13 Docker platforms

This chapter covers

- The factors that inform the choice of Docker platform
- The areas of consideration needed when adopting Docker
- The state of the Docker vendor landscape as of 2018

The title of this chapter might seem confusing. Did the previous chapter not cover Docker platforms like Kubernetes and Mesos already?

Well, yes and no. Although Kubernetes and Mesos are arguably platforms on which you can run Docker, in this book we're taking a *platform* to mean a product (or integrated set of technologies) that allows you to run and manage the operation of Docker containers in a structured way. You could think of this chapter as being more infrastructural than purely technical.

As of the time of writing, there are several Docker platforms:

- AWS Fargate
- AWS ECS (Elastic Container Service
- AWS EKS (Elastic Kubernetes Service)
- Azure AKS (Azure Kubernetes Service)
- OpenShift
- Docker Datacenter
- "Native" Kubernetes

NOTE "Native" Kubernetes means running and managing your own cluster on whichever underlying infrastructure you prefer. You might want to run it on dedicated hardware in your own data centre or on VMs on a cloud provider.

The hard part of platform adoption is deciding which platform to choose, and knowing what to consider when looking at Docker adoption across an organization. This chapter will provide a map of the decisions that need to be made in order to make a sensible choice of platform. It'll help you understand why you might choose OpenShift over Kubernetes, or AWS ECS over Kubernetes, and so on.

This chapter is structured in three parts. The first part discusses the factors that inform decisions about which technologies or solutions are appropriate to an organization looking to adopt Docker. The second part discusses the areas that need to be considered when looking to adopt Docker. The third discusses the state of the vendor landscape as of 2018.

We've deployed Docker in multiple organizations, and we've spoken about the challenges of adoption at numerous conferences as well as within these organizations. What these experiences have taught us is that although the combination of challenges these organizations face are unique, there are patterns of decisions and classes of challenges that need to be understood before you go on the container journey.

13.1 Organizational choice factors

This section will outline some of the major factors within your organization that may drive your platform choice for Docker. Figure 13.1 shows some of these factors and their interrelations.

Before discussing these factors in detail, we'll briefly define each and what is meant by it. You may have considered all these factors before and understand what they are, but different terminology within and between organizations can make the terminology unclear, and some terms are more commonly used in some organizations than others.

- *Buy vs. build*—This refers to a difference in approach that organizations have toward new software deployment. Some organizations prefer to buy solutions, and others prefer to build and maintain them themselves. This in turn can influence which platform (or platforms) are chosen.
- *Technical drivers*—Some businesses differentiate themselves on the specific characteristics of their technology, such as high levels of performance or cost efficiency of operation. What underpins these characteristics can be very niche, and specific technical components may not be catered for by commodity services or tooling. This can drive more bespoke solutions that drive a "build" rather than "buy" approach.
- *Monolithic vs. piecemeal*—Again, this is a general cultural approach that organizations can take toward software solutions. Some prefer to centralize solutions in a single monolithic entity (a centralized server or service), and others prefer to tackle problems piece by piece. The latter approach can be seen as more flexible and adaptable, whereas the former can be more efficient at scale.

