Chapter 13. Model Serving Infrastructure

Just like any other application, your ML infrastructure can be trained and deployed on premises on your own hardware infrastructure. However, this approach necessitates procurement of the hardware (physical machines) and the GPUs for training and inference of large models (deep neural networks, or DNNs). This can be viable for large companies that run and maintain ML applications for a long time.

The viable option for small to medium-size businesses and individual teams is to deploy on a cloud and leverage the hardware infrastructure provided by cloud service providers such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. Most of the popular cloud service providers have specialized training and deployment solutions for ML models. These include AutoML on GCP and Amazon SageMaker Autopilot on AWS.

When you’re deploying ML models on premises (on your own hardware infrastructure), you can use an open source prebuilt model server such as TensorFlow Serving, KServe, or NVIDIA Triton.

If you choose to deploy ML models on a cloud, you can deploy trained models on virtual machines (VMs) such as EC2 or Google Compute Engine, and use model servers such as TensorFlow Serving to serve

inference requests. Or you may choose to use compute cluster offerings such as Google Kubernetes Engine.

Cloud service providers also offer solutions for managing the entire ML workflow, including data cleaning, data preparation, feature engineering, training, validation, model monitoring, and deployment. Examples of such services are Amazon SageMaker, Google Vertex AI, and Microsoft Azure.

In this chapter, we'll introduce some of the currently available model servers and look at ways to build scalable serving infrastructure. We'll also discuss using a container-based approach to implement your serving infrastructure and enable it to scale. Finally, we will examine ways to ensure that your servers are always reliable and available through the use of redundancy.

Model Servers

Whether you are deploying on premises or on a cloud, model servers simplify the task of deploying ML models at scale. They are similar to application servers that simplify the task of delivering APIs. They can handle scaling and performance, and they perform some amount of model lifecycle management.

Most modern model servers are usually accessible through REST and/or gRPC endpoints. The client sends an inference request to the model server,

and the model server queries the trained model to get the inference result, which it returns to the client. Let’s take a look at three of the leading model servers, starting with TensorFlow Serving and then continuing with NVIDIA Triton and TorchServe.

TensorFlow Serving

TensorFlow Serving (TF Serving) is a flexible, high-performance serving system for ML models (see Figure 13-1). It provides out-of-the-box integration with TensorFlow models and can be extended to serve other types of models. It supports both batch and real-time inferencing. TF Serving helps manage model lifetimes, and it provides clients with versioned access via a high-performance and reference-counted look-up table.

TF Serving also supports multimodel serving, meaning it can serve multiple instances of the same model or different models simultaneously. It exposes the models through gRPC and REST inference endpoints. Deployment of new models can be done easily without changing client code.

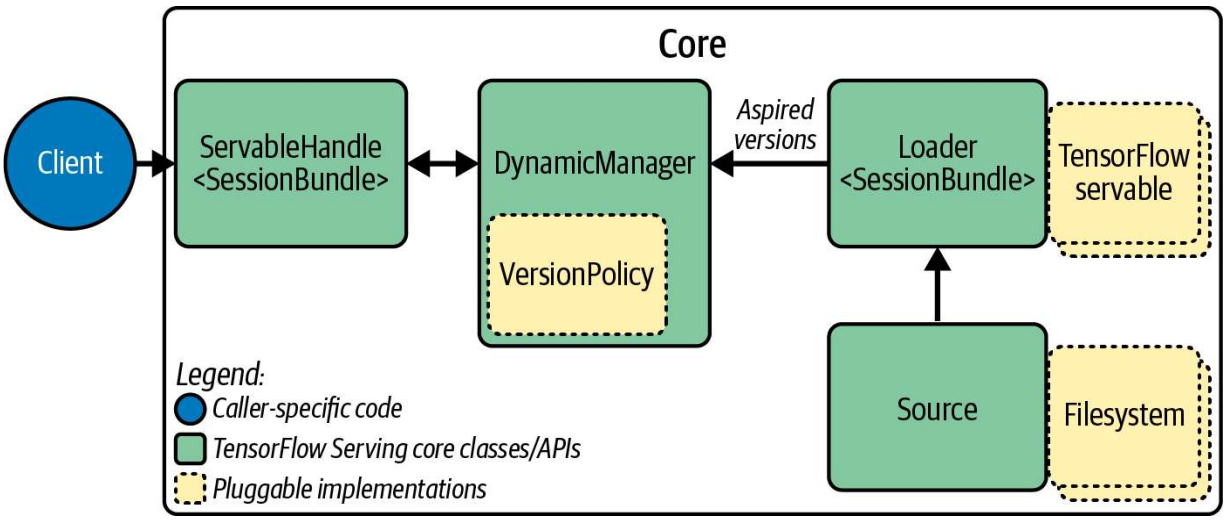

Figure 13-1. The TF Serving architecture

TF Serving has a scheduler that can group individual inference requests into batches for execution on GPUs. It also supports canary deployments and A/B testing.

Let’s take a look at the main parts of this architecture in more detail.

Servables

TensorFlow servables are the central abstraction in TF Serving. Servables are “pluggable implementations,” meaning they are created by developers and added to an instance of TF Serving for their specific serving needs. By far the most common form of servable is a trained model. Servables are the underlying objects that clients use to perform computation (e.g., a lookup or inference). They can be of any type and interface, enabling flexibility and future improvements such as streaming results, experimental APIs, and asynchronous modes of operation.

Typical servables include a TensorFlow `SavedModelBundle` (`tensorflow::Session`), and a lookup table for embedding or vocabulary lookups.

Servable versions

TF Serving can handle one or more versions of a servable over the lifetime of a single server instance. This enables fresh algorithm configurations, weights, and other data to be loaded over time. Versions enable more than one version of a servable to be loaded concurrently, supporting gradual rollout and experimentation. At serving time, clients may request either the latest version or a specific version ID for a particular model.

Models

TF Serving represents a model as one or more servables. A machine-learned model may include one or more algorithms (including learned weights) and lookup or embedding tables.

Loaders

Loaders manage a servable’s lifecycle. The Loader API enables common infrastructure independent from specific learning algorithms, data, or use cases. Specifically, loaders standardize the APIs for loading and unloading a servable.

Sources

Sources are plug-in modules that find and provide servables. Each source provides zero or more servable streams. For each servable stream, a source supplies one loader instance for each version it makes available to be loaded. TF Serving’s interface for sources can discover servables from arbitrary storage systems. Sources can maintain state that is shared across multiple servables or versions.

