Using Serverless Strategy with Gloo and Knative (GKE Only)

This lesson discusses the serverless strategy and how to use it with Gloo and Knative. At the end of lesson we see if the serverless strategy has fulfilled our needs.

We'll cover the following

Judging by the name of this section, you might be wondering why we start with serverless deployments. We're starting with serverless simply because that's what we used in the previous chapter. So, we'll start with what we have right now, at

least for those who are running Jenkins X in GKE.

Another question you might be asking is why we cover serverless with **Knative** given that we already discussed it in the previous chapter. The answer to that question lies in *completeness*. Serverless deployments are one of the essential options we have when choosing the strategy, and this chapter would not be complete without it. If you did go through the previous chapter, consider this one a refresher with the potential to learn something new. If nothing else, you'll get a better understanding of the flow of events with **Knative** and to see a few diagrams. In any case, the rest of the strategies will build on top of this one. On the other hand, you might be impatient and bored with repetition. If that's the case, feel free to skip this section altogether.

> ⚠ At the time of this writing (September 2019), serverless deployments with **Knative** work out-of-the-box only in GKE (issue 4668). That does not mean that **Knative** does not work in other Kubernetes flavors, but rather that the Jenkins X installation of **Knative** works only in GKE. I encourage you to set up **Knative** yourself and follow along in your Kubernetes flavor. If you cannot run **Knative**, I still suggest you stick around even if you cannot run the examples. I'll do my best to be brief and make the examples clear even for those not running them.

Instead of discussing the pros and cons first, we'll start each strategy with an example. We'll observe the results and, comment on their advantages and disadvantages as well as the scenarios when they might be a good fit. In that spirit, let's create a serverless deployment and see what we get.

Checking the `ksvc.yaml` file

Let's go into the project directory and take a quick look at the definition that makes the application serverless.

```
cd jx-progressive

cat charts/jx-progressive/templates/ksvc.yaml
```

We won't go into details on **Knative** specification. It was briefly explained in the Using Jenkins X To Define And Run Serverless Deployments chapter and details

can be found in the official docs. What matters in the context of the current

discussion is that the `YAML` you see in front of you defined a serverless deployment using **Knative**.

Inspecting *jx-progressive* activities

By now, if you created a new cluster, the application we imported should be up-and-running. But, to be on the safe side, we'll confirm that by taking a quick look at the *jx-progressive* activities.

> ⚠ There's no need to inspect the activities to confirm whether the build is finished if you are reusing the cluster from the previous chapter. The application we deployed previously should still be running.

```
jx get activities \
    --filter jx-progressive \
    --watch
```

Once you confirm that the build is finished press *ctrl+c* to stop watching the activities.

Inspecting *environment-jx-rocks-staging* activities

As you probably already know, the activity of an application does not wait until the release is promoted to the staging environment. So, we'll confirm that the build initiated by changes to the *environment-jx-rocks-staging* repository is finished as well.

```
jx get activities \
    --filter environment-jx-rocks-staging/master \
    --watch
```

Just as before, feel free to press *ctrl+c* once you confirm that the build was finished.

Checking the Pods

Finally, the last verification we'll do is to confirm that the Pods are running.

```
kubectl --namespace jx-staging \
    get pods
```

The output is as follows:

```
NAME                  READY STATUS  RESTARTS AGE
jx-jx-progressive-... 2/2   Running 0        45s
```

In your case, `jx-progressive` deployment might not be there. That means it's been a while since you used the application and **Knative** made the decision to scale it to zero replicas. We'll go through a scaling-to-zero example later. For now, imagine that you do have that Pod running.

Observations gathered from the Pods

At first look, everything looks normal. It is as if the application was deployed like any other. The only strange thing we notice is the name of the Pod created through the *jx-progressive* Deployment and that it contains two containers instead of one. We'll ignore the naming strangeness and focus on the latter observation.

Knative injected a container into our Pod. It contains `queue-proxy` that, as the name suggests, serves as a proxy responsible for the request queue parameters. It also reports metrics to the Autoscaler through which it might scale up or down depending on the number of different parameters. Requests are not going directly to our application but through this container.

