hazelcast®

Hazelcast with Kubernetes

White Paper

By Neil Stevenson
Principal Architect, Hazelcast

Table of Contents

Introduction

The term “cloud-native” is applied to applications that fit naturally into cloud environments, such as those orchestrated by Kubernetes. Architecturally, this means the application is modularized. It is a service composed of multiple module instances, deployed collectively across multiple execution hosts.

The key point is around the multiple instances. The number can be adjusted to control service capacity, and the failure of any one instance does not necessarily result in a service outage.

In-memory data grids (IMDGs) are a natural fit into this methodology. An IMDG is a collective service formed of multiple processes clustering together, to provide data hosting and data processing operations. For Hazelcast IMDG (which we will use in the following examples), these processes are Java virtual machines (JVMs). On Kubernetes, these JVMs run in containers within pods, and the Hazelcast IMDG service is formed using multiple pods.

Hazelcast IMDG

Hazelcast IMDG is a collection of Java processes that join together to provide a communal data service. Each of these storage processes is a Java virtual machine, running in a Docker container, which is the only container in a Kubernetes pod. Kubernetes runs multiple pods to make the IMDG as a whole.

Kubernetes is responsible for managing the pods and finding machines on which to start the pods to build the Hazelcast IMDG cluster of the requested size. Hazelcast will then spread the data across the available pods so that each one stores some of the data.

Both capacity and throughput are scalable. If the pod count is increased or decreased, the data records are rearranged across the pods to balance the load. If the data capacity needs to double, doubling the number of pods achieves this demand. Each pod can host the same number of data records as before, but now there are twice as many of them.

If the data throughput needs to double, doubling the number of pods achieves this. The same data volume on a grid twice this size halves the amount of data per pod. With half as much data as before, each pod can cope with twice the number of requests per record.

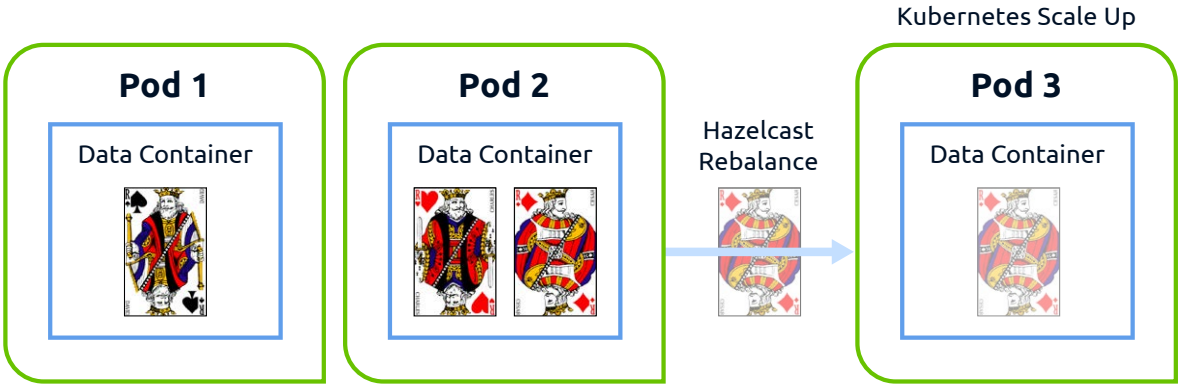

Diagram 1

Diagram 1 shows two pods for Hazelcast IMDG. Each pod has a Docker container running a Hazelcast instance and data is spread across the Hazelcast instances. Here we use playing cards as an example. Pod 1 has "spades." Pod 2 has "hearts" and "diamonds." There are no data records for "clubs" yet. The data balance is as good as it can be with three records and two pods. When Kubernetes is dictated to scale up, pod 3 is started, and Hazelcast can migrate the "diamonds" data records onto it to even the load.

Mainly the IMDG consists of that one type of pod. This provides both data storage and data processing services, where the processing operates on data in-situ to give the highest speed.

Optionally, the grid can have a second type of service: JVMs that don't host data but are still able to process it. This is used when in-situ data processing is not viable, such as when the data records being processed do not reside together.