Aspired versions

Aspired versions represent the set of servable versions that should be loaded and ready. Sources communicate this set of servable versions for one servable stream at a time. When a source gives a new list of aspired versions to the manager (see the next section), it supersedes the previous list for that servable stream. The manager unloads any previously loaded versions that no longer appear in the list.

Managers

Managers handle the full lifecycle of servables, including loading, serving, and unloading. They also listen to sources and track all versions.

Core

Using the standard TF Serving APIs, TF Serving Core manages the lifecycle and metrics of servables. It also treats servables and loaders as opaque objects. Let’s look at an example.

Imagine that a source represents a TensorFlow graph with frequently updated model weights that are stored in a file on disk. When the model is updated, the following events will occur in a running instance of TF Serving:

1. The source detects a new version of the model weights and creates a loader that contains a pointer to the model data on disk.
2. The source notifies the manager of the aspired version.
3. The manager applies the version policy and decides to load the new version.
4. The manager tells the loader that there is enough memory. The loader then instantiates the TensorFlow graph with the new weights.
5. A client requests a handle to the latest version of the model, and the manager returns a handle to the new version of the servable.

NVIDIA Triton Inference Server

NVIDIA’s Triton Inference Server simplifies the deployment of models at scale in production. It is an open source inference server that lets teams

deploy trained models from any framework (TensorFlow, TensorRT, PyTorch, ONNX runtime, or a custom framework), from local storage, or from GCP or AWS S3, on any GPU- or CPU-based infrastructure (cloud, data center, or edge).

Triton uses CUDA streams to run multiple models concurrently. The models can be in any framework that Triton supports. If you have more than one GPU per server, Triton creates an instance of each model on each GPU. All of these instances increase GPU utilization without any extra coding from the user.

Triton supports both real-time and batch inferencing, and even does audio streaming. Users can use shared memory support to achieve higher performance. Inputs and outputs that need to be passed to and from a Triton Inference Server instance are stored in system/CUDA shared memory. This reduces HTTP/gRPC overhead, increasing overall performance.

Triton integrates with Kubernetes for orchestration, metrics, and autoscaling, and it supports both Kubeflow and Kubeflow Pipelines. The Triton Inference Server exports Prometheus metrics for monitoring GPU utilization, latency, memory usage, and inference throughput. It supports the standard HTTP/gRPC interface to connect with other applications, such as load balancers.

Through its model control API, the Triton Inference Server can serve tens or hundreds of models. Models can be explicitly loaded and unloaded into and out of the inference server, based on changes made in the model-control configuration to fit in the GPU or CPU memory. It supports heterogeneous clusters with both GPUs and CPUs, and it helps standardize inference across platforms.

TorchServe

TorchServe is an open source model server designed for serving PyTorch models. It supports both eager and graph mode. It also supports serving multiple models concurrently, as well as versioning, dynamic loading, logging, a CLI, and metrics. TorchServe provides handlers out of the box for common use cases, including image classification, object detection, image segmentation, and text classification.

Figure 13-2 shows the high-level architecture of TorchServe.

To better understand this architecture, let's quickly discuss its main elements:

Frontend

The frontend is responsible for handling requests and responses as well as the model lifecycle.

Model workers

These are running instances of the model that have been loaded from the model store. They are responsible for performing the actual inference. You can see that multiple workers can be run simultaneously on TorchServe. They can be different instances of the same model or instances of different models. Instantiating more instances of a model enables handling more requests at the same time, or increases throughput.

Models

These can be loaded from cloud storage or local hosts. TorchServe supports the serving of eager mode models and JIT-saved models from PyTorch.

Plug-ins (not shown in the figure)

Plug-ins are custom endpoints or batching algorithms that can be dropped into TorchServe.

Model store

A model store is a directory in which all loadable models exist.