Sending a request to the application

Now, let's confirm that the application is accessible from outside the cluster.

```
STAGING_ADDR=$(kubectl \
    --namespace jx-staging \
    get ksvc jx-progressive \
    --output jsonpath="{.status.url}")

curl "$STAGING_ADDR"
```

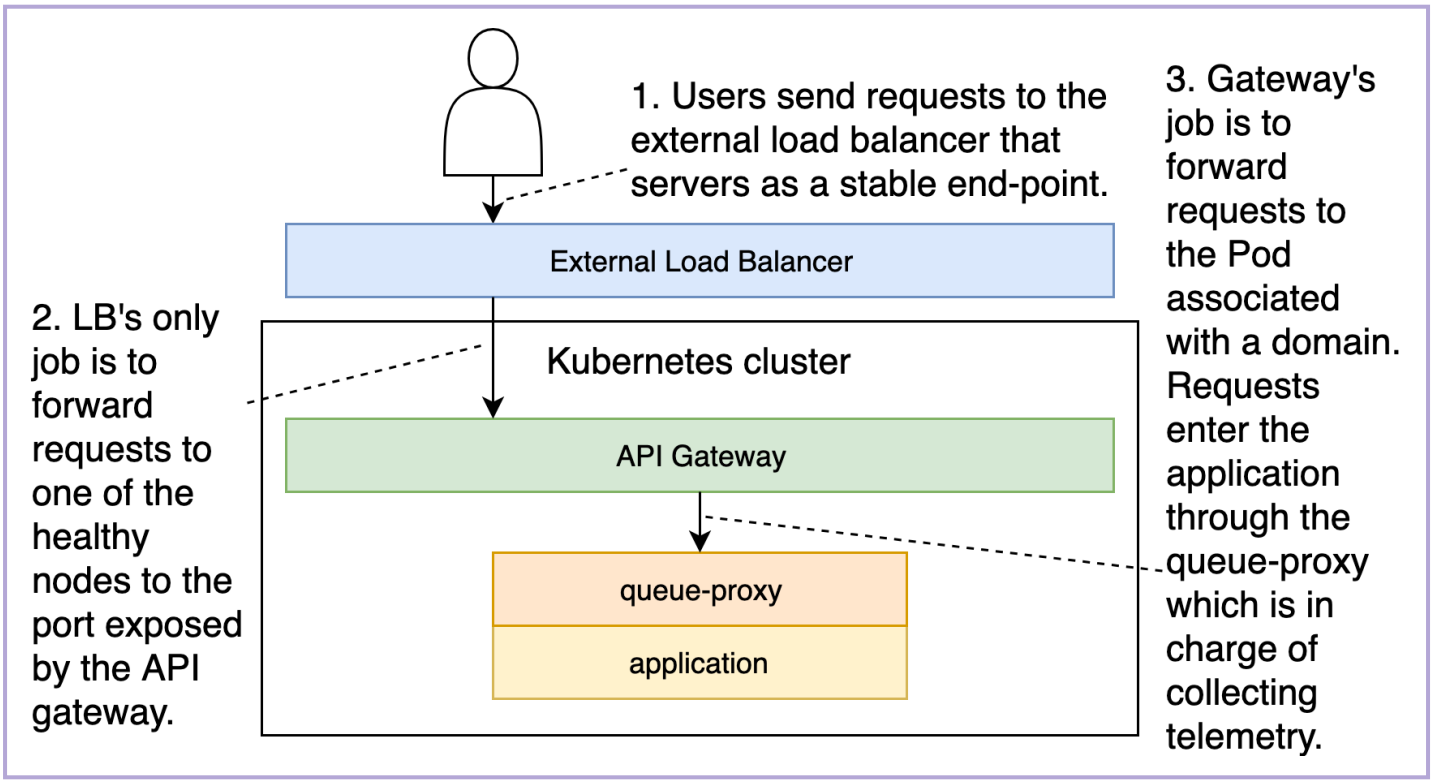
We retrieved the address where we can reach the application running in the staging environment, and we used `curl` to send a request. The output should be `Hello from: Jenkins X golang http example`.