Finally, there is a third type of service, similar to the second but not Java-based. These are pods that run modules written in other languages, such as Golang and Node.js.

All of these can be scaled independently. The size of the data hosting service is driven by the storage capacity required, whether the processing types are scaled according to workload.

How Hazelcast IMDG Deploys to Kubernetes

Hazelcast has a Helm chart (https://github.com/hazelcast/charts/), and for basic needs, this is the simplest way to deploy the IMDG. All that is needed is the `helm install` command and Kubernetes will do the rest, creating data storage service instances that join together to form a Hazelcast IMDG cluster.

This can be configured with various options, such as `cluster.memberCount` to set the initial pod count for Hazelcast IMDG. Once running, the pod count can be amended while the grid runs, up or down, without breaking service.

One possible customization is the choice of Docker image. This defaults to a pre-built Hazelcast image, but it is easy enough to build your own if you need to bundle other Java classes into the execution image.

If Helm configuration requires significant amendment, the needs are more specialized, or it is your preference, Docker images can be deployed using a `deployment.yaml` file from `kubectl`.

Ultimately, the Hazelcast IMDG service is just a JVM running inside a Docker container, with one container per pod. YAML files give a declaration of the required pods, service locators, and Kubernetes treats it like anything else.

How These Grids Interact with Kubernetes for Discovery Formation

IMDGs provide a collective data service across the pods. Data hosted on one pod may need to be moved to another pod to rebalance the load when the pod count changes. Data hosted on one pod may need to be duplicated onto another pod to provide a resilient mirror.

The Hazelcast pods communicate with each other directly, and for this to work they need to know each other's locations. This is done via the Kubernetes API or Kubernetes DNS, depending on if role-based security is required. Either way, the process is the same.

A Hazelcast pod at start-up uses the API or DNS to find the IP addresses of existing pods and directly connects to those from then on. Pods that host data are part of the grid are registered with Kubernetes against the service label, so that future joiners can find them. Pods that don't host data, such as Golang, only communicate with the data hosting pods and not with each other.

Data Sources

Data is obviously the fundamental part of an IMDG, and in order for data to be hosted, it has to come from somewhere. In some cases, the data is created by running applications, such as HTTP sessions. The data usually already exists on a disk-based store, such as a database, and is loaded into the IMDG and saved back as necessary.

Typically, these databases are single instances, and therefore not cloud-native. They could be run in Kubernetes, but without the ability to scale the pods, this provides little actual benefit. Instead, they are external resources to the Kubernetes cluster.

For Hazelcast, this resource is just a URL, provided as part of the cluster configuration. Each of the data storage pods creates a connection to this external resource and uses it for load and save operations.

External Storage

Hazelcast in Kubernetes runs as a JVM inside a Docker container in a pod. As a general rule, these pods should be thought of as being transient. If a container is restarted, there is no state carried over from the previous start. This allows it to be restarted elsewhere in the Kubernetes cluster.

This is part of being cloud-native. Kubernetes should be given as free a choice as possible as to where the pod/container can go, by minimizing the demands of the application. This fits nicely with Hazelcast; the pod receives the data it is to host across the network from the other Hazelcast pods or from a centralized, external database.

A Hazelcast instance can also save its data to a filesystem and use this as the source to reload from upon a restart. This requires an external filesystem to be mounted as a local filesystem on the container. This is useful if there is ever a need to take the entire Hazelcast IMDG offline. Hazelcast can use this filesystem for its "Hot Restart" store, so if the whole grid is powered down each member can restore its memory content from the saved file.

Dynamically Changing the IMDG

Downtime is a luxury few modern businesses can afford, so the IMDG service needs to stay running. This is where Kubernetes excels.

As already mentioned, the size of the IMDG can be varied by adjusting the number of pods. While this may not be something you need to do, it might happen anyway—hardware can fail and take pods offline. Kubernetes will start these pods up somewhere else, if it can.

The grid size can go up and down, or in this case down and up. Each time the grid size changes, the data records are rearranged to even out the load on the pods. If a request is made for a data record that has been moved, the Hazelcast process in that pod retrieves it from the right pod and returns it to the caller.