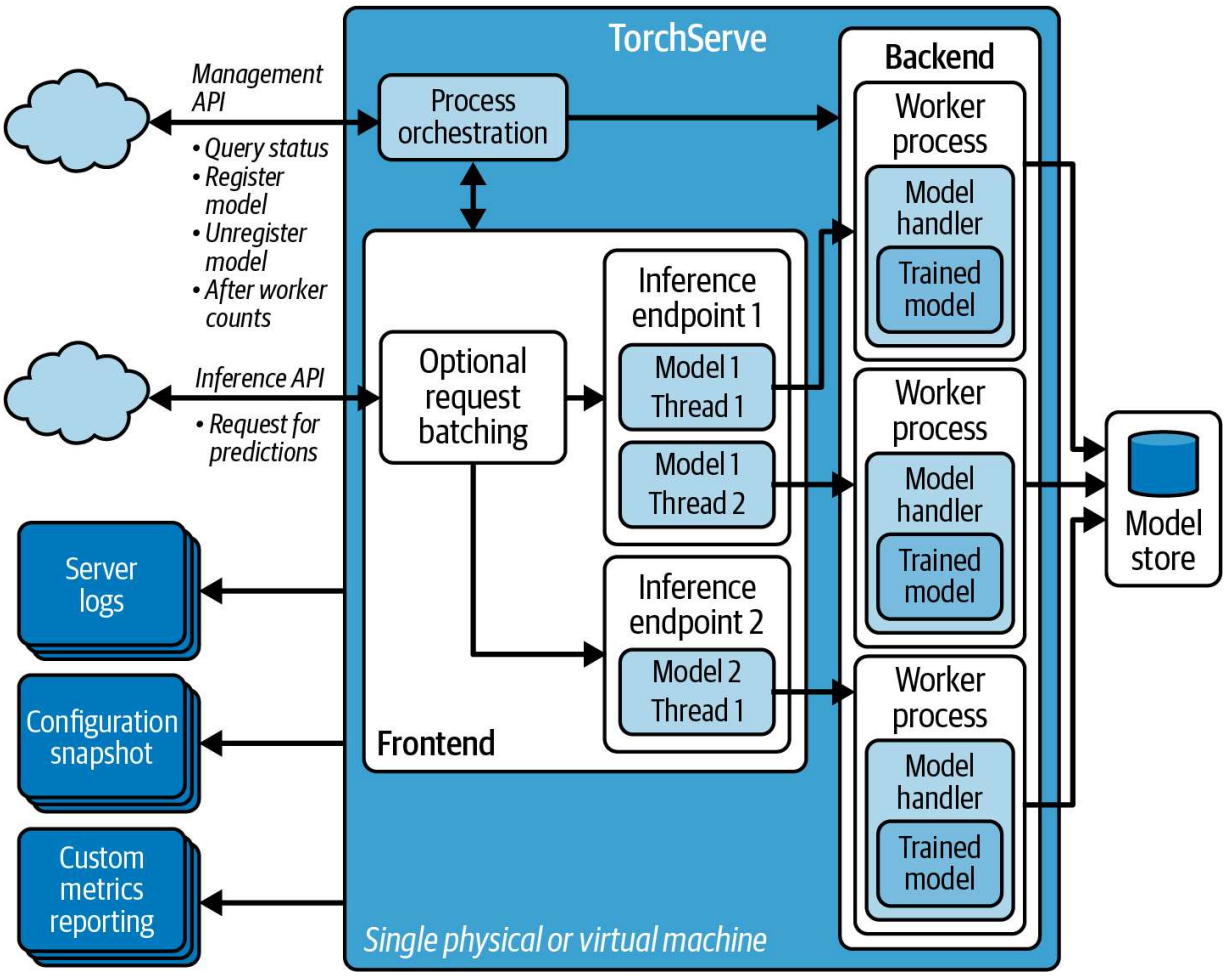

Figure 13-2. The TorchServe architecture (CC BY 4.0)

Building Scalable Infrastructure

For many production use cases, deploying ML models at scale is very important. We are often training large models with billions of parameters on huge datasets. If our infrastructure does not scale gracefully, this can be a significant blocker to operational performance, aka “a big headache.”

Large models and large datasets can easily take days to complete training on a standard CPU or a single GPU. During inference, we need to be able to deal with a high volume of inference requests, to be served simultaneously, often at minimal latencies.

At a high level, there are two types of scaling: horizontal and vertical. *Vertical scaling* adds more power to an existing single instance/node/machine. This usually involves increasing the CPU power and RAM size of a single node used for deployment. *Horizontal scaling* adds more compute nodes to your hardware used for inference. It adds more GPUs or CPUs when load increases, in order to meet the minimal latency and throughput requirements.

In cloud environments, horizontal scaling usually offers the advantage of elasticity. We can scale up the number of nodes based on load, throughput, and latency requirements, and scale back down when we no longer need them, saving the cost of running nodes that we aren’t using.

When you vertically scale a single machine, you will have to take your application offline to upgrade its resources. When you horizontally scale your application, it never goes offline, since you are only adding more servers rather than upgrading existing ones.

Imagine that the load on your application increases due to an increased user base. The application may not be able to handle the increased number of

inference requests with the current hardware infrastructure. Using elastic horizontal scaling, you can simply scale up without disturbing the existing infrastructure by adding more GPUs/CPUs.

If your application uses horizontal scaling, you might run into instance sizing issues. Most cloud platforms have GPUs and CPUs with fixed sizes. You often need to select instance sizes that meet peak requirements, which means many instances may be underused.

As an example, let’s consider scaling in the GCP Compute Engine on the Google cloud. Compute Engine provides three types of scaling:

Manual scaling

: You can simply start and stop instances manually to scale your application.

Basic scaling

: This creates instances when your application receives requests. Each instance will be shut down when the application becomes idle. Basic scaling is ideal for work that is intermittent or driven by user activity.

Autoscaling

: This creates instances based on request rate, response latencies, and other application metrics. You can specify thresholds for each of

these metrics and a minimum number of instances to keep running at all times.

Containerization

Approaches for managing the scaling of infrastructure have evolved over the years. The dominant approach at the time of this writing is known as containerization. Figure 13-3 shows how scaling approaches have evolved.