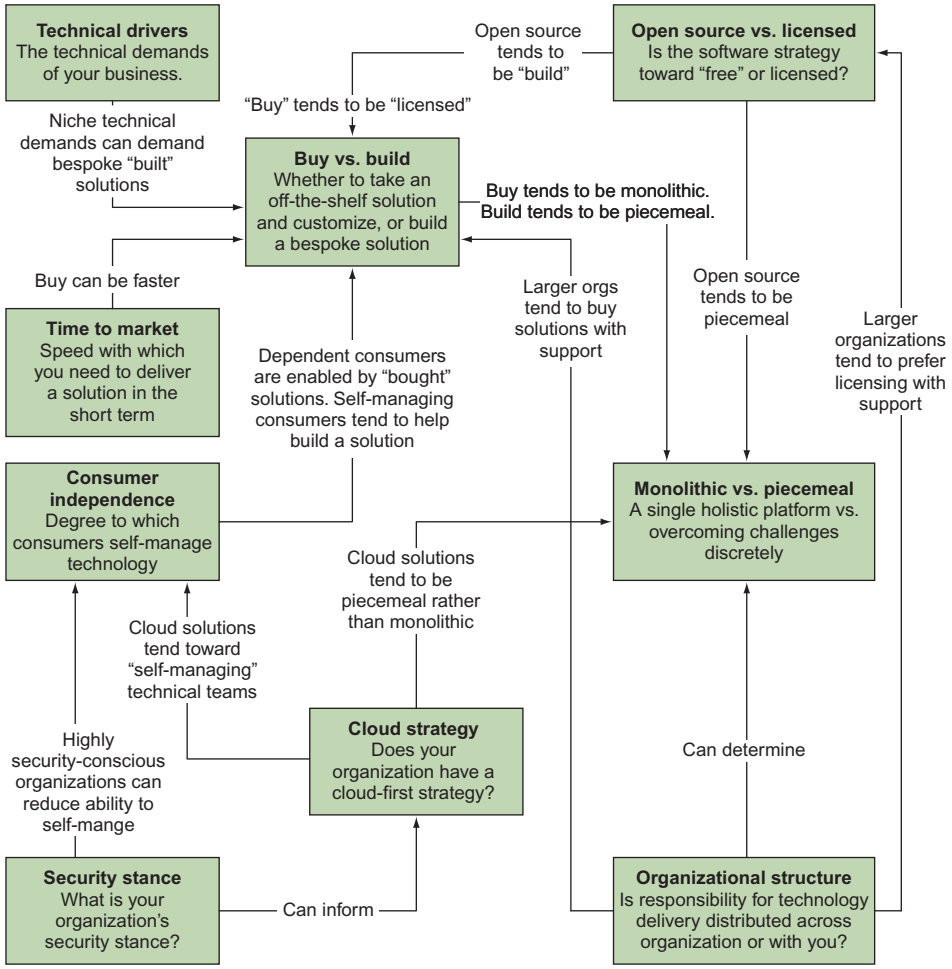

Figure 13.1 Factors driving platform choice

- *Time to market*—Frequently organizations feel a pressure (for commercial or cultural reasons) to deliver a solution to their users quickly. This pressure can favor certain platforms over others at the expense of cost or flexibility in the future.
- *Open source vs. licensed*—Organizations usually have a preference for open source over licensed products these days, but there can still be good reasons to license a product from a vendor. Another related subject that pushes organizations toward open source solutions is fear of lock-in to a particular vendor or platform, leading to increased license costs as dependency on that product persists over time.
- *Consumer independence*—The platform you deploy will have consumers. These could be individuals, teams, or entire business units. Whatever the size of these

consumers, they will have a culture and mode of operation. Key questions to ask here are how technically self-managing are they in their operational context, and how bespoke are their development needs? Answers to these questions may determine the character of platform you decide to deploy.

- *Cloud strategy*—Few organizations have no position defined toward cloud computing these days. Whether you're looking to move workloads to the cloud immediately or not, the degree to which a solution is cloud native can be a factor in your decision-making process. Even if you've decided to move to the cloud, you'll still need to consider whether the strategy is limited to one cloud or is designed to be portable across clouds, and even back to the data center.
- *Security stance*—Increasingly, organizations are taking security more seriously as part of their IT strategy. Whether it's state-sponsored actors, amateur (or professional) hackers, industrial espionage, or plain theft, security is something that everyone has a position on. The level of attention devoted to this area can vary, so this can play a part in platform choice.
- *Organizational structure*—Many of the preceding definitions will potentially mean more to you if you work for an enterprise organization than if you work for the opposite kind of organization.

In this book we've defined *enterprise* broadly as an organization in which there's a low degree of independence between the separate functions within it. For example, if you run a centralized IT function, can you deploy solutions without reference to any other part of the business (such as security, development teams, dev tooling teams, finance, operations/DevOps teams) without consequence? If so, we regard that as the opposite of an enterprise organization. Enterprise organizations tend to be larger (so functions are more discrete), and more regulated (internally and externally), which tends to constrain their freedom to enact change with less consequence.

By contrast, a non-enterprise organization (in this book) is one in which functions are free to deploy solutions as they see fit, through a process of self-determination. By this definition, startups are often seen as non-enterprise organizations because they can make decisions quickly and without reference to—or with speedier determination of—others' needs.

Although non-enterprise organizations tend to favor some strategies (such as build over buy), it can still pay to think about the consequences of such decisions for the business over the long term.

Let's look more specifically at how the various factors interact to militate for or against different platforms. Hopefully some of these will resonate with your experience or situation.

After this discussion, we'll go on to look at the specific challenges that running a Docker platform can bring. With these factors as context, you can come to an informed decision about what technology best fits your organization's needs.

13.1.1 Time to market

It may be helpful first to consider the simplest of the factors: time to market. Everyone working within an organization feels some pressure to deliver solutions quickly, but the extent to which this is negotiable or desirable can vary.

If a direct competitor has adopted a containerization strategy and is using this successfully to drive down costs, then senior management can get interested in how long your solution is taking to deliver.

Alternatively, if you work for a more conservative organization, a speedily delivered solution might be seen to result in negative effects, such as lock-in to a hastily delivered or flavor-of-the-month platform that can't move with changing needs.

Wiser heads may counsel you to resist the urge to adopt the first credible solution in the face of these dangers.

In general, pressure to deliver quickly drives a move toward "buy" over "build" and "monolithic" over "piecemeal" solutions to complex enterprise challenges. (These choices will be discussed further in the next section.) These challenges can be met by assigning responsibility for solving them to those vendors' solutions. But this isn't always possible, especially if the product isn't mature.

Pressure to deliver can also result in the hasty delivery of bespoke solutions that fulfill the short term needs of the business. This is especially prevalent in organizations with a highly technical focus, and it can be very effective, providing an edge over the competition through control over the core technologies and knowledge of their workings. If technology isn't a critical differentiator for your business, though, this can result in white-elephant technology that becomes difficult to move away from later, should the industry outpace your leading edge.

Similarly, adopting click-and-consume cloud technologies can reduce your time to market significantly. The downside can be a consequent lock-in to that provider's solution, driving up costs as you scale, and the cost of any future move away. It can also reduce flexibility in technical features or solutions, making you dependent on the growth and development of the cloud vendor's product.

13.1.2 Buy vs. build

Buying a solution can be an effective strategy in a number of ways. As you've seen, it can result in reducing time to market. If your organization is constrained in terms of development staff, you can also leverage the product's (presumably) expanding feature set to offer more to your customers with relatively little investment.

Buying can also take off the operational cost, if you choose to operate it off-premises, as a service provided by the vendor. The degree to which you're able to take this path may be limited by your security stance: software may be considered safe to run only if it's on hardware owned and operated by the organization using it.