So, data change is easy, but what about processing logic change? Historically, everything gets restarted to pick up new code, and that hardly fits with continuous operation. Instead, a Hazelcast instance in one pod can send code to other pods. This enables new code to be rolled out without loss of service.

In Java terms, a data record is just a class where we think mainly about its fields, and processing logic is a class where we think mainly about its methods. Moving either from place to place is part of the built-in Hazelcast logic.

Accessing from Inside of Kubernetes Scope

Applications that work with Hazelcast IMDG use a Hazelcast client library that connects to the IMDG. If these applications run within Kubernetes, then discovering the location of the IMDG pods involves the Hazelcast client library using the same discovery mechanism to find the IMDG pods as the pods use to find each other. Namely, the Kubernetes API or DNS.

If the application that works with the IMDG is also cloud-native, then connectivity is trivial. The Hazelcast client library maintains a connection to all IMDG pods. Requests for particular data records are sent directly to the pod that hosts the record in question. Requests for multiple records are sent in parallel. Interaction with the IMDG is automatically optimized.

Accessing from Outside of Kubernetes Scope

Applications that sit outside Kubernetes can still work with Hazelcast IMDG, easily or efficiently but not both.

The easy way is to connect via a Kubernetes `LoadBalancer` to the IMDG. The external application has no awareness that multiple IMDG pods exist, only that single endpoint. Requests sent to the load balancer pass through to an IMDG pod. If this is not the pod hosting the required data item, that pod will retrieve it from the pod that does. Everything will work correctly, but proxying requests from the selected pod to the right pod adds overhead.

The efficient way is to expose each Hazelcast IMDG pod using its own `NodePort`. The application can then use the Kubernetes API to find all IMDG pods with which to connect. Although not substantially harder to achieve, making individual pods visible is considered an anti-pattern.

If optimal performance is the goal, having the application that works with Hazelcast IMDG outside of the Kubernetes cluster is not ideal.

Data Safety for Multiple Nodes

Data in Hazelcast is sharded and split into sections, known as partitions, which are allocated across the available pods in the Hazelcast cluster service, enabling scalability. Pods can be added or removed from the Hazelcast cluster service to adjust capacity, and the partitions are rearranged on the pods to maintain a balance of the data.

However, failure is an option to consider. Each pod hosts some of the data, and if a pod crashes then the data that was on it is lost. At best this is inconvenient. The data that is lost may be recoverable, either from an external store such as a database or it may be the result of a calculation that can be rerun. Often though, the data is not quickly recoverable or at all, and we don't wish to lose it.

In this scenario, we can use mirroring, a technique in which a data partition that is placed on one pod has a mirror copy placed on another pod. If the original partition is lost, the replica can be used. Naturally, both must be kept up to date when the data records in them are changed.

With a mirroring level of one, each partition is in one pod with one replica of that partition in another pod. Potentially both pods could fail, removing both mirror copies from the cluster. This is most likely to occur if both pods are hosted on the same node, which could have some hardware fault.

If sufficient nodes exist, use of the Hazelcast `ZONE_AWARE` option will direct Hazelcast to use the Kubernetes API to determine appropriate nodes to host partitions and their replica copies. This will insulate against a node failure triggering data loss, which makes the hosted data safer.