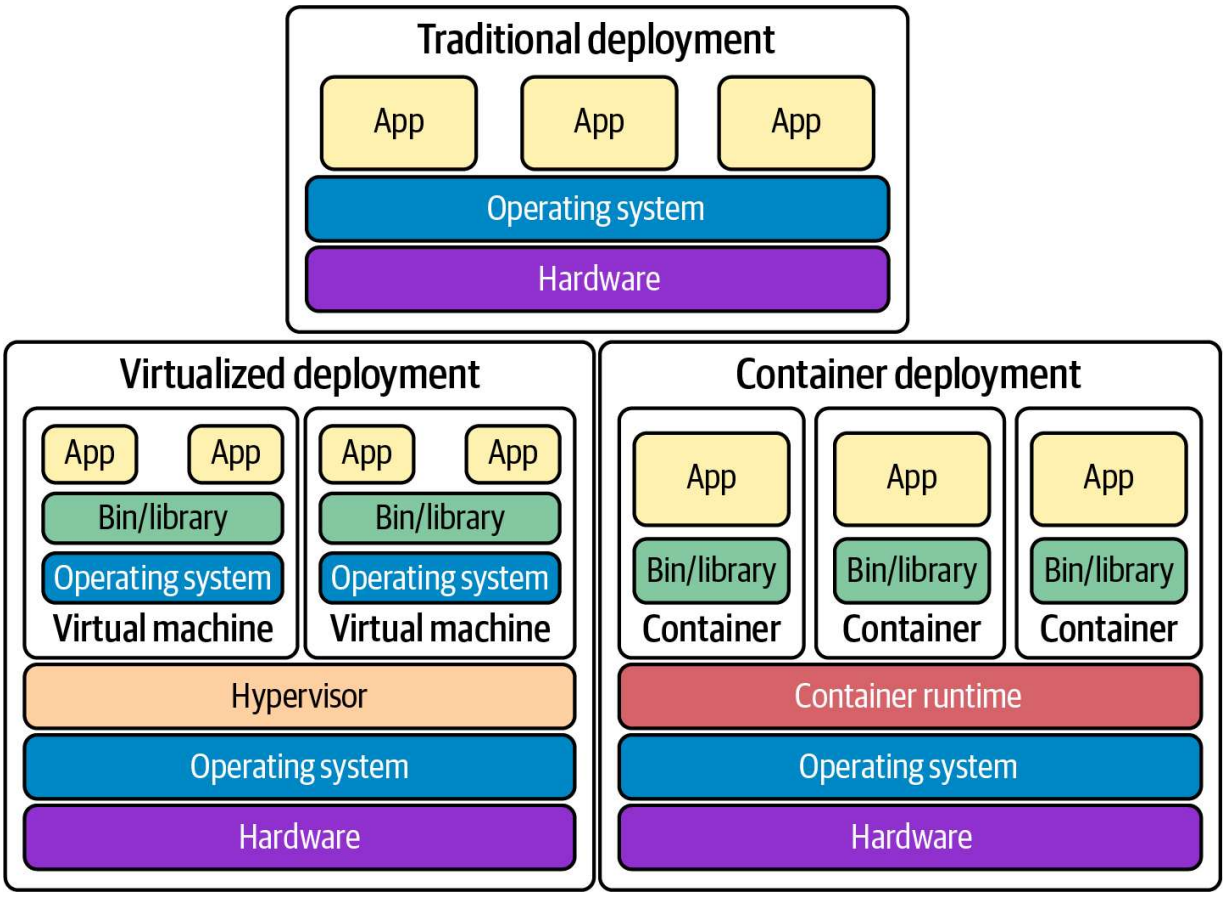

Figure 13-3. The evolution of scaling

Traditional Deployment Era

Back in the day (pre-1999), organizations ran applications on physical servers. It was difficult to define resource boundaries on physical servers, and sometimes this resulted in resource allocation issues.

If multiple applications ran on a single physical server, one of those applications might take up more resources than the others, making it impossible to run applications simultaneously. Depending on the OS, you might even have problems with deadlock. One solution adopted in those times was to run each application on a different physical server, but that doesn’t scale well and results in resources being underutilized.

Virtualized Deployment Era

To solve these issues, virtualization was introduced. The key concept is to use software emulators of machine hardware, known as VMs, to run applications. Applications think that they are running on physical machines because each VM has a full OS and emulated hardware.

Each application runs in its own VM, and is isolated from other VMs running on the same machine, which preserves security between applications. A single physical machine typically runs multiple VMs. It also allows for better scalability, since applications can be easily added and updated, which reduces hardware cost and offers better utilization of

physical hardware. But VMs tend to have a lot of “bloat” in the form of common components, especially the OS itself.

Container Deployment Era

Containers are similar to VMs, but they seek to optimize the isolation properties to share the OS among applications. That makes them much more lightweight than traditional VMs.

By sharing the OS across multiple containers, the size of each container is much smaller than an equivalent VM would be. But from the point of view of an application running in a container, there is no difference between running in a container, in a VM, or on a physical machine. Another benefit is easier and far more fluid deployment of containers.

The most widely used containerization framework today is Docker. Let’s take a look at the Docker framework in detail.

The Docker Containerization Framework

To run containers, you need a container runtime. The most popular container runtime is Docker. A high-level view of the Docker architecture is shown in Figure 13-4.

Docker’s open source container technology started as container technology for Linux and has since grown to become the dominant container runtime

on several platforms. It’s available for Windows applications as well, and it can be used in data centers, personal machines, or a cloud. Docker partners with major cloud services for containerization.

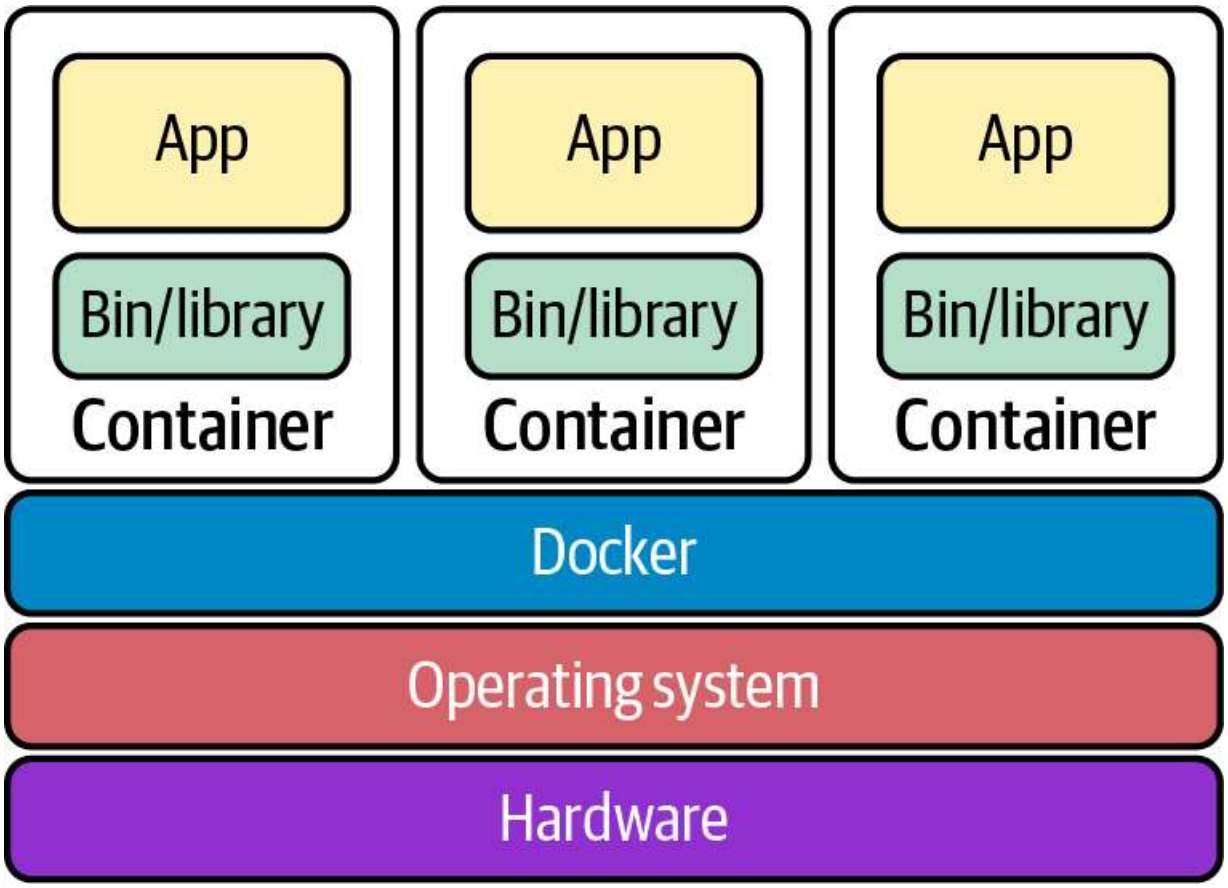

Figure 13-4. A high-level view of the Docker architecture

Docker uses a client/server architecture. As shown in Figure 13-5, the Docker daemon builds, runs, and distributes Docker containers. You can run the Docker client and daemon on the same system, or you can connect to the daemon remotely. Both the client and daemon use REST to communicate. The following subsections describe each element in the architecture in more detail.