Building a platform yourself, either from scratch or from existing open source software, may appeal to you, since you're reading this book. You would undoubtedly learn

a lot from the process, but there are numerous dangers in such an approach from a business point of view.

First, you'll likely need a highly skilled staff to continue to build and maintain this product. It can be much harder than you think (especially if you've been surrounded by computer scientists at work and at university) to source people who can program and operate complex IT systems, especially in recent years when such skills have been in high demand.

Second, as time goes on, the container platform world will mature, with established players offering similar feature sets and commoditized skills around them. Against these offerings, a bespoke solution built for a specific organization's needs some years ago can seem needlessly expensive where once it was a market differentiator.

One strategy that can be adopted is "build, then buy," where an organization builds a platform to meet its immediate needs, but looks to buy when the market has settled on a product that looks to be a standard. Of course, there's a danger that the built platform becomes a "pet" that's difficult to give up. As of the time of writing, Kubernetes appears to have gained almost complete dominance as the basis of most popular Docker platforms. Therefore, you might drop your bespoke solution in favor of a Kubernetes one if you take the view that that's a good bet for the future.

One platform that made two bets early was OpenShift, which embraced Docker soon after it burst onto the tech scene. It rewrote its entire codebase around Docker and Kubernetes. It's currently a very popular option with enterprises as a result. By contrast, Amazon used Mesos as the basis of its ECS solution, which increasingly appeared niche as Kubernetes became more prevalent.

13.1.3 Monolithic vs. piecemeal

The question of whether to run a single "monolithic" platform for all your Docker needs or to build functionality up from separate "piecemeal" solutions is closely related to the "buy vs. build" question. When considering buying a monolithic solution from a vendor, time to market can be a compelling reason to throw your lot in with them. Again, there are trade-offs with this approach.

The biggest danger is so-called *lock-in*. Some vendors charge for each machine the solution is deployed on. If your Docker estate grows significantly over time, the licensing costs can become prohibitive, and the platform can become a financial millstone around your neck. Some vendors even refuse to support Docker containers delivered by other vendors, which makes realistic adoption of them almost impossible.

Against this is the piecemeal approach. By piecemeal we mean that you can (for example) have one solution for building containers, another for storing containers (such as a Docker registry), another for scanning containers, and yet another for running containers (perhaps even multiple solutions for this or any of the preceding categories). We'll go into more depth about what "pieces" might need solving for in the next section of this chapter.

Again, if you're a small (and perhaps cash-rich) operation that needs to move quickly, the monolithic approach can deliver for you. The piecemeal approach allows you to adopt different solutions for various pieces as the need arises, giving you more flexibility and focus in your efforts.

13.1.4 Open source vs. licensed

Open source has come a long way in the last decade, so that it's now a standard requirement for vendored or supported solutions. This contains within it a danger that's not often obvious. Although many solutions are open source, lock-in isn't necessarily avoided. In theory, the intellectual property of the software is available to use if you fall out with a supporting vendor, but often the skills required to manage and support the codebase are not.

As one conference speaker put it recently, "open source plus vendor support is the new lock-in." One could argue that this is a valid justification for the value the vendor brings to your organization—if it takes a lot of rare skill to manage a required platform, you'll need to pay for it one way or another.

An interesting addition to this mix is cloud computing solutions, which could be regarded as both open sourced and licensed. They're often based on open source software and open standards (such as Amazon's EKS), but they can tie you in to their particular implementation of those standards and technologies, and gain your lock-in that way.

Another interesting mix is seen with platforms like OpenShift from Red Hat. OpenShift is a vendor-supplied platform with licenses required to run it. But its code is available on GitHub, and contributions from the community can be accepted into the main line. What Red Hat supplies as a value-add is support, feature development, and maintenance of the historical codebase. In theory, therefore, you can move off their implementation if you feel you aren't getting value from their offering.

13.1.5 Security stance

Security concerns can have a strong influence on platform choice. Enterprise vendors such as Red Hat have a strong history of managing security, and OpenShift adds in SELinux protection for container security on top of protections already supplied by native Kubernetes.

The degree to which security matters to you can vary enormously. We have been involved in companies where developers have full and trusted access to production databases, as well as companies where paranoia about security is at its highest. These different levels of concern drive very different behaviors in development and production, and therefore in platform choices.

To take one simple example: do you trust your data and code to Amazon Web Services' (AWS's) security standards and products? We aren't singling out AWS here—as far as we know and have experienced, their security standards are generally considered second to none in the cloud space. Moreover, do you trust your development

teams to manage the responsibilities that necessarily lie with the application teams? There have been enough stories of private data being exposed on AWS S3 buckets for this to be a concern for many companies.

> **NOTE** the responsibility for exposing data on S3 is firmly with the consumer of AWS, and not with AWS itself. AWS gives you comprehensive tools to manage security, but they can't manage your security requirements and operations for you.

13.1.6 Consumer independence

One factor that's not often considered is the degree to which teams wish to self-manage. In smaller organizations this tends to vary less than in larger organizations. In larger organizations you can get development teams ranging from highly skilled ones that demand cutting-edge technological platforms to less skilled ones that simply want a curated way to deploy simple and stable web applications.

These differing demands can lead to different platform choices. For example, we've seen environments where one business unit is happy with a centralized, curated, and monolithic platform, whereas another business unit demands a high degree of control and has specific technical requirements. Such users may push you toward a more bespoke platform than the vendored ones. If those users are willing to help build and maintain the platform, a productive partnership can ensue.

If you're large enough, and your development community is heterogeneous enough, you may even want to consider pursuing multiple options for your Docker platforms.

