```
11.11.11.14

11.11.11.15
```

Next, you define the following *Ansible playbook*:

```
- hosts: webservers

  roles:

  - webserver
```

Finally, you execute the playbook as follows:

```
ansible-playbook playbook.yml
```

This instructs Ansible to configure all five servers in parallel. Alternatively, by setting a parameter called `serial` in the playbook, you can do a *rolling deployment*, which updates the servers in batches. For example, setting `serial` to `2` directs Ansible to update two of the servers at a time, until all five are done. Duplicating any of this logic in an ad hoc script would take dozens or even hundreds of lines of code.

Server Templating Tools

An alternative to configuration management that has been growing in popularity recently are *server templating tools* such as Docker, Packer, and Vagrant. Instead of launching a bunch of servers and configuring them by running the same code on each one, the idea behind server templating tools is to create an *image* of a server that captures a fully self-contained "snapshot" of the operating system (OS), the software, the files, and all other relevant details. You can then use some other IaC tool to install that image on all of your servers, as shown in Figure 1-3.

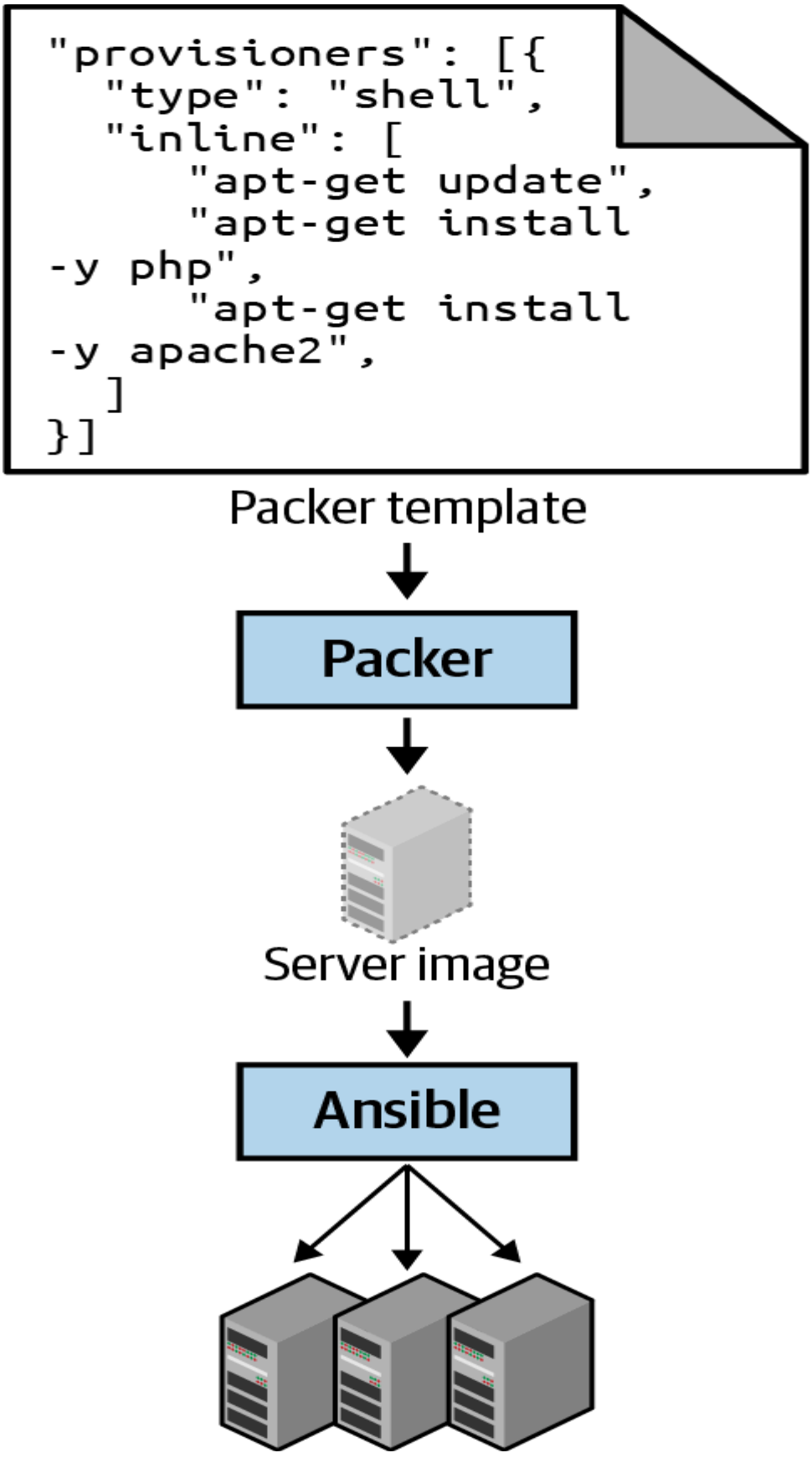
"provisioners": [{
"type": "shell",
"inline": [
"apt-get update",
"apt-get install
-y php",
"apt-get install
-y apache2",
]
}]
Packer template
Packer
Server image
Ansible

Figure 1-3. You can use a server templating tool like Packer to create a self-contained image of a server. You can then use other tools, such as Ansible, to install that image across all of your servers.

There are two broad categories of tools for working with images (Figure 1-4):

Virtual machines

A *virtual machine* (VM) emulates an entire computer system, including the hardware. You run a *hypervisor*, such as VMware, VirtualBox, or Parallels, to virtualize (i.e., simulate) the underlying CPU, memory, hard drive, and networking.

The benefit of this is that any *VM image* that you run on top of the hypervisor can see only the virtualized hardware, so it's fully isolated from the host machine and any other VM images, and it will run exactly the same way in all environments (e.g., your computer, a QA server, a production server). The drawback is that virtualizing all this hardware and running a totally separate OS for each VM incurs a lot of overhead in terms of CPU usage, memory usage, and startup time. You can define VM images as code using tools such as Packer and Vagrant.

Containers

A *container* emulates the user space of an OS.[2] You run a *container engine*, such as Docker, CoreOS rkt, or cri-o, to create isolated processes, memory, mount points, and networking.

The benefit of this is that any container you run on top of the container engine can see only its own user space, so it's isolated from the host machine and other containers and will run exactly the same way in all environments (your computer, a QA server, a production server, etc.). The drawback is that all of the containers running on a single server share that server's OS kernel and hardware, so it's much more difficult to achieve the level of isolation and security you get with a VM.[3] However, because the kernel and hardware are shared, your containers can boot up in milliseconds and have virtually no CPU or memory overhead. You can define container images as code using tools such as

Docker and CoreOS rkt; you'll see an example of how to use Docker in Chapter 7.

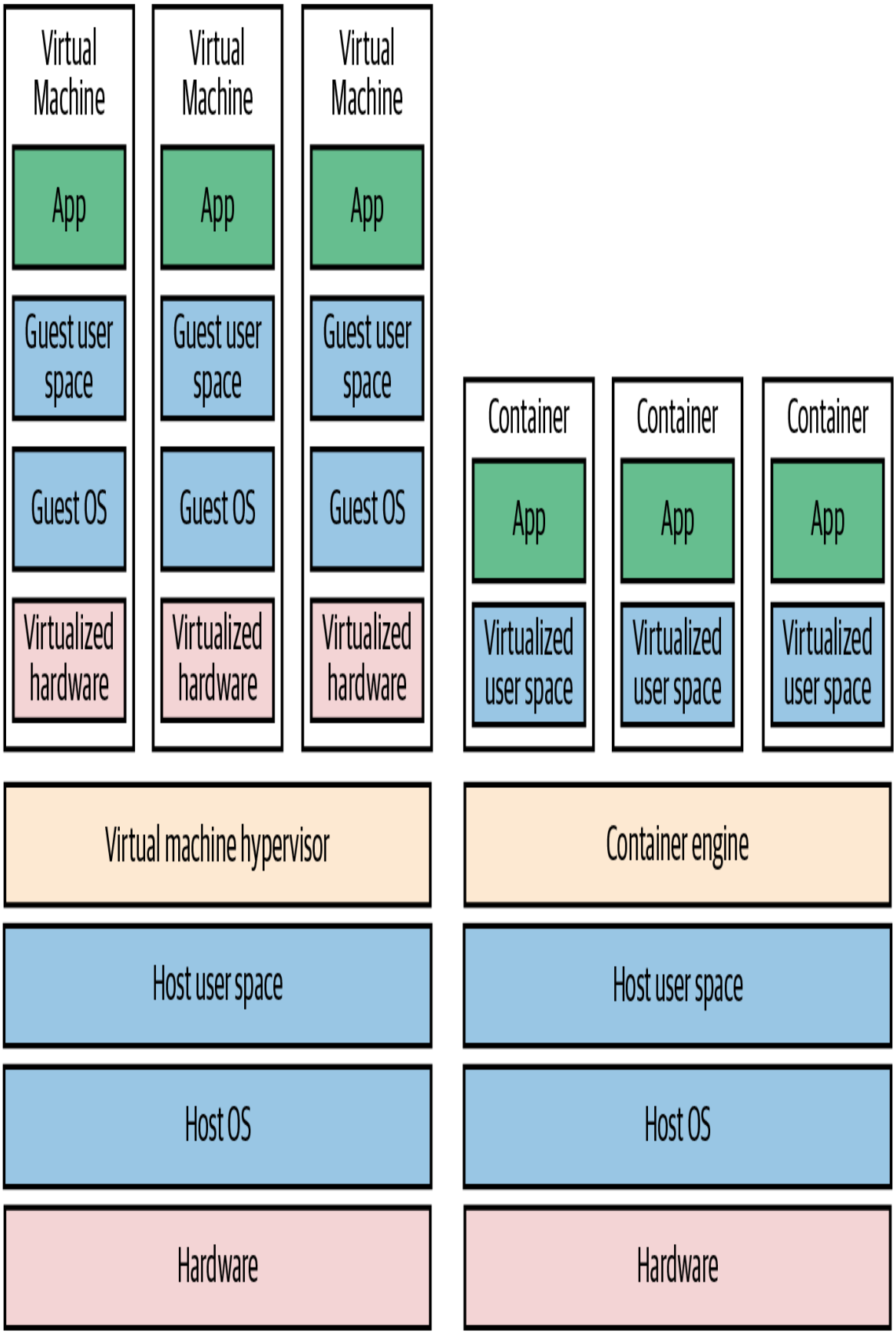
Virtual Machine
App
Guest user space
Guest OS
Virtualized hardware
Virtual Machine
App
Guest user space
Guest OS
Virtualized hardware
Virtual Machine
App
Guest user space
Guest OS
Virtualized hardware
Container
App
Virtualized user space
Container
App
Virtualized user space
Container
App
Virtualized user space
Virtual machine hypervisor
Container engine
Host user space
Host user space
Host OS
Host OS
Hardware
Hardware

Figure 1-4. The two main types of images: VMs, on the left, and containers, on the right. VMs virtualize the hardware, whereas containers virtualize only user space.

For example, here is a Packer template called *web-server.json* that creates an *Amazon Machine Image* (AMI), which is a VM image that you can run on AWS:

```
{
  "builders": [{
    "ami_name": "packer-example-",
    "instance_type": "t2.micro",
    "region": "us-east-2",
    "type": "amazon-ebs",
    "source_ami": "ami-0fb653ca2d3203ac1",
    "ssh_username": "ubuntu"
  }],
  "provisioners": [{
    "type": "shell",
    "inline": [
      "sudo apt-get update",
      "sudo apt-get install -y php apache2",
      "sudo git clone https://github.com/brikis98/php-app.git
/var/www/html/app"
    ],
    "environment_vars": [
      "DEBIAN_FRONTEND=noninteractive"
    ],
    "pause_before": "60s"
  }]
}
```

This Packer template configures the same Apache web server that you saw in *setup-webserver.sh* using the same Bash code. The only difference between the code in the Packer template and the previous examples is that this Packer template does not start the Apache web server (e.g., by calling `sudo service apache2 start`). That's because server templates are typically used to install software in images, but it's only when you run the image—for example, by deploying it on a server—that you should actually run that software.

To build an AMI from this template, run `packer build webserver.json`. After the build completes, you can install that AMI