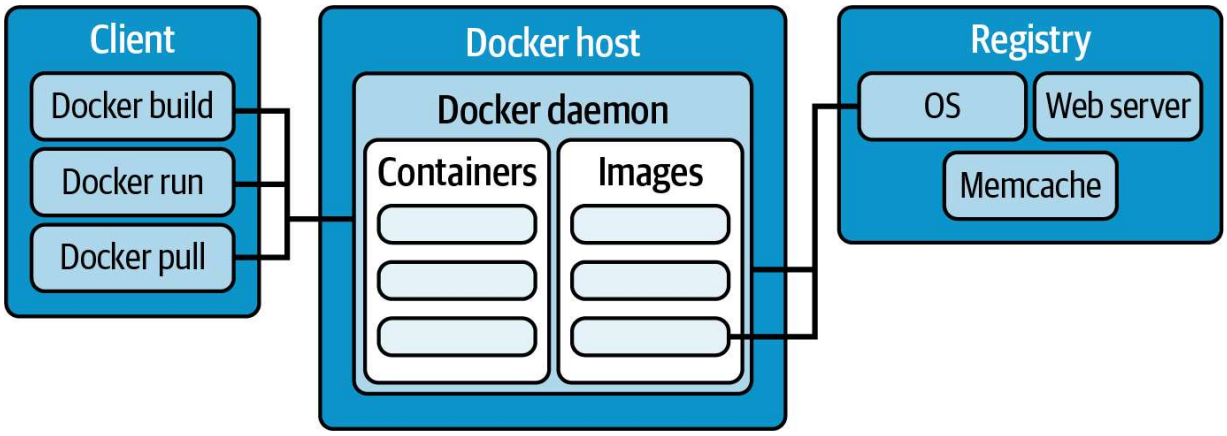

Figure 13-5. Docker processes and communication

Docker daemon

The Docker daemon manages most aspects of a Docker host, including Docker images, containers, networks, and volumes. Daemons on multiple hosts can also communicate and cooperate with each other.

Docker client

Most of the time, you use the Docker client to interact with Docker. That includes basic commands such as `docker run`. The client communicates with daemons to perform your commands and return status.

Docker registry

The Docker registry stores Docker images, which are templates that you use to create container instances. By default, Docker looks for images on Docker Hub, but you can also run your own registry or use a cloud-based

registry such as Amazon Container Registry or Google Cloud Artifact Registry.

Docker objects

You create and use images, containers, networks, volumes, plug-ins, and other objects.

Docker image

A Docker image is a template for creating a Docker container. Images are often based on other images, so you build up an image in layers by inheriting from other images. This usually begins with an image that includes an OS, and then you add things like a web server or other applications by adding new layers.

Docker container

You create a new container by instantiating an image. Containers often need compute resources assigned to them, such as networks and disk space. If you make changes to a container after instantiating it, you can create a new image based on that container.

Container Orchestration

Containers virtualize CPU, memory, storage, and network resources at the OS level, providing developers with a sandboxed view of the OS that is logically isolated from other applications. But as your containerized infrastructure grows, you might need to run multiple containers on multiple machines; start another container when one container goes down to ensure zero downtime; or scale your application to available machines based on varying load. Doing these things with just a container platform like Docker is complex, so orchestration frameworks have emerged.

Container orchestration, shown in Figure 13-6, manages the lifecycle of containers in large production environments. A container orchestration framework is used to perform such tasks as:

- Provisioning and deployment of containers
- Scaling containers up or down to distribute application load across machines
- Ensuring reliability of containers (minimum downtime)
- Distributing resources between containers
- Monitoring the health of containers