13.1.7 Cloud strategy

Most companies that deal in IT have some kind of stance toward cloud platforms. Some have embraced it wholeheartedly, and others are still starting their journey toward it, are in the process of moving, or are even moving back to the old fashioned data centre.

Whether your organization adopts a cloud Docker platform can be determined by this stance. Factors to consider center around whether there's a fear of so-called "cloud vendor lock-in," where moving your applications and data from the cloud vendor's data centers becomes too costly to countenance. This can be guarded against by using open standards and products, or even by running existing products atop the generic compute resources supplied by those cloud vendors (rather than using their curated and sometimes cloud vendor–specific products).

13.1.8 Organizational structure

Organizational structure is a fundamental characteristic of any company, and it informs all the other factors here. For example, if development teams are separated from operations teams, this tends to argue for adopting a standardized platform that both teams can manage and work against.

Similarly, if responsibility for different parts of the operation are atomized in different groups, this tends to support a piecemeal approach to platform delivery. One example of this that we've seen is the management of Docker registries in larger organizations. If there's already a centrally managed artifact store, it can make sense to simply upgrade the existing one and use it as a Docker registry (assuming it supports that use case). That way the management and operation of the store is cheaper than building a separate solution for what is essentially the same challenge.

13.1.9 Multiple platforms?

One pattern that may be appropriate to mention at this point is that for large organizations with divergent needs, another approach is possible. You may have some consumers that prefer managed platforms they can use, and other consumers in the same organization may demand more bespoke solutions.

In such cases, it can make sense to provide a highly opinionated and easier-to-manage platform for the first set of users, and a more flexible and perhaps more self-managed solution for others. In one case we're aware of, three options are available: a self-managed Nomad cluster, an AWS-managed solution, and an OpenShift option.

The obvious difficulty with this approach is the increased cost of management in running multiple classes of platform and the challenges of communicating these options effectively across the organization.

13.1.10 Organizational factors conclusion

Hopefully that discussion resonated with you, and gave some idea of the complexities of choosing an appropriate platform for Docker (or indeed any technology) within organizations with differing needs. Although it may have seemed somewhat abstract, the next section will be much less so, as we look at the specific challenges you may need to consider when choosing solutions for your business. This discussion has given us the appropriate lenses with which to evaluate those problems and their possible solutions.

13.2 Areas to consider when adopting Docker

Finally, we get to talking about the specific functional challenges that might need to be addressed when implementing a Docker platform.

It's divided into three sections:

- *Security and control*—Looks at items that will depend on your organization's security and control stance
- *Building and shipping images*—Looks at some of the things you'll need to consider regarding development and delivery of your images and workloads
- *Running containers*—Considers what needs to be thought about as you operate your platform

Along the way we'll consider specific current technologies. Mentioning a product in no way implies our endorsement, nor will the products we mention be exhaustive. Software products can improve and decline, and can be replaced or merged. They're mentioned here only to illustrate the practical consequences of your platform choices.

If many of the items we discuss seem obscure, or irrelevant to your organization, it's likely you don't operate under many constraints and therefore have greater freedom to do as you please. If so, you can consider this chapter as offering insight into some of the challenges seen in large-scale and regulated enterprises.

13.2.1 Security and control

We'll deal with security first, because in many ways your security and control stance will fundamentally affect the way you approach all the other topics. Also, if your organization is less concerned with security than other organizations, you may be less concerned with solving the problems outlined in this section.

> **NOTE** By "control" we mean the systems of governance that are overlaid on the development team's and run team's operations. This includes centrally managed software development life cycles, license management, security audits, general audits, and so on. Some organizations have a very light touch, and others are more heavyweight.

IMAGE SCANNING

Wherever you store your images, you have a golden opportunity at the point of storage to check that these images are as you wish them to be. What you might want to check depends on your use case, but here are some examples of specific questions you might want answered in more or less real time:

- Which images have a shellshock version of bash?
- Is there an out-of-date SSL library on any image?
- Which images are based on a now-suspect base image?
- Which images have nonstandard (or plain wrong) development libraries or tools on them?

> **NOTE** Shellshock was a particularly serious set of security flaws in bash discovered in 2014. Security companies recorded millions of attacks and probes related to the bug in the days following the disclosure of the first of a series of related bugs.

Figure 13.2 shows the basic workflow for an image scan in the software development lifecycle. The image is built and pushed to the registry, and this triggers an image scan. The scanner can either inspect the image in place on the registry or download it and work on it. Depending on the level of paranoia you have about images, you can synchronously check the image and prevent it from being used until it's got the OK, or you can check the image asynchronously and provide a report to the submitting

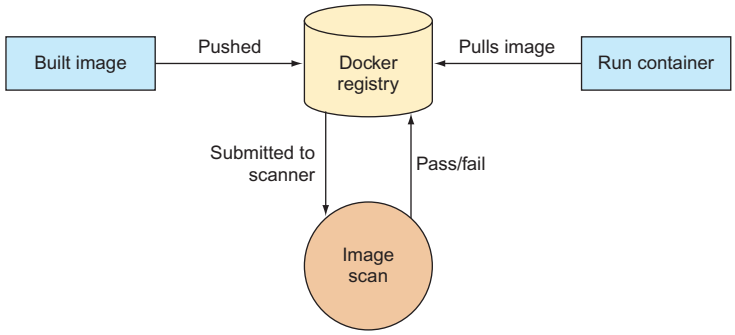

Figure 13.2 Image scanning workflow

user. Usually the paranoid approach is taken for images used in production, and the asynchronous advisory approach is used in development.