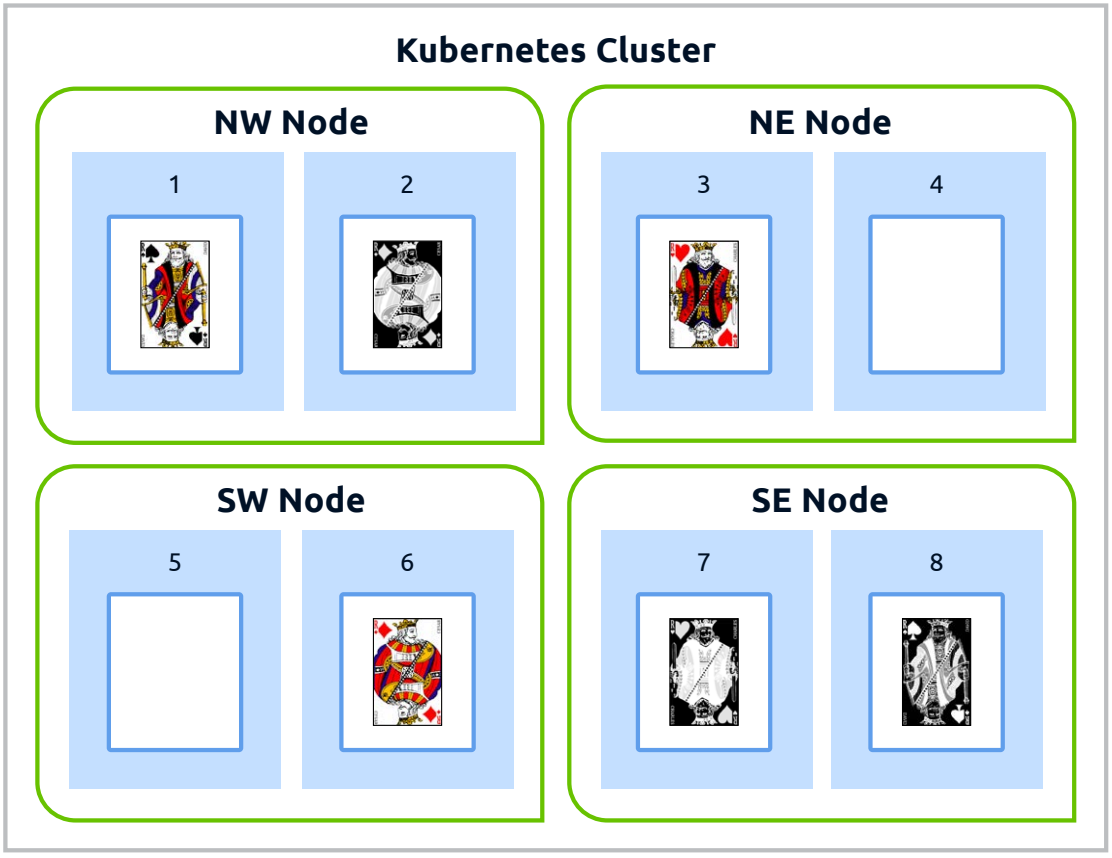

Diagram 2

Diagram 2 shows a Kubernetes cluster made up of four host nodes (NW, NE, SE, and SW). There are again three data records, but now Hazelcast IMDG has been scaled to use eight pods. Each data record has a backup copy shown in reverse color, so eight pods can hold that data, with one backup, that we would normally find in pods without backups. The data record for "spades" goes on pod 1. The backup record could go on any of the other seven pods, but for maximum safety, the `ZONE_AWARE` setting deselects pod two as a candidate as it shares infrastructure with pod 1. A failure on the NW node takes pods 1 and 2 out of action, so pod 2 is a bad choice for hosting a replica of data on pod 1.

Pod Management for Recovery

If a pod should fail, both Hazelcast and Kubernetes will be aware, and conduct recovery according to their declarative configurations.

For Hazelcast, a lost pod means the loss of some partitions (shards) and a size change for the cluster. First, Hazelcast and Kubernetes will recreate the lost partitions on the remaining pods from their replica copies and then migrate some of the partitions from pod to pod to rebalance the data load on the cluster. Both are dependent on the amount of data that needs to be copied and the speed of the network.

For Kubernetes, a lost pod means a reduction in the number of instances below the configured amount, so Kubernetes will restart the lost pod in response. Restarting the pod relies on the infrastructure layer. Generally, the failed pod will not be online again until the Hazelcast partition migration is underway. As this results in a size change for the Hazelcast cluster, this will require that partition migration be re-evaluated against the new cluster size, and the exercise is repeated.

Automatic recovery from Hazelcast is the safer option. Interleaving with Kubernetes recovery may mean more rebalance migration steps are done than necessary, but the cluster remains in a safe state throughout.