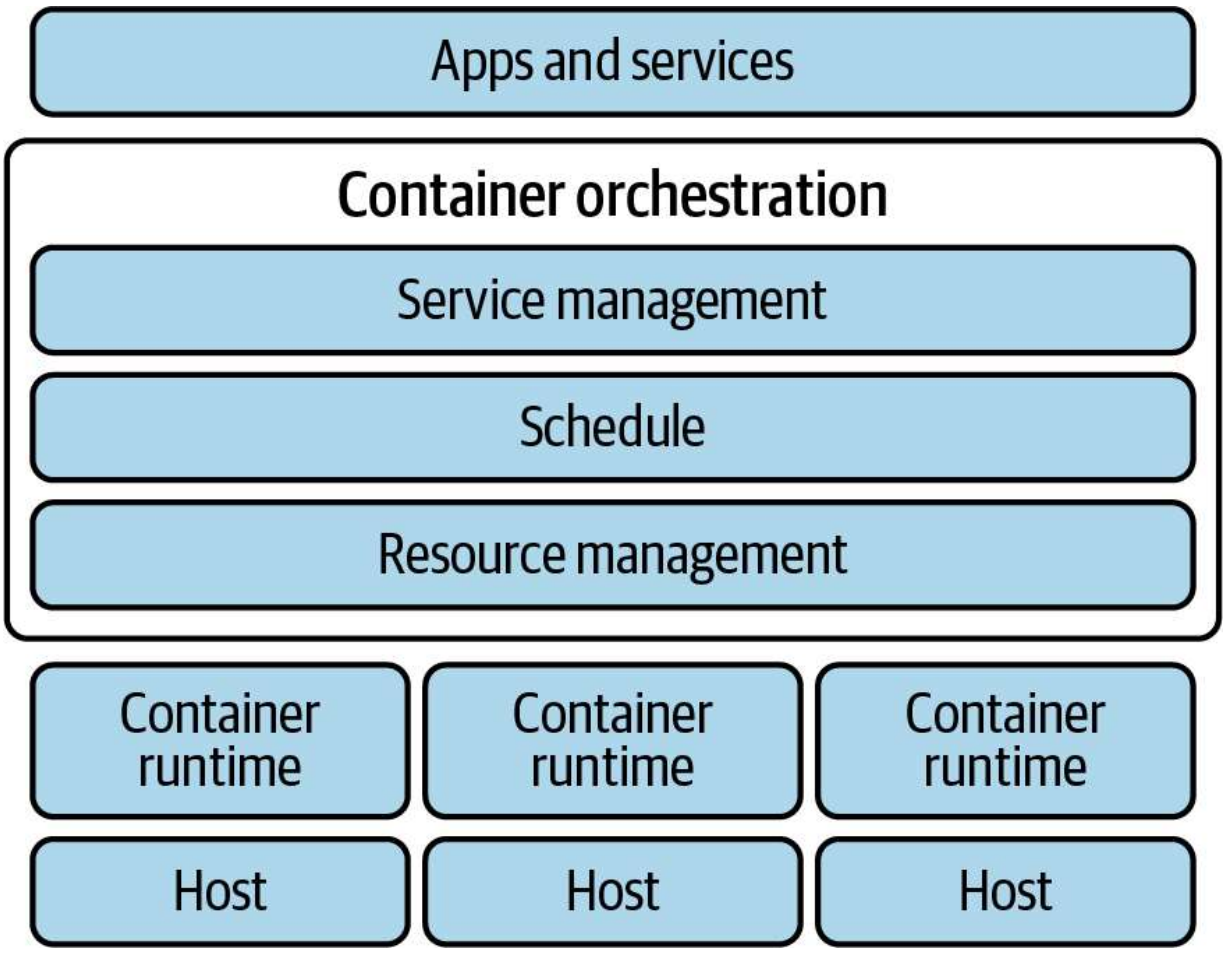

Figure 13-6. Container orchestration

Container orchestration frameworks are generally configuration driven.

You describe the configuration of your application in a set of files using a format such as YAML or JSON, and these files tell orchestration tools where to gather images from, how to establish a connection between the containers, and where to store logs.

Containers are deployed onto hosts in replicated groups. When it’s time to deploy a container, the orchestration framework schedules the deployment and looks for a host to place the container based on predefined constraints.

Once a container is running, the orchestration framework manages its lifecycle based on constraints. Most orchestration frameworks and environments are built for Docker containers, but containers based on the Open Container Initiative can also be used, and are becoming increasingly common.

By far, the two most widely used container orchestration frameworks are Kubernetes (k8s) and the Swarm mode of Docker Engine. Although we will not discuss Docker Engine Swarm mode here, the concepts and structure are very similar to Kubernetes and a Kubernetes feature is supported in Docker Desktop.

Kubernetes

Google originally developed Kubernetes (k8s) as an offshoot of the Borg project. Kubernetes is currently the most widely used framework for container orchestration. Kubernetes can be run on cloud service providers such as Google Cloud Platform, AWS, and Microsoft Azure. It can also be run on premises.

Kubernetes provides you with service discovery, load balancing, storage orchestration, automated rollbacks, bin packing, self-healing, and secret and configuration management. Let’s take a deeper look at these features:

Service discovery and load balancing

Kubernetes routes network requests to a container using a DNS name or IP address. It will also scale container instances to load-balance based on traffic.

Storage orchestration

Kubernetes will mount volumes, such as local volumes, cloud volumes, and more.

Automated rollouts and rollbacks

Kubernetes can automate the creation of new containers for your deployment, removing existing containers as needed and moving all their resources to the new container.

Automatic bin packing

Kubernetes manages a cluster of nodes to run containerized applications. You configure the CPU and RAM requirements for each container, and Kubernetes fits containers onto nodes for maximum resource utilization.

Self-healing

Kubernetes monitors container health, restarting containers that fail, replacing containers, and killing containers that don’t respond to health checks.

Secret and configuration management

Kubernetes stores secrets, including passwords, OAuth tokens, and SSH keys. Secrets and application configuration can be updated without rebuilding container images.

Kubernetes components

To understand Kubernetes you need to have a basic understanding of the components that make up a Kubernetes deployment:

Clusters

A cluster is a set of nodes. Each cluster has at least one master node and at least one worker node (sometimes referred to as *minions*), that can be virtual or physical machines.

Kubernetes control panel

The control panel manages the scheduling and deployment of application instances across nodes. The full set of services the master node runs is known as the *control plane*. The master communicates with nodes through the Kubernetes API server. The scheduler assigns nodes to pods (one or more containers) depending on the resource and policy constraints you've defined.

Pods

A pod is a group of one or more containers. Each container in a pod shares the pod's storage and network resources.

Kubelet

Kubelets are agents. A kubelet runs on each node in the cluster, making sure that the containers in the pod are running. Kubelets start, stop, and maintain application containers based on instructions

from the control plane, and receive all of their information from the Kubernetes API server.

Containers on clouds

The following major cloud providers offer Kubernetes as a service offering:

- Amazon Elastic Kubernetes Service (EKS) fully abstracts the management, scaling, and security of your Kubernetes cluster across multiple zones. It integrates with Kubernetes and Amazon offerings such as Route 53, AWS Application Load Balancer, and Auto Scaling.
- Google Kubernetes Engine (GKE) runs on Google's servers and uses autoscalers and health checks in high-availability environments. It uses autoscalers to manage the scaling of Kubernetes clusters to meet the needs of your application.
- Azure Kubernetes Service (AKS) manages deployment of containerized applications on secure clusters and deploys apps across Azure's data centers.

Kubeflow

Kubeflow is a framework that runs on Kubernetes clusters and is dedicated to making deployments of data workflows on Kubernetes simple, portable, and scalable. Anywhere you are running Kubernetes, you should be able to

run Kubeflow. Kubeflow can be run on premises or on the EKS, GKE, and AKS cloud offerings.

Kubeflow enables deploying and managing data workflows, including complex ML systems at scale. It can also be used for experimentation during the training of an ML model when resource needs are substantial, beyond what can be run on a single machine. It can even be used for end-to-end hybrid and multicloud ML workloads, or for tuning model hyperparameters during training.

Reliability and Availability Through Redundancy

Reliability is usually measured as the probability of infrastructure performing the required functions for a certain period without failure. This can also be expressed as *uptime*, meaning the percentage of time a system is working and available. Reliability is closely related to the concept of *availability*, which is the percentage of time infrastructure will operate satisfactorily at a given point in time under normal circumstances.

To implement a reliable system it’s important to first define your reliability goals using service-level objectives (SLOs) and error budgets. You also need to build observability into your infrastructure and applications, and design for scale. While you usually can’t design for infinite scaling, it’s a

good practice to design for the ability to smoothly scale up to a significant multiple of the highest load you expect to see.