The flow of a request

So far, the significant difference when compared with normal Kubernetes

deployments is that the access to the application is not controlled through **Ingress**

anymore. Instead, it goes through a new resource type abbreviated as `ksvc` (short for **Knative Service**). Apart from that, everything else seems to be the same, except if we left the application unused for a while. If that's the case, we still get the same output, but there was a slight delay between sending the request and receiving the response. The reason for such a delay lies in **Knative**'s scaling capabilities. It saw that the application is not being used and scaled it to zero replicas. But, the moment we sent a request, it noticed that zero replicas is not the desired state and scaled it back to one replica. All in all, the request entered into a gateway (in our case served by **Gloo Envoy**) and waited there until a new replica was created and initialized. After that, it forwarded the request to it, and the rest is the standard process of our application responding.



The flow of a request with API gateway

I cannot be sure whether your serverless deployment indeed scaled to zero or it didn't. So, we'll use a bit of patience to validate that it does indeed scale to nothing after a bit of inactivity. All we have to do is wait for five to ten minutes. Get a coffee or some snack.

Checking the Pods again after some time

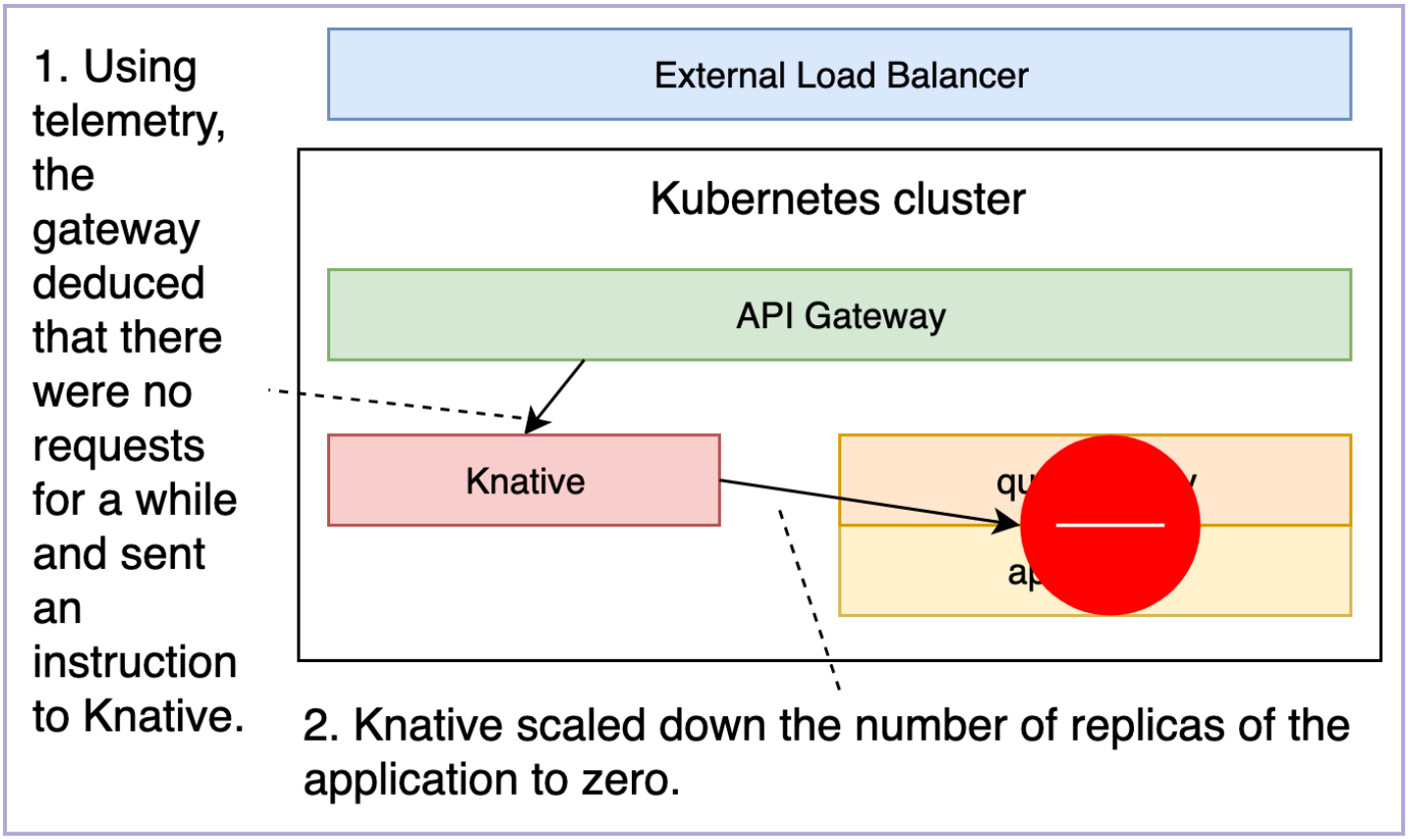
```
kubectl --namespace jx-staging \
    get pods
```