A more efficient recovery requires manual control, enabling and disabling the `NO_MIGRATION` Hazelcast option at the correct points. This requires a higher degree of diligence. Data safety is at risk while recovery is deactivated, and there is no guarantee that Kubernetes will be able to find a node to re-activate the missing pod.

Alerting and Ad-Hoc Operations

Hazelcast and Kubernetes do reactive repair. If something fails, recovery actions are triggered. While this is good, we would rather things don't fail in the first place, and this requires more proactive monitoring.

An additional service type is available, `Management Center`, a single pod that provides a federated view of the main IMDG. Although optional, this provides a more granular view of the health of the IMDG than Kubernetes itself is aware of.

This provides Java Management Extension (JMX) statistics, a standard that most remote monitoring tools support, enabling alerts to be generated on thresholds. This means, for example, that capacity needn't run out, as we can detect when it is running low and boost the grid size. Conversely, we can also create alerts when volumes are lower than expected, and investigate if something is impacting incoming work.

The Management Center service also provides a UI, and can thus be exposed via `NodePort` for direct login. This enables a human operator to view the health of the grid, inspect throughput rates, browse data records, and do all the usual sort of console-level operations. For commercial users, license keys can also be managed here.

In addition to alerting and inspection, this UI allows for ad-hoc data operations. If there is a problematic data record that has somehow been loaded to the IMDG, one-off delete or mutate operations could be run, subject to security control.

Benefits

Hazelcast IMDG is composed of multiple clone processes providing a collective service, and each of these containerized processes is a natural fit for Kubernetes. Due to this, Kubernetes provides a solid ecosystem for running Hazelcast IMDG, which is just another application that follows the Kubernetes pattern.

The required grid size can be easily changed using standard `kubectl` commands, so that capacity and responsiveness are manageable. Automatic recovery of the IMDG itself and of the Kubernetes layer means that hardware faults and other unavoidable failures are only a temporary impairment to service quality.

Securing access can be done via Kubernetes, Hazelcast, or both depending on the need. Kubernetes permits or denies access to the IMDG for applications. Hazelcast can refine this further, restricting groups of data records to selected users. Via service labels on Kubernetes, the IMDG service can be made available to other services, making the microservice approach simple to deploy.

Drawbacks

An IMDG is a collection of processes, but the size of the collection is an abstraction that Kubernetes tries not to expose.

A load balancer will alternate requests across the IMDG service's pods, but with each pod holding a different slice of the data, the likely effect is that a request will be sent to the wrong pod (which then has to pass the request to the right pod). The load balancer will appear to equalize the load by stripping requests across the grid but it indirectly increases load by requiring Hazelcast pods to proxy requests to the other Hazelcast pods.

An IMDG inside Kubernetes accessed from outside Kubernetes may be exposed to this issue. The drawback here is that applications may not work ideally if migrated into Kubernetes control individually.

Summary

Hazelcast IMDG provides a data hosting and service that is easily deployed with Helm charts or custom deployment via YAML control files. Once deployed, Hazelcast and Kubernetes will collectively look after the scaling and resilience.

Scale-up and scale-down are controlled as per any Kubernetes pod set. Recovery from failure of individual pods is automatic. This makes for a cloud-native Java data service that provides a communal layer available for easy use by other Java and non-Java cloud-native applications.

About Hazelcast

Hazelcast is the fast, cloud application platform trusted by Global 2000 enterprises to deliver ultra-low latency, stateful and data-intensive applications. Featuring real-time streaming and memory-first database technologies, the Hazelcast platform simplifies the development and deployment of distributed applications on-premises, at the edge or in the cloud as a fully managed service.

Hazelcast is headquartered in San Mateo, CA, with offices across the globe. To learn more about Hazelcast, visit https://hazelcast.com.

2 West 5th Ave., San Mateo CA 94402 USA
Email: sales@hazelcast.com Phone: +1 (650) 521-5453
Visit us at www.hazelcast.com

@hazelcast
https://www.linkedin.com/company/hazelcast
https://www.facebook.com/hazelcast