Sometimes developers neglect the need to build flexible and automated deployment capabilities. This is especially important in ML when working in domains that require frequent model updates. You should always anticipate that things will go wrong, and build efficient alerting. This should include a collaborative process for incident management that involves all necessary teams.

For user-facing workloads, look for measures of the user experience; for example, the query success ratio, as opposed to just server metrics such as CPU usage. For batch and streaming workloads, you might need to measure key performance indicators (KPIs), such as rows being scanned per time window, to ensure, for example, that a quarterly report is on track to finish on time, as opposed to just server metrics such as disk usage.

It’s a good idea to establish a service-level agreement (SLA), even if it’s only visible to your own team. An SLA is an agreement you make with clients or users that includes SLOs. When visible to users or customers, an SLA will typically include consequences for failure to meet the SLOs. For example, you may be required to refund or pay fees to customers if you don’t meet an SLO for 99.999% availability.

The following discussions are intended only to introduce these topics, each of which is an entire area of study by itself.

Observability

Designing for observability includes implementing monitoring, logging, tracing, profiling, debugging, and other similar systems. The transparency of your system, and your ability to understand its operation, depends on your implementation of observability. Without it, your system is basically a black box.

You should instrument your code to maximize observability. Write log entries and trace entries, and export monitoring metrics with debugging and troubleshooting in mind, prioritizing by the most likely or most frequent failure modes of the system. Evolve your instrumentation in successive releases of your system, based on what you learn from outages or warning conditions.

High Availability

A system with high availability must have no single points of failure. To achieve this, resources must be replicated across multiple failure domains. A *failure domain* is a pool of resources that can fail independently, such as a VM, zone, or region. For example, a single region master database can cause a global outage if that region has an outage. So, deploying multiple

masters in multiple regions can help guarantee that a failure in one region does not cause a global outage. Figure 13-7 shows how using a load balancer between two deployments (failure domains) in two different regions helps ensure that at least one deployment will be available.

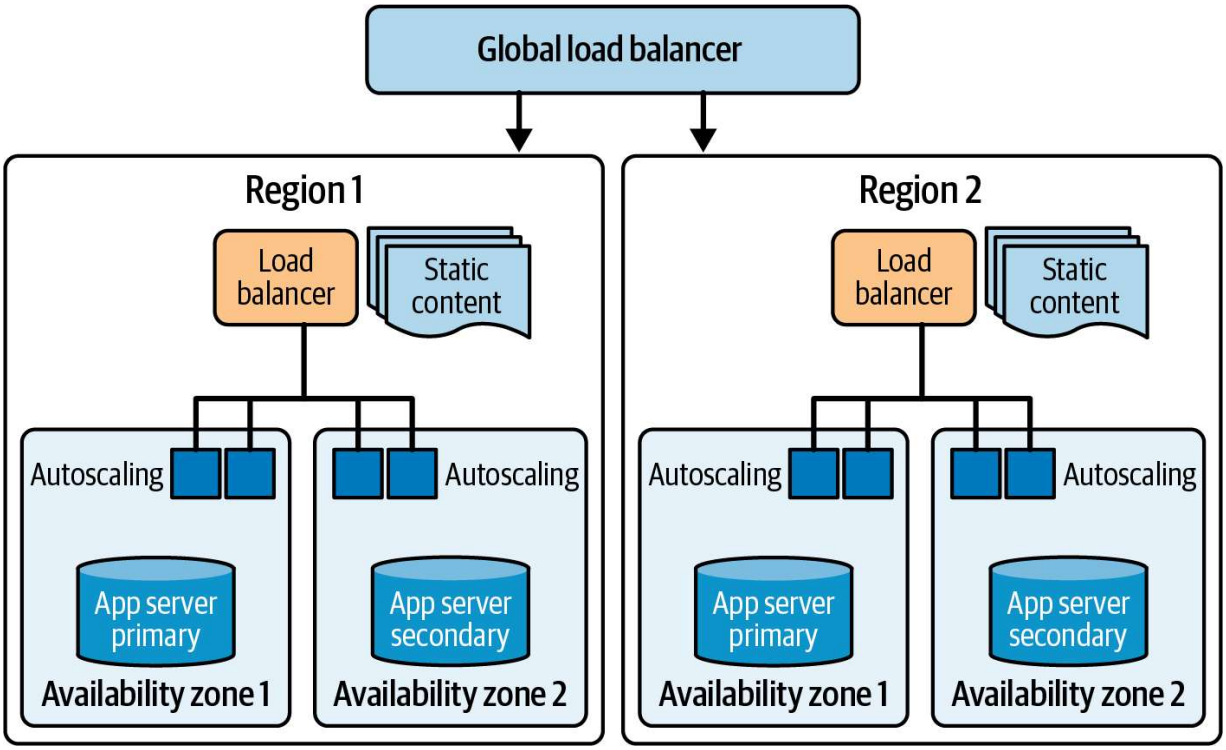

Figure 13-7. How a global load balancer ensures high availability

High availability also requires automatic failover when a failure domain goes down. To the extent possible, you should seek to eliminate single points of failure and deploy redundant systems in multiple failure domains with failover. In many deployments, failover is achieved through the use of load balancers, as shown in Figure 13-7.

System components should be horizontally scalable using *sharding* (partitioning across VMs or zones) so that growth in traffic or usage can be handled easily by adding more shards. Shards should use VM or container types that can be added automatically to handle increases in per-shard load. As an alternative to redesign, consider replacing these components with managed services that have been designed to scale horizontally without requiring user action.

Design your services to detect overload and gracefully deliver lower-quality responses to the user or to partially drop traffic, rather than failing completely when experiencing overload. And of course, design your services to alert the responsible teams or on-call staff. For example, a service can respond to user requests with static web pages while temporarily disabling dynamic behavior that is more expensive, or it can allow read-only operations while temporarily disabling data updates.

If your system experiences known periods of peak traffic (such as Black Friday for retailers), invest time in preparing for such events to avoid significant loss of traffic and revenue. Forecast the size of the traffic spike, add a buffer, and ensure that your system has sufficient compute capacity to handle the spike. If possible, load-test the system with the expected mix of user requests to ensure that its estimated load-handling capacity matches the actual capacity. Run exercises in which your Ops team conducts simulated outage drills, rehearsing its response procedures and exercising the collaborative cross-team incident management procedures. If you can

anticipate a period of significant increase in load, it’s a good practice to scale the system up before that load begins. Don’t wait for a disaster to strike; periodically test and verify your disaster recovery procedures and processes.

Automated Deployments

Automatic deployments of applications should only be implemented as part of automated integration testing using CI/CD pipelines. Assuming that test coverage is sufficient, this should catch any issues before a deployment proceeds.

When implementing automated deployments of your application it’s critical to ensure that every change can be rolled back. Design the service to support rollback, and test the rollback processes periodically. This can be costly to implement for mobile applications, and we suggest that developers apply tooling such as Firebase Remote Config to make feature rollback easier.

A good practice for timed promotions and launches is to spread out the traffic over a longer period, which helps smooth out spikes. For promotional events such as sales that start at a precise time—for example, midnight—and incentivize many users to connect to the service simultaneously, design client code to spread the traffic over a few seconds by adding random delays before initiating requests. This prevents

instantaneous traffic spikes that could crash your servers at the scheduled start time.

Hardware Accelerators

Hardware acceleration is the use of computer hardware specially made to perform some particular set of functions, such as I/O acceleration or floating-point math acceleration. A GPU or TPU is designed to accelerate mathematical computations that are important in training models—in particular, matrix math operations. By using an accelerator in serving infrastructure, compute-intensive functions such as ML model training/inference run much faster than is possible when running on a general-purpose CPU. This is especially important when working with large models, ensembles of models, or tight latency requirements.

There are several popular hardware accelerators. In this section, we will discuss the two most commonly used accelerators: GPUs and TPUs.

GPUs

A graphics processing unit (GPU) is a specialized processor designed to accelerate operations required for rendering graphics. By a happy coincidence, these include matrix math operations, which are also important in ML. This was recognized very early, and GPUs have been used for many years to accelerate processing for ML.

GPUs are designed with a highly parallel structure with multiple arithmetic logic units (ALUs), which helps in increasing throughput. They can be used to speed up training of deep learning models that require billions of operations which GPUs can typically run in parallel. But they can also be used to speed up inference as well.

Currently, NVIDIA manufactures some of the best GPUs in the market. These feature cutting-edge Pascal-architecture Tesla P4, P40, and P100 GPU accelerators.

NVIDIA performed a study that compared the inference performance of AlexNet, GoogleNet, ResNet-152, and VGG-19 on a CPU-only server (single Intel Xeon E5–2690 v4 at 2.6 GHz) versus a GPU server (the same CPU with 1XP100 PCIe). The results showed a peak of 33x higher throughput when using a GPU as compared to a single-socket CPU server, with a maximum 31x lower latency.

However, GPUs, like other accelerator types, do add to the cost of infrastructure. When GPUs are used to accelerate training, this cost may only be incurred during a relatively brief period, but when they are used to accelerate inference, this cost is incurred during the entire uptime of the application, for as many replicas as are required to build reliability and high availability.

TPUs

New accelerators specifically designed for ML applications are currently emerging, and overall the accelerators available are only getting faster. Google's Tensor Processing Units (TPUs) were the first such accelerators, and they remain the most highly developed accelerators designed specifically for ML applications. They are designed to accelerate the performance of linear algebra computations, and they can be used to speed up the training and inference of models, which is heavily dominated by matrix math operations.

TPUs also have on-chip high-bandwidth memory that allows for larger models and batch sizes. They can be connected in groups or pods that scale up workloads with little to no code changes. They are often more power efficient than GPUs. In addition to performance, this also has cost advantages. Figure 13-8 shows a specific example of how TPUs are often more cost-efficient than GPUs. It compares the cost of eight V100 GPUs with one TPU v2 pod.

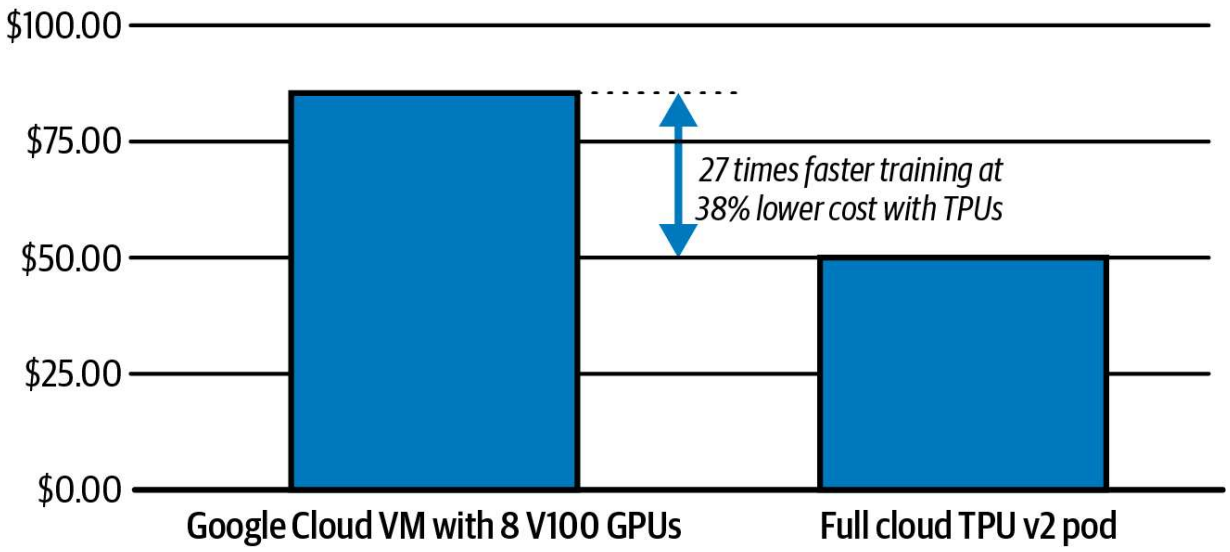

Figure 13-8. Speed and cost advantages of TPUs

TPUs achieve outstanding inference performance because of the focus of their design, which includes Int8 quantization, a DNN inference–specific CISC instruction set, massively parallel matrix processors, and a minimal deterministic design. As shown in Figure 13-8, this results in not only faster performance but also decreased cost.

Conclusion

Outside of academia and research settings, the only reason to train a model is to use it to generate responses to requests, which is generally referred to as “serving the model” or “running inference.” When this is done on centralized infrastructure, such as in a data center or in the cloud, this requires a model server and the surrounding software infrastructure to run it. In this chapter, we discussed in detail the types of model servers