false alarm. The following screenshot shows a pending alert:

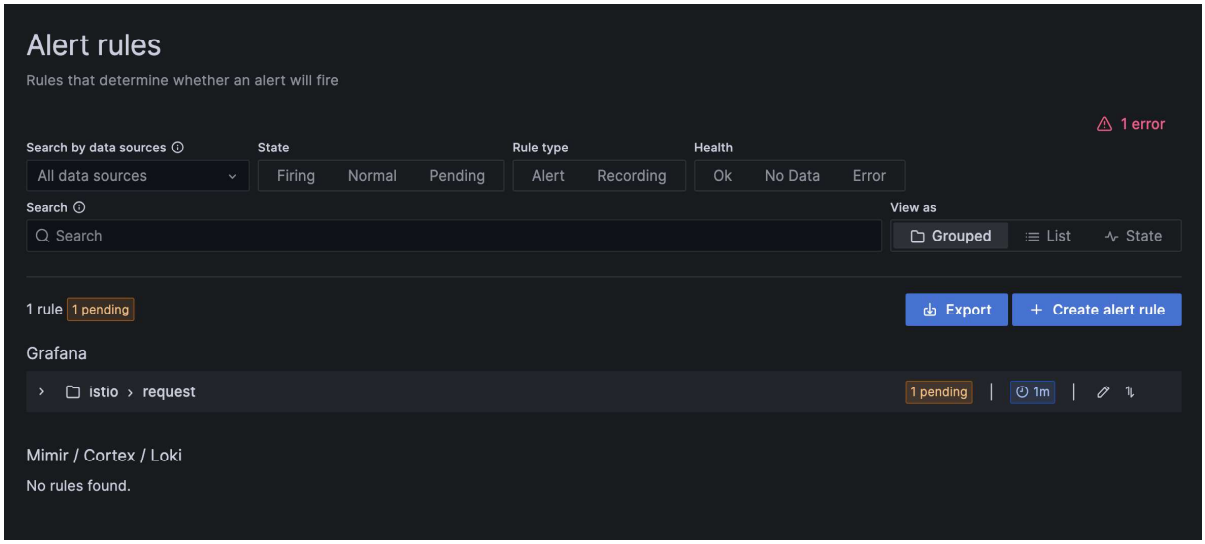

Figure 15.14 – Alert pending

After the 2-minute monitoring period, you should observe the alert being triggered, as depicted in the following screenshot. This indicates that the alert rule has successfully identified a sustained violation of the defined criteria and is now actively notifying relevant parties or systems:

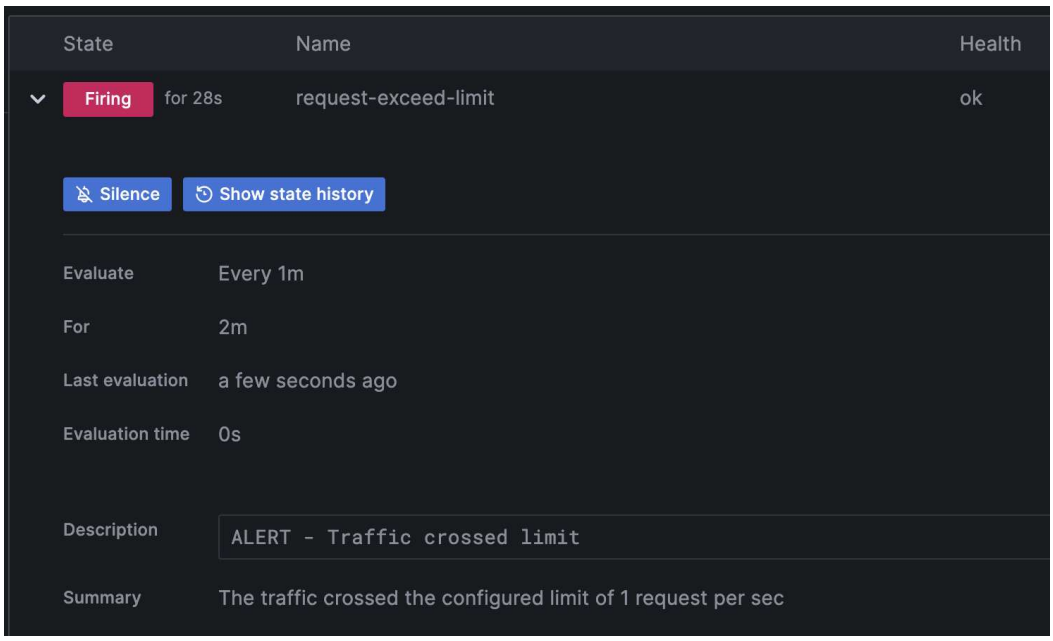

Figure 15.15 – Alert firing

Since no specific alert channels have been configured in this context, the fired alerts will be visible within the Grafana dashboard only. It is highly advisable to set up a designated alert destination for sending alerts to your designated channels, using a tool such as **PagerDuty** to page on-call engineers or **Slack** notifications to alert your on-call team. Proper alert channels ensure that the right individuals or teams are promptly notified of critical issues, enabling rapid response and issue resolution.

Summary

As we conclude this chapter and wrap up this book, our journey has taken us through an array of diverse concepts and functionalities. While we've covered substantial ground in this chapter, it's essential to recognize that Istio is a rich and multifaceted technology, making it a challenge to encompass all its intricacies within a single chapter.

This chapter marked our initiation into the world of service mesh, shedding light on its particular advantages in the context of microservices. Our exploration extended to various dimensions of Istio, beginning with installing Istio and extending our sample Blog App to utilize it using automatic sidecar injection. We then moved on to security, delving into the intricacies of securing ingress gateways with mTLS, enforcing strict mTLS among microservices, and harnessing authorization policies to manage traffic flows.

Our journey then led us to traffic management, where we introduced essential concepts such as destination rules and virtual services. These enabled us to carry out canary rollouts and traffic mirroring, demonstrating the power of controlled deployments and real-time traffic analysis. Our voyage culminated in observability, where we harnessed the Kiali dashboard to visualize service interactions and ventured deep into advanced monitoring and alerting capabilities using Grafana.

As we end this remarkable journey, I want to extend my heartfelt gratitude to you for choosing this book and accompanying me through its pages. I trust you've found every part of this book enjoyable and enlightening. I hope this book has equipped you with the skills necessary to excel in the ever-evolving realm of modern DevOps. I wish you the utmost success in all your present and future endeavors.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Which approach would you use to install Istio among the available options using GitOps methodology?

   A. Istioctl

   B. Helm charts

   C. Kustomize

   D. Manifest bundle

2. What configuration is necessary for Istio to inject sidecars into your workloads automatically?

   A. Apply the `istio-injection-enabled: true` label to the namespace

   B. No configuration is needed – Istio automatically injects sidecars into all pods

   C. Modify the manifests so that they include the Istio sidecars and redeploy them

3. Istio sidecars automatically communicate with each other using mTLS. (True/False)

4. Which resource enforces policies that dictate which services are permitted to communicate with each other?

   A. `AuthenticationPolicy`

   B. `AuthorizationPolicy`

   C. `PeerAuthentication`

5. Which of the following resources would you use for canary rollouts? (Choose two)

   A. `VirtualService`

   B. `IngressGateway`

   C. `DestinationRule`

   D. `Egress Gateway`

6. Why would you use traffic mirroring in production? (Choose three)

   A. Real-time monitoring for production performance and behavior analysis

   B. To route traffic to a new version to duplicate traffic to test the performance of your backend service

C. Safe testing of changes or updates without risking production disruptions

D. Streamlined troubleshooting and debugging for issue identification and resolution

7. Which observability tool would you use to visualize real-time service interactions?

A. Prometheus

B. Grafana

C. Kiali

D. Loki

Answers

Here are the answers to this chapter's questions:

1. B
2. A
3. True
4. B
5. A and C
6. A, C, and D
7. C

Appendix: The Role of AI in DevOps

The recent developments in **artificial intelligence** (**AI**) with the launch of generative AI using ChatGPT have taken the tech industry by storm. It has made many existing AI players pivot, and most companies are now looking at the best ways to use it in their products. Naturally, DevOps and the tooling surrounding it are no exceptions, and slowly, AI is gaining firm ground in this discipline, which historically relied upon more traditional automation methods. Before we delve into how AI changes DevOps, let's first understand what AI is.

This appendix will cover the following topics:

- What is AI?
- The role of AI in the DevOps infinity loop

What is AI?

AI emulates human intelligence in computing. You know how our computers do fantastic things, but they need to be told everything to do? Well, AI doesn't work like that. It learns a ton from looking at lots of information, like how we learn from our experiences. That way, it can figure out patterns independently and make decisions without needing someone to tell it what to do every time. This makes AI intelligent because it can keep learning new things and get better at what it does.

Imagine if your computer could learn from everything it sees, just like you remember from everything around you. That's how AI works—it's a computer's way of getting more intelligent. Instead of needing step-by-step instructions, AI learns from vast amounts of information. This makes it great at spotting patterns in data and deciding things on its own. And when it comes to DevOps, AI can be of great help! Let's look at that next.

The role of AI in the DevOps infinity loop

As we are already aware, instead of following a linear path of software delivery, DevOps practices generally follow an infinity loop, as shown in the following figure:

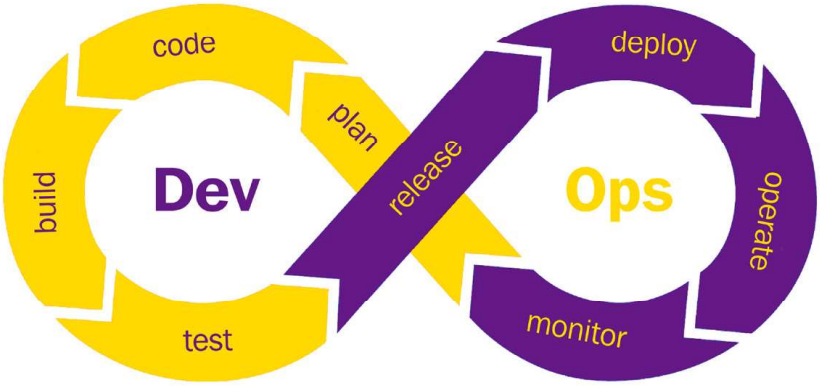

Figure A.1 – DevOps infinity loop

DevOps practices heavily emphasize automation to ensure that this infinity loop operates smoothly, and we need tools. Most of these tools help build, deploy, and operate your software. You will typically start writing code in an **integrated development environment** (**IDE**) and then check code into a central source code repository such as Git. There will be a continuous integration pipeline that will build code from your Git repository and push it to an artifact repository. Your QA team might write automated tests to ensure the artifact is tested before it is deployed to higher environments using a continuous deployment pipeline.

Before the advent of AI, setting up all of the toolchains and operating them relied on traditional coding methods; that is, you would still write code to automate the processes, and the automation would behave more predictably and do what it was told to. However, with AI, things are changing.

AI is transforming DevOps by automating tasks, predicting failures, and optimizing performance. In other words, by leveraging AI's capabilities, DevOps teams can achieve greater efficiency, reduce errors, and deliver software faster and more reliably.

Here are some key roles of AI in DevOps:

- **Automating Repetitive Tasks**: AI can automate repetitive and tedious tasks, such as code testing, deployment, and infrastructure provisioning. This frees up DevOps engineers to focus on more strategic and creative work, such as developing new features and improving application performance.
- **Predicting and Preventing Failures**: AI can analyze vast amounts of data, including logs, performance metrics, and user feedback, to identify patterns and predict potential failures. This proactive approach allows DevOps teams to address issues before they impact users or cause major disruptions.

- **Optimizing Resource Utilization**: AI can analyze resource usage data to optimize infrastructure allocation and prevent resource bottlenecks. This ensures that applications have the resources they need to perform optimally, minimizing downtime and improving overall system efficiency.
- **Enhancing Security**: AI can be used to detect and prevent security threats by analyzing network traffic, identifying anomalous behavior, and flagging suspicious activity. This helps DevOps teams maintain a robust security posture and protect sensitive data.
- **Improving Collaboration and Communication**: AI can facilitate collaboration and communication among DevOps teams by providing real-time insights, automating workflows, and enabling seamless communication channels. This breaks down silos and promotes a more cohesive DevOps culture.

Let's look at the areas of the DevOps infinity loop and see how AI impacts them.

Code development

This area is where we see the most significant impact of generative AI and other AI technologies. AI revolutionizes code development by automating tasks such as code generation, bug detection, optimization, and testing. Through autocomplete suggestions, bug detection algorithms, and predictive analytics, AI accelerates coding, enhances code quality, and ensures better performance while aiding in documentation and code security analysis. Its role spans from assisting in writing code to predicting issues, ultimately streamlining the software development life cycle, and empowering developers to create more efficient, reliable, and secure applications.

Many tools employ AI in code development, and one of the most popular tools in this area is **GitHub Copilot**.

GitHub Copilot is a collaborative effort between **GitHub** and **OpenAI**, introducing a code completion feature that utilizes OpenAI's **Codex**. Codex, trained on vast code repositories from GitHub, quickly generates code based on the current file's content and cursor location. Compatible with popular code editors such as **Visual Studio Code**, **Visual Studio**, **Neovim**, and **JetBrains IDEs**, Copilot supports languages such as **Python**, **JavaScript**, **TypeScript**, **Ruby**, and **Go**.

Praised by GitHub and users alike, Copilot generates entire code lines, functions, tests, and documentation. Its functionality relied on the context provided and the extensive code contributions by developers on GitHub, regardless of their software license. Dubbed the world's first AI pair programmer by **Microsoft**, it is a paid tool and charges a subscription fee of $10 per month or $100 per year per user after a 60-day trial period.

With Copilot, you can start by writing comments on what you intend to do, and it will generate the required code for you. This speeds up development many times, and most of the time, you just need to review and test your code to see whether it does what you intend it to do. A great power indeed! It can optimize existing code and provide feedback by generating code snippets. It can also scan your code for security vulnerabilities and suggest alternative approaches.

If you don't want to pay that $10, you can also look at free alternatives such as **Tabnine**, **Captain Stack**, **GPT-Code Clippy**, **Second Mate**, and **Intellicode**. Paid alternatives include Amazon's **Code Whisperer** and Google's **ML-enhanced code completion**.

AI tools not only help enhance the development workflow but also help in software testing and quality assurance. Let's look at that next.

Software testing and quality assurance

Traditionally, software testing has taken more of a manual approach because most developers don't want software testing as a full-time profession. Though automation testing has gained ground recently, the knowledge gap has hindered this process in most organizations. Therefore, AI would most impact the testing function as it bridges the human-machine gap.

AI-integrated testing techniques revolutionize every stage of the **software testing life cycle** (**STLC**); some of them are as follows:

- **Test script generation**: Traditionally, creating test scripts was time-consuming, involving deep system understanding. AI and **machine learning** (**ML**) now expedite this process by analyzing requirements, existing test cases, and application behavior to craft more optimized test scripts, offering ready-to-use templates with preconfigured code snippets and comprehensive comments, and translating plain language instructions into complete test scripts using **natural language processing** (**NLP**) techniques.
- **Test data generation**: AI-equipped testing tools provide detailed and ample test data for comprehensive coverage. They achieve this by generating synthetic data from existing sets for specific test objectives, transforming data to create diverse testing scenarios, refining existing data for higher precision and relevance, and scanning large code bases for context comprehension.
- **Intelligent test execution**: AI alleviates test execution challenges by automatically categorizing and organizing test cases, efficiently selecting tests for various devices, operating systems, and configurations, and smartly executing regression tests for critical functionalities.
- **Intelligent test maintenance**: AI/ML minimizes test maintenance challenges by implementing self-healing mechanisms to handle broken selectors and analyzing UI and code change relationships to identify affected areas.
- **Root cause analysis**: AI aids in understanding and rectifying issues by analyzing logs, performance metrics, and anomalies to pinpoint impact areas, tracing issues back to affected user stories and feature requirements, and utilizing knowledge repositories for comprehensive root cause analysis.

Multiple tools in the market help you achieve all of it; some of the most popular ones are the following:

- **Katalon platform**: A comprehensive quality management tool that simplifies test creation, execution, and maintenance across various applications and environments. It boasts AI features such as **TrueTest**, **StudioAssist**, **self-healing**, **visual testing**, and **AI-powered test failure analysis**.
- **TestCraft**: Built on **Selenium**, TestCraft offers both manual and automated testing capabilities with a user-friendly interface and AI-driven element identification, allowing tests to run across multiple browsers in parallel.
- **Applitools**: Known for its AI-based visual testing, Applitools efficiently identifies visual bugs, monitors app visual aspects, and provides accurate visual test analytics using AI and ML.
- **Function**: Utilizes AI/ML for functional, performance, and load testing with simplicity, allowing test creation through plain English input, self-healing, test analytics, and multi-browser support.
- **Mabl**: An AI-powered tool offering low-code testing, intuitive intelligence, data-driven capabilities, end-to-end testing, and valuable insights generation, promoting team collaboration.
- **AccelQ**: Automates test designs, plans, and execution across the UI, mobile, API, and PC software, featuring **automated test generation**, **predictive analysis**, and comprehensive test management.
- **Testim**: Uses ML to expedite test creation and maintenance, allowing for quick end-to-end test creation, smart locators for resilient tests, and a blend of recording functions and coding for robust test creation.

As we've already seen the benefits of AI in development and testing, let's move on to software delivery.

Continuous integration and delivery

In **continuous integration** (**CI**) and **continuous delivery** (**CD**), AI brings a transformative edge by optimizing and automating various stages of the software development pipeline. AI augments CI by automating code analysis, identifying patterns, and predicting potential integration issues. It streamlines the process by analyzing code changes, suggesting appropriate test cases, and facilitating faster integration cycles. Through ML, AI can understand historical data from past builds, recognizing patterns that lead to failures, thereby aiding in more efficient debugging and code quality improvement.

In CD, AI optimizes deployment pipelines by automating release strategies, predicting performance bottlenecks, and suggesting optimizations for smoother delivery. It analyzes deployment patterns, user feedback, and system performance data to recommend the most efficient delivery routes. Additionally, AI-driven CD tools enhance risk prediction, allowing teams to foresee potential deployment failures and make informed decisions to mitigate risks before they impact production environments. Ultimately, AI's role in CI/CD accelerates the development cycle, improves software quality, and enhances the reliability of software releases.

Here are some AI-powered tools used in software release and delivery:

- **Harness**: Harness utilizes AI to automate software delivery processes, including continuous integration, deployment, and verification. It employs ML to analyze patterns from deployment pipelines, predict potential issues, and optimize release strategies for better efficiency and reliability.
- **GitClear**: GitClear employs AI algorithms to analyze code repositories and provides insights into developer productivity, code contributions, and team performance. It helps understand code base changes, identify bottlenecks, and optimize development workflows.
- **Jenkins**: Thanks to its plugin-based architecture, Jenkins, a widely used automation server, employs a lot of AI plugins and extensions to enhance its capabilities in CI/CD. AI-powered plugins help automate tasks, optimize build times, and predict build failures by analyzing historical data.
- **CircleCI**: CircleCI integrates AI and ML to optimize CI/CD workflows. It analyzes build logs, identifies patterns leading to failures, and provides recommendations to improve build performance and reliability.

These AI-powered tools improve software release and delivery processes' speed, quality, and reliability by automating tasks, optimizing workflows, predicting issues, and providing valuable insights for better decision-making.

Now, let's look at the next stage in the process—software operations.

Software operations

AI is pivotal in modern software operations, revolutionizing how systems are monitored, managed, and optimized. By leveraging ML algorithms, AI helps automate routine tasks such as monitoring system performance, analyzing logs, and identifying anomalies in real time. It enables predictive maintenance by detecting patterns that precede system failures, allowing for proactive intervention and preventing potential downtime. Additionally, AI-powered tools streamline incident management by correlating alerts, prioritizing critical issues, and providing actionable insights, enhancing the overall resilience and reliability of software operations.

Moreover, AI augments decision-making processes by analyzing vast amounts of data to identify trends, forecast resource requirements, and optimize infrastructure utilization. AI adapts to changing environments through its continuous learning capabilities, enabling software operations teams to stay ahead of evolving challenges and complexities. Overall, AI's role in software operations ensures greater efficiency, improved system performance, and proactive problem resolution, contributing significantly to the seamless functioning of IT infrastructures.

Here are some AI-powered tools used in software operations:

- **Dynatrace**: Dynatrace utilizes AI for application performance monitoring and management. It employs AI algorithms to analyze vast amounts of data, providing real-time insights into application performance, identifying bottlenecks, and predicting potential issues before they impact end users.
- **PagerDuty**: PagerDuty integrates AI-driven incident management, alerting, and on-call scheduling. It uses ML to correlate events and alerts, reducing noise and providing intelligent notifications for critical incidents.
- **Opsani**: Opsani leverages AI for autonomous optimization of cloud applications. It analyzes application performance, dynamically adjusts configurations, and optimizes resources to maximize performance and cost-efficiency.
- **Moogsoft**: Moogsoft offers AI-driven IT operations and AIOps platforms. It uses ML to detect anomalies, correlate events, and automate incident resolution, helping teams proactively manage and resolve issues in complex IT environments.
- **Sumo Logic**: Sumo Logic employs AI for log management, monitoring, and analytics. It uses ML to identify patterns, anomalies, and security threats within logs and operational data, enabling proactive troubleshooting and security incident detection.
- **New Relic**: New Relic utilizes AI for application and infrastructure monitoring. Its AI-powered platform helps identify performance issues, predict system behavior, and optimize resource utilization for better application performance.
- **LogicMonitor**: LogicMonitor uses AI for infrastructure monitoring and observability. It analyzes metrics and performance data to provide insights into system health, predict potential issues, and optimize resource allocation in complex environments.
- **OpsRamp**: OpsRamp employs AI for IT operations management, offering capabilities for monitoring, incident management, and automation. It uses ML to detect anomalies, automate routine tasks, and optimize workflows for better operational efficiency.

These AI-powered tools assist in automating tasks, predicting and preventing issues, optimizing resource allocation, and enhancing overall system reliability in software operations.

The integration of AI into DevOps practices is still in its early stages, but its potential impact is significant. By automating tasks, optimizing processes, and enhancing collaboration, AI can revolutionize the way software is developed, deployed, and managed. As AI technology continues to develop, we can expect to see even more ways in which AI is used to improve the DevOps process.

Summary

AI revolutionizes DevOps practices by infusing intelligence into every development and operations cycle stage. It streamlines processes, enhances efficiency, and ensures smoother collaboration between development and operations teams. AI automates routine tasks, predicts potential bottlenecks, and optimizes workflows, transforming how software is built, tested, deployed, and monitored. From automating code analysis to predicting system failures, AI empowers DevOps by enabling quicker decision-making, reducing errors, and fostering a more agile and responsive software development environment.

In essence, AI acts as a silent partner, continuously learning from data, suggesting improvements, and helping DevOps teams foresee and address issues before they impact the software's performance. It's the catalyst that drives agility and innovation, allowing DevOps to evolve from a mere collaboration between teams to a symbiotic relationship where AI enhances the capabilities of both development and operations, paving the way for more efficient and reliable software delivery.