In the world of image scanning, there are plenty of options, but they aren't all equal. The most important thing to understand is that scanners roughly divide into two categories: those that focus on packages installed, and those that are primarily designed for deep scanning the software in the image. Examples of the first are Clair and OpenSCAP, and examples of the second are Black Duck Software, Twistlock, Aqua Security, Docker Inc., and many others. There's some overlap between the two categories, but the principal dividing line is cost: it's more expensive to maintain the necessary databases of information to keep up with weaknesses in various types of libraries or binaries, so the deep scanners tend to be far more costly.

This division might be relevant for your decision making. If your images are semi-trusted, you might be able to assume that users aren't being malicious and use a simpler package scanner. This will give you metrics and information about standard packages and their appropriate level of risk without too much cost.

Although scanners can reduce the risk of malicious or unwanted software in your images, they aren't magic bullets. Our experience in evaluating them suggests that even the best ones aren't perfect, and that they tend to be better at identifying issues with some types of binaries or libraries than others. For example, some might more successfully identify npm package issues than (say) ones written in C++, or vice versa. See technique 94 in chapter 14 for an image we've used to exercise and test these scanners.

Another thing to be aware of is that although scanners can work on immutable images and examine the static content of those images, there's still an outstanding risk that containers can build and run malicious software at runtime. Static image analysis can't solve that problem, so you might need to consider runtime control also.

As with all the topics in this section, you must think about what you want to achieve when choosing a scanner. You might want to

- Prevent malicious actors from inserting objects into your builds
- Enforce company-wide standards on software usage
- Quickly patch known and standard CVEs

NOTE A CVE is an identifier for a software vulnerability, to allow for common and unambiguous identification of specific faults.

Finally, you might also want to consider the cost of integrating this tool into your DevOps pipeline. If you find a scanner you're happy with, and it's well-integrated with your platform (or other related DevOps tooling), that might be another factor in its favor.

IMAGE INTEGRITY

Image integrity and image scanning are often confused, but they aren't the same thing. Whereas *image scanning* determines what's *in* an image, *image integrity* ensures that what's *retrieved* from the Docker registry is the same as what was securely placed there. (*Image verification* is another common way to describe this requirement.)

Imagine the following scenario: Alice places an image in a repository (image A), and after it has gone through whatever mandated process exists to check that image, Bob wishes to run that image on a server. Bob requests image A from the server, but unknown to him, an attacker (Carol) has compromised the network, and placed a proxy between Bob and the registry. When Bob downloads the image, he is actually handed a malicious image (image C) that will run code that siphons off confidential data to a third-party IP outside the network. (See figure 13.3.)

The question arises: when a Docker image is downloaded, how can you be sure it's the one you asked for? Being sure of this is what image integrity addresses.

Docker Inc. led the way here with their Content Trust product, also known as Notary. This product signs image manifests with a privately held key that ensures that when the content is decrypted with a public key, the content is the same as what was

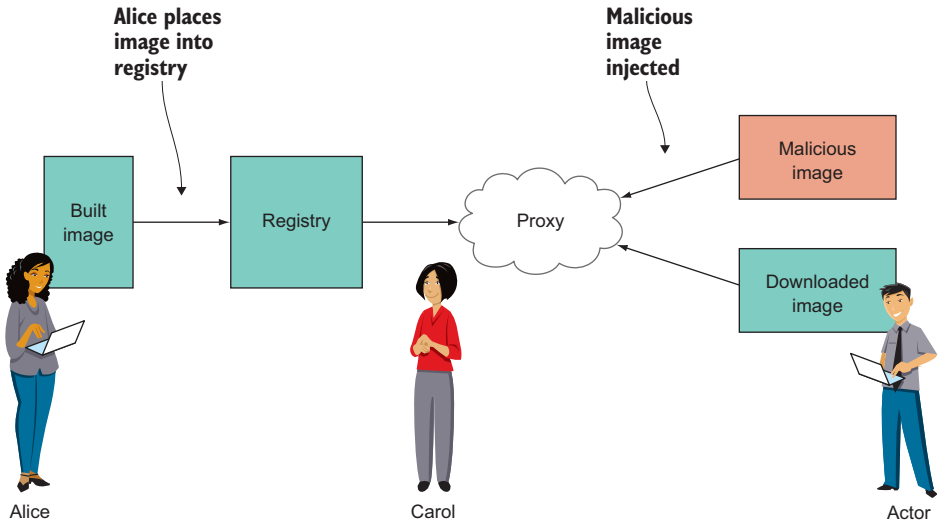

Figure 13.3 Image integrity compromise

uploaded to the registry. Content Trust offers further functionality around delegation of key responsibility that we won't go into here.

Outside of Docker's offering, there's not much to report as of 2018, which is something of a tribute to their engineering lead on this. Leading products like Kubernetes and OpenShift offer very little in this area out of the box, so if you don't buy Docker's products, you may have to integrate these yourself. For many organizations, such an endeavor isn't worth the effort, so they'll rely on existing (likely perimeter) defenses.

If you do manage to implement an image integrity solution, you still must consider how the keys will be managed within your organization. Organizations that care enough to get this far will probably have policies and solutions in place for this.

THIRD-PARTY IMAGES

Keeping on the subject of images, another common challenge when providing a platform is how you're going to approach the subject of external images. Again, the basic difficulty here is one of trust: if you have a vendor that wants to bring a Docker image to your platform, how can you be sure that it's safe to run? This is an especially significant question in a multi-tenant environment, where different teams (who don't necessarily trust each other) must run containers on the same hosts.