Assuming that sufficient time has passed, the output should state that `no resources` were `found` in the Namespace, unless you have other applications there. The application is now gone. If we ignore the other resources and focus only on the Pods, it seems like the application is wiped out completely. That is true in terms that nothing application-specific is running. All that's left are a few Knative definitions and the common resources used for all applications.

> 🔍 If you still see the *jx-progressive* Pod, all I can say is that you are impatient, and you didn't wait long enough. If that's what happened, wait for a while longer and repeat the `get pods` command.

Knative's ability to scale to zero replicas

Using telemetry collected from all the Pods deployed as **Knative** applications, **Gloo** detected that no requests were sent to *jx-progressive* for a while and decided that the time has come to scale it down. It sent a notification to **Knative** that executed a series of actions that resulted in our application being scaled to zero replicas.



Knative's ability to scale to zero replicas

Bear in mind that the actual process is more complicated than that and that there are quite a few other components involved. Nevertheless, for the sake of brevity, the simplistic view we presented should suffice. I'll leave it up to you to go deeper into **Gloo** and **Knative** or accept it as magic. In either case, our application was successfully scaled to zero replicas. We started saving resources that could be better used by other applications and save us some costs in the process.

If you've never used serverless deployments or if you've never worked with **Knative**, you might think that your users would not be able to access it once the application stops running. You might think that it will be scaled up once requests start coming in, but you might be scared that you'll lose those sent before the new replica starts running. You might have read the previous chapter and know that those fears are unfounded. In any case, we'll put that to the test by sending three hundred concurrent requests for twenty seconds.

Running Siege to send concurrent requests

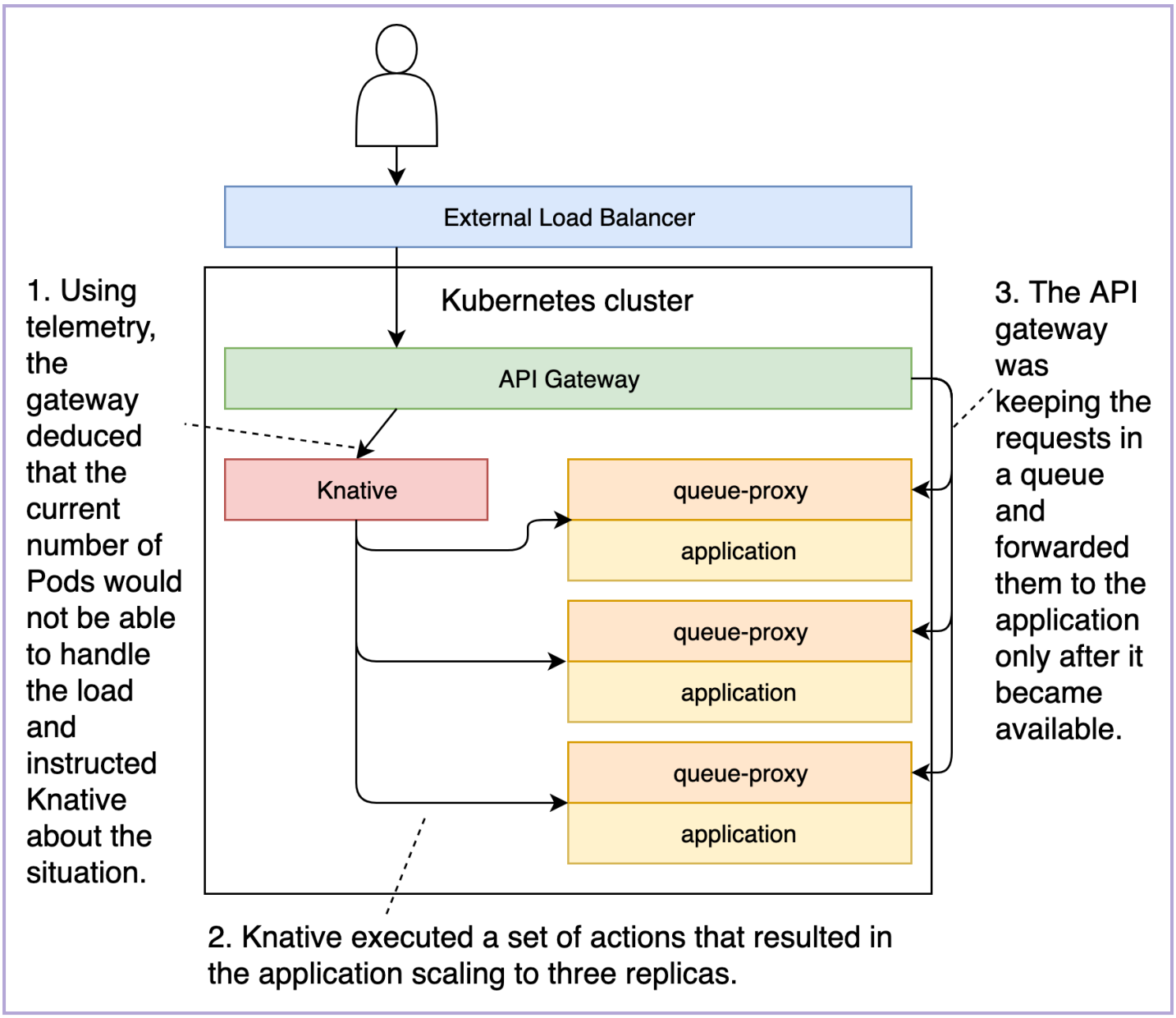
```
kubectl run siege \
    --image yokogawa/siege \
    --generator "run-pod/v1" \
    -it --rm \
    -- --concurrent 300 --time 20S \
    "$STAGING_ADDR" \
    && kubectl \
    --namespace jx-staging \
    get pods
```

We won't go into details about **Siege**. Read the previous chapter if you want to know more. What matters is that we finished sending a lot of requests and that the previous command retrieved the Pods in the staging namespace. That output is as follows:

```
...
NAME                                READY STATUS  RESTARTS AGE
jx-progressive-dzghl-deployment-... 2/2   Running 0        19s
jx-progressive-dzghl-deployment-... 2/2   Running 0        16s
jx-progressive-dzghl-deployment-... 2/2   Running 0        18s
jx-progressive-dzghl-deployment-... 2/2   Running 0        16s
```

Knative's ability to scale from zero to multiple replicas

Our application is up-and-running again! Not only that, but it was scaled to three replicas to accommodate the high number of concurrent requests.



Knative's ability to scale from zero to multiple replicas

Does serverless deployment fit our needs?

What did we learn from serverless deployments in the context of our quest to find one that fits our needs the best?

High Availability

High availability is easy in Kubernetes, as long as our applications are designed with that in mind. What that means is that our apps should be scalable and should not contain state. If they cannot be scaled, they cannot be highly available. When a replica fails (note that I did not say *if* but *when*), no matter how fast Kubernetes will reschedule it somewhere else, there will be downtime, unless other replicas take over its load. If there are no other replicas, we are bound to have downtime

both due to failures but also whenever we deploy a new release. So, scalability

(running more than one replica) is the prerequisite for high availability. At least, that's what logic might make us think.

In the case of serverless deployments with **Knative**, not having replicas that can respond to user requests is not an issue, at least not from the high availability point of view. While in a "normal" situation, the requests would fail to receive a response, in our case, they were queued in the gateway and forwarded after the application is up-and-running. So, even if the application is scaled to zero replicas (if nothing is running), we are still highly available. The major downside is in potential delays between receiving the first requests and until the first replica of the application is responsive.

Responsiveness

The problem we might have with serverless deployments, at least when used in Kubernetes, is responsiveness. If we keep the default settings, our application will scale to zero if there are no incoming requests. As a result, when someone does send a request to our app, it might take longer than usual until the response is received. That could be a couple of milliseconds, a few seconds, or much longer. It all depends on the size of the container image, whether it is already cached on the node where the Pod is scheduled, the amount of time the application needs to initialize, and quite a few other criteria. If we do things right, that delay can be short. Still, any delay reduces the responsiveness of our application, no matter how short or long it is. What we need to do is compare the pros and cons. The results will differ from one app to another.

Let's take the static Jenkins as an example. In many organizations, it is under heavy usage throughout working hours, and with low or no usage at night. We can say that half of the day it is not used. What that means is that we are paying double to our hosting vendor. We could have shut it down overnight and potentially remove a node from the cluster due to decreased resource usage. Even if the price is not an issue, surely those resources reserved by inactive Jenkins could be better used by some other processes. Shutting down the application would be an improvement, but it would also produce potentially very adverse effects.

What if someone is working overnight and pushes a change to Git. A webhook would fire trying to notify Jenkins that it should run a build. But, such a webhook

would fail if there is no Jenkins to handle the request. A build would never be executed. Unless we set up a policy that says *"you are never allowed to work after 6 pm, even if the whole system crashed"*, having a non-responsive system is unacceptable.

Another issue would be to figure out when our system is not in use. If we continue using the *"traditional"* Jenkins as an example, we could say that it should shut-down at 9 pm. If our official working hours end at 6 pm, that will provide three hours margin for those who do stay in the office longer. But, that would still be a suboptimal solution. During much of those three hours, Jenkins would not be used, and it would continue wasting resources. On the other hand, there is still no guarantee that no one will ever push a change after 9 pm.

Knative solves those and quite a few other problems. Instead of shutting down our applications at predefined hours and hoping that no one is using them while they are unavailable, we can let **Knative** (together with **Gloo** or **Istio**) monitor requests. It would scale down if a certain period of inactivity passed. On the other hand, it would scale back up if a request is sent to it. Such requests would not be lost but queued until the application becomes available again.

All in all, I cannot say that **Knative** might result in non-responsiveness. What I can say is that it might produce slower responses in some cases (between having none and having some replicas). Such periodical slower responsiveness might produce a less negative effect than the good it brings. *Is it such a bad thing if static Jenkins takes an additional ten seconds to start building something after a whole night of inactivity?* Even a minute or two of delay is not a big deal. On the other hand, in that particular case, the upside outweighs the downsides. Still, there are even better examples of the advantages of serverless deployments than Jenkins.

Preview environments might be the best example of wasted resources. Every time we create a pull request, a release is deployed into a temporary environment. That, by itself, is not a waste. The benefits of being able to test and review an application before merging it to master outweigh the fact that most of the time we are not using those applications. Nevertheless, we can do better. Just as we explained in the previous chapter, we can use **Knative** to deploy to preview environments, no matter whether we use it for permanent environments like staging and production. After all, preview environments are not meant to provide a place to test something before promoting it to production (staging does that). Instead, they provide us with relative certainty that what we'll merge to the master branch is

likely code that works well.

If the response delay caused by scaling up from zero replicas is unacceptable in certain situations, we can still configure **Knative** to have one or more replicas as a minimum. In such a case, we'd still benefit from **Knative** capabilities. For example, the metrics it uses to decide when to scale might be easier or better than those provided by **HorizontalPodAutoscaler (HPA)**. Nevertheless, the result of having **Knative** deployment with a minimum number of replicas above zero is similar to the one we'd have with using HPA. So, we'll ignore such situations since our applications would not be serverless. That is not to say that **Knative** is not useful if it doesn't scale to zero. What it means is that we'll treat those situations separately and stick to serverless features in this section.

What's next in our list of deployment requirements?

Progressive rollout

Even though we did not demonstrate it through examples, serverless deployments with **Knative** do not produce downtime when deploying new releases. During the process, all new requests are handled by the new release. At the same time, the old ones are still available to process all those requests that were initiated before the new deployment started rolling out. Similarly, if we have health checks, it will stop the rollout if they fail. In that aspect, **we can say that rollout is progressive**.

Rollback

On the other hand, it is not a "true" progressive rollout but similar to those we get with rolling updates. **Knative**, by itself, cannot choose whether to continue progressing with a deployment based on arbitrary metrics. Similarly, it cannot roll back automatically if predefined criteria are met. Just like rolling updates, it will stop the rollout if health checks fail, and not much more. If those health checks fail with the first replica, even though there is no rollback, all the requests will continue being served with the old release. Still, there are too many ifs in those statements. We can only say that serverless deployments with **Knative** (without additional tooling) partially fulfill the progressive rollout requirement and that they are incapable of automated rollbacks.

Cost-effectiveness

Finally, the last requirement is that our deployment strategy should be cost-

effective. Serverless deployments, at least those implemented with **Knative**, are probably the most cost-effective deployments we can have. Unlike vendor-specific serverless implementations like **AWS Lambda**, **Azure Functions**, and **Google Cloud**'s serverless platform, we are in (almost) full control. We can define how many requests are served by a single replica. We control the size of our applications given that anything that can run in a container can be serverless (but is not necessarily a good candidate). We control which metrics are used to make decisions and what the thresholds are. Truth be told, that is likely more complicated than using vendor-specific serverless implementations. It's up to us to decide whether additional complications with **Knative** outweigh the benefits it brings. I'll leave such a decision in your hands.

Conclusion

So, what did we conclude? Do serverless deployments with **Knative** fulfill all our requirements? The answer to that question is a resounding "no". No deployment strategy is perfect. Serverless deployments provide **huge benefits** with **high-availability** and **cost-effectiveness**. They are **relatively responsive and offer a certain level of progressive rollouts**. The major drawback is the **lack of automated rollbacks**.

Requirement	Fullfilled
High-availability	Fully
Responsiveness	Partly
Progressive rollout	Partly
Rollback	Not
Cost-effectiveness	Fully

 Please note that we used **Gloo** in conjunction with **Knative** to perform serverless deployments. We could have used **Istio** instead of **Gloo**. Similarly, we could have used **OpenFaaS** instead of **Knative**. Or we could have opted

> we could have used **OpenFaaS** instead of **Knative**. Or we could have opted for something completely different. There are many different solutions we could assemble to make our applications serverless. Still, the goal was not to compare them all and choose the best one. Instead, we explored serverless deployments in general as one possible strategy we could employ. I do believe that **Knative** is the most promising one, but we are still in early stages with serverless in general and especially in Kubernetes. It would be impossible to be sure of what will prevail. Similarly, for many engineers, **Istio** would be the service mesh of choice due to its high popularity. I chose **Gloo** mostly because of its simplicity and its small footprint. For those of you who prefer **Istio**, all I can say is that we will use it for different purposes later on in this chapter.

Finally, I decided to present only one serverless implementation mostly because it would take much more than a single chapter to compare all those that are popular. The same can be said for service mesh (**Gloo**). Both are fascinating subjects that I might explore in the next book. But, at this moment I cannot make that promise because I do not plan a new book before the one I'm writing (this one) is finished.

What matters is that we've finished with a very high-level exploration of the pros and cons of using serverless deployments and now we can move into the next one. But, before we do that, we'll revert our chart to the good old Kubernetes Deployment.

Reverting charts to Kubernetes Deployment

```
jx edit deploy \
    --kind default \
    --batch-mode

cat charts/jx-progressive/values.yaml \
    | grep knative
```

We edited the deployment strategy by setting it to `default` (it was `knative` so far). Also, we output the `knative` variable to confirm that it is now set to `false`.

The last thing we'll do is go out of the local copy of the *jx-progressive* directory. That way we'll be in the same place as those who could not follow the examples because their cluster cannot yet run Knative or those who were too lazy to set it up.

```
cd ..
```

Next, we are going to learn about another deployment strategy called the recreate strategy.