One approach is simply to ban all third-party images, and only allow images to be built from known and curated base images using code and artifacts stored within the corporate network. Some vendor images can still be made to work within this regime. If the vendor image is essentially a JAR (Java archive) file running under a standard JVM, the image can be recreated and built within the network from that artifact and run under an approved JVM image.

Inevitably, though, not all images or vendors will be amenable to this approach. If the pressure to allow third-party images is strong enough (and in our experience it is), you have several options:

- Trust your scanner
- Examine the image by eye
- Make the team bringing the image into the organization responsible for its management

It's unlikely you'll entirely trust your scanner to give you sufficient certainty about the safety of a third-party image without the image being fully embedded over time, so responsibility will possibly need to rest somewhere else.

The second option, manually examining images, isn't scalable and is prone to error. The last option is the simplest and easiest to implement.

We've seen environments where all three approaches are taken, with the platform-management team sanity-checking the image, but with final responsibility resting with the application team bringing it. Often there's an existing process for bringing virtual machine images into the organization, so a simple approach is to copy this procedure for Docker images. One key difference worth pointing out here is that although VMs are multi-tenant in that they share a hypervisor with their fellow tenants, Docker

images share a fully featured operating system, which gives attacks a much larger surface area (see chapter 14 on security for more about this).

A further option is to sandbox the running of images on their own hardware environment, such as through labeling Kubernetes nodes on a cluster, or using separate instances of cloud products like ECS, or running an entirely separate platform on separate hardware or even networks.

Secrets

Somehow (and especially when you get to production), privileged information will need to be managed in a secure way. Privileged information includes files or data passed in to builds, such as

- SSL keys
- Username/password combinations
- Customer-identifying data

This passing of secret data into the software lifecycle can be done at several points. One approach is to embed the secrets into your images at build time. This approach is highly frowned upon, as it spreads the privileged data wherever the image goes.

A more approved method is to have the platform place the secrets into your containers at runtime. There are various ways to do this, but several questions that need to be answered:

- Is the secret encrypted when stored?
- Is the secret encrypted in transit?
- Who has access to the secret (in the store or at runtime in the container)?
- How is the secret exposed within the container?
- Can you track or audit who saw or used the secret?

Kubernetes has a so-called "secrets" capability. What surprises many about this is that it's stored in plain text in the persistent store (an etcd database). Technically, it's base64-encoded, but from a security point of view, this is plain text (not encrypted, and easily reversed). If someone were to walk off with a disk containing this information, they could get access to these secrets without difficulty.

As it stands, there are proof-of-concept implementations of applications like HashiCorp's vault to integrate with Kubernetes. Docker Swarm has more secure secrets support out of the box, but Docker Inc. appears to have thrown its lot in with Kubernetes in late 2017.

Audit

When running in production (or any other sensitive environment) it can become key to demonstrate that you have control over who ran what command and when. This is something that can be non-obvious to developers, who aren't so concerned with recovering this information.

The reasons for this "root" problem are covered in chapter 14, but they can be briefly covered here by saying that giving users access to the Docker socket effectively

gives them root control over the whole host. This is forbidden in many organizations, so access to Docker usually needs to be traceable at the very least.

These are some of the questions you might be required to answer:

- Who (or what) is able to run the `docker` command?
- What control do you have over who runs it?
- What control do you have over what is run?

Solutions exist for this problem, but they're relatively new and generally part of other larger solutions. OpenShift, for example, has led the way by adding robust RBAC (role-based access control) to Kubernetes. Kubernetes later added this to its core. Cloud providers usually have more cloud-native ways to achieve this kind of control through (in the case of AWS) use of IAM roles or similar features embedded in ECS or EKS.

Container security tools provided by vendors such as Twistlock and Aqua Security offer a means of managing which particular Docker subcommands and flags can be run by whom, usually by adding an intermediary socket or other kind of proxy between you and the Docker socket that can broker access to Docker commands.

In terms of recording who did what, native functionality has been slow in coming in products like OpenShift, but it's there now. If you look at other products, don't assume functionality like this has been fully implemented!

RUNTIME CONTROL

Runtime control can be considered as auditing at a higher level. Regulated enterprises are likely to want to be able to determine what's running across their entire estate and to report on this. The output of such reports can be compared with an existing configuration management database (CMDB) to see whether there are any anomalies or running workloads that can't be accounted for.

At this level, these are the questions you may be asked to answer:

- How can you tell what's running?
- Can you match that content up to your registry/registries and/or your CMDB?
- Have any containers changed critical files since startup?

Again, this comes with some other products that might form part of your Docker strategy, so watch out for them. Or it may be a side-effect of your overall application deployment strategy and network architecture. For example, if you build and run containers with an Amazon VPC, establishing and reporting what's in them is a relatively trivial problem to solve.

Another frequently seen selling point in this space is anomaly detection. Security solutions offer fancy machine-learning solutions that claim to learn what a container is supposed to do, and they alert you if it appears to do something out of the ordinary, like connect to a foreign application port unrelated to the application.

This sounds great, but you need to think about how this will work operationally. You can get a lot of false positives, and these may require a lot of curation—are you resourced to handle that? Generally speaking, the larger and more security-conscious an organization, the more likely they are to be concerned with this.

FORENSICS

Forensics is similar to auditing, but it's much more focused. When a security incident occurs, various parties will want to know what happened. In the old world of physicals and VMs, there were a lot of safeguards in place to assist post-incident investigation. Agents and watcher processes of all descriptions might have been running on the OS, or taps could be placed at a network or even hardware level.

These are some of the questions that a forensic team might want answered following a security incident:

- Can you tell who ran a container?
- Can you tell who built a container?
- Can you determine what a container did once it's gone?
- Can you determine what a container might have done once it's gone?

In this context you might want to mandate the use of specific logging solutions, to ensure that information about system activity persists across container instantiations.

Sysdig and their Falco tool (which is open source) is another interesting and promising product in this area. If you're familiar with tcpdump, this tool looks very similar, allowing you to query syscalls in flight. Here's an example of such a rule:

```
container.id != host and proc.name = bash
```

It matches if a bash shell is run in a container.

Sysdig's commercial offering goes beyond monitoring to allow you to take actions based on the tracked behaviors against your defined rulesets.

13.2.2 Building and shipping images

With security covered, we come to building and shipping. This section looks at what you might need to think about when constructing and distributing your images.

BUILDING IMAGES

When it comes to building images, there are a few areas you might want to consider.

First, although Dockerfiles are the standard, other methods of building images exist (see chapter 7), so it might be desirable to mandate a standard if a variety of ways might cause confusion or aren't compatible with each other. You might also have a strategic configuration management tool that you'll want to integrate with your standard OS deployment.

Our real-world experience suggests that the Dockerfile approach is deeply ingrained and popular with developers. The overhead of learning a more sophisticated CM tool to conform to company standards for VMs is often not something developers have time or inclination for. Methods like S2I or Chef/Puppet/Ansible are more generally used for convenience or code reuse. Supporting Dockerfiles will ensure that you'll get fewer questions and pushback from the development community.

Second, in sensitive environments, you may not want the building of images to be open to all users, as images may be trusted by other teams internally or externally.

Building can be limited by appropriate tagging or promotion of images (see below), or through role-based access control.

Third, it's worth thinking about the developer experience. For security reasons, it's not always possible to give users open access to download Docker images from public repositories, nor even the ability to run Docker tools in their local environment (see chapter 14). If this is the case, there are several options you might want to pursue:

- Getting approval for the standard tooling. This can be costly and sometimes too costly to achieve due to the security challenges and demands of the business.
- Creating a throwaway sandbox in which Docker images can be built. If the VM is transient, locked down, and heavily audited, many security concerns are significantly alleviated.
- Offering remote access to the above-mentioned sandbox via any Docker client (but note that this does not necessarily significantly reduce many attack surfaces).

Fourth, it's also worth thinking about the consistency of the developer experience when deploying the application. For example, if developers are using docker-compose on their laptop or test environments, they might balk at switching to Kubernetes deployments in production. (As time goes on, this last point is becoming increasingly moot, as Kubernetes becomes a standard.)

REGISTRY

It should be obvious by now that you'll need a registry. There's an open source example, Docker Distribution, but it is no longer the dominant choice, mainly because a Docker registry is an implementation of a well-known API. There are now numerous offerings out there to choose from if you want to pay for an enterprise registry, or if you want to run an open source one yourself.

Docker Distribution comes as part of Docker's Data Centre product, which has some compelling features (such as Content Trust).

Whichever product you choose, there are some potentially less obvious points to consider:

- Does this registry play nicely with your authentication system?
- Does it have role-based access control (RBAC)?

Authentication and authorization is a big deal for enterprises. A quick and cheap, free-for-all registry solution will do the job in development, but if you have security or RBAC standards to maintain, these requirements will come to the top of your list.

Some tools have less-fine-grained RBAC features, and this can be quite a hole to fill if you suddenly find yourself audited and found wanting.

- *Does it have a means of promoting images?*—All images are not created equal. Some are quick-and-dirty dev experiments where correctness isn't a requirement, whereas others are intended for bulletproof production usage. Your organization's workflows may require that you distinguish between the two, and a registry

can help you with this by managing a process via separate instances, or through gates enforced by labels.

- *Does it cohere well with your other artifact stores?*—You likely already have an artifact store for TAR files, internal packages, and the like. In an ideal world, your registry would simply be a feature within that. If that's not an option, integration or management overhead will be a cost you should be aware of.

Base images

If you're thinking about standards, the base image (or images) that teams use might need some consideration.

First, what root image do you want to use, and what should go into it? Usually organizations have a standard Linux distribution they prefer to use. If so, that one is likely to be mandated as a base.

Second, how will you build and maintain these images? In the event of a vulnerability being found, who (or what) is responsible for identifying whether you're affected or which images are affected? Who is responsible for patching the affected estate?

Third, what should go into this base image? Is there a common set of tooling that all users will want, or do you want to leave that to individual teams to decide? Do you want to separate these requirements out into separate subimages?

Fourth, how will these images and subimages be rebuilt? Usually some sort of pipeline needs to be created. Typically this will use some kind of CI tool, such as Jenkins, to automatically build the base image (and subsequently from that any subimages) when some trigger is effected.

If you're responsible for a base image, you may be challenged frequently about the size of this image. It's often argued that thin images are better. In some respects (such as security) this might be argued, but this "problem" is more often imagined than real, particularly with respect to performance. The paradoxical nature of this situation is discussed in technique 60.

Software development lifecycle

A software development lifecycle (SDLC) is the defined process for how software is procured, created, tested, deployed, and retired. In its ideal state, it exists to help reduce inefficiencies by ensuring software is consistently evaluated, bought, and used within a group with a common interest in pooling resources.

If you already have SDLC processes, how does Docker fit in? One can get into philosophical discussions about whether a Docker container is a package (like an rpm) or an entire Linux distribution (because its contents are arguably under the developer's control). Either way, the key point of contention is usually over ownership. Who is responsible for what in the image? This is where Docker's layered filesystem (see chapter 1) comes into its own. Because who created what within the final image is completely auditable (assuming the content is trusted), then tracking back to who is responsible for what part of the software stack is relatively straightforward.

Once responsibility is identified, you can consider how patches will be handled:

- *How do you identify which images need updating?*—A scanner can help here, or any tool that can identify files in artifacts that may be of interest.
- *How do you update them?*—Some platforms allow you to trigger rebuilds and deployments of containers (such as OpenShift, or possibly your hand-rolled pipeline).
- *How do you tell teams to update?*—Is an email sufficient? Or do you need an identifiable person as an owner. Again, your corporate policy will likely be your guide here. Existing policies should exist for more traditional software that's deployed.

The key point in this new world is that the number of teams responsible for containers may be higher than in the past, and the number of containers to assess or update may be significantly higher also. All this can place a high burden on your infrastructure teams if you don't have the processes in place to handle this uptick in software deliveries. If push comes to shove, you may need to force users to update by adding layers to their images if they don't get in line. This is especially important if you run a shared platform. You could even consider using your orchestration tools to put "naughty" containers on specific isolated hosts to reduce risk. Usually these things are considered too late, and an answer must be improvised.

13.2.3 Running containers

Now we'll look at the running of containers. In many respects, running containers is little different from running individual processes, but the introduction of Docker can bring its own challenges, and the changes of behavior that Docker enables can also force you to think about other aspects of your infrastructure.

OPERATING SYSTEM

The operating system you run can become significant when running your Docker platform. Enterprise operating systems can lag behind the latest and greatest kernel versions, and as you'll see in chapter 16, the kernel version being run can be very significant for your application.

Historically, Docker has been a very fast-moving codebase, and not all curated OSes have been able to keep up (1.10 was a particularly painful transition for us, with significant changes to the storage format of images). It's worth checking which versions of Docker (and related technologies, such as Kubernetes) are available to you in your package managers before you promise vendors their applications will run on your Kubernetes cluster.

SHARED STORAGE

As soon as your users start deploying applications, one of the first things they'll be concerned about is where their data goes. Docker has in its core the use of volumes (see chapter 5) that are independent of the running containers.

These volumes can be backed by numerous kinds of storage mounted locally or remotely, but the key point is that the storage can be shared by multiple containers, which makes it ideal for running databases that persist across container cycles.

- *Is shared storage easy to provision?*—Shared storage can be expensive to maintain and provision, both in terms of the infrastructure required and the hourly cost. In many organizations, provisioning storage is not simply a matter of calling an API and waiting a few seconds, as it can be with cloud providers like AWS.
- *Is shared storage support ready for increased demand?*—Because it's so easy to deploy Docker containers and fresh environments for development or test, demand on shared storage can increase dramatically. It's worth considering whether you're ready for this.
- *Is shared storage available across deployment locations?*—You might have multiple data centers or cloud providers, or even a mix of the two. Can all these locations talk to each other seamlessly? Is it a requirement that they do? Or is it a requirement that they do not? Regulatory constraints and a desire to enable capabilities to your developers can both create work for you.

NETWORKING

Regarding networking, there are a few things you might need to think about when implementing a Docker platform.

As seen in chapter 10, by default each Docker container has its own IP address allocated from a reserved set of IP addresses. If you're bringing in a product that manages the running of containers on your network, other sets of network addresses may be reserved. For example, Kubernetes' service layer uses a set of network addresses to maintain and route to stable endpoints across its cluster of nodes.

Some organizations reserve IP ranges for their own purposes, so you need to be wary of clashes. If an IP address range is reserved for a particular set of databases, for example, applications that use the IP range within your cluster for their containers or services may take over those IPs and prevent other applications within the cluster from gaining access to that set of databases. Traffic intended for those databases would end up being routed within the cluster to the container or service IPs.

Network performance can also become significant. If you have software-defined networks (SDNs, such as Nuage or Calico) layered on top of your network already, adding more SDNs for Docker platforms (such as OpenVSwitch or even another Calico layer) can noticeably reduce performance.

Containers can also affect networking in ways you might not expect. Many applications have traditionally used a stable source IP address as part of authentication to external services. In the container world, however, the source IP presented from the container may be either the container IP or the IP of the host on which the container runs (which performs network address translation [NAT] back to the container). Furthermore, if it comes from a cluster of hosts, the IP that's presented can't be guaranteed to be stable. There are ways of ensuring the stability of IP presentation, but they usually need some design and implementation effort.

Load balancing is another area that potentially requires a great deal of effort. There's so much to cover on this topic that it might well be the subject for another book, but here's a brief list:

- Which product is preferred/standard (for example, NGinx, F5s, HAProxy, HTTPD)?
- How and where will you handle SSL termination?
- Do you need a mutual authentication TLS solution?
- How will certificates be generated and managed across your platform?
- Does your load balancer affect headers in a way that's consistent with other applications across the business (be prepared to do a lot of debugging here if it doesn't)?

Finally, if you're using a cloud provider in addition to any data centers you already own or use, you may need to consider whether and how users will connect back to on-premises services from the cloud provider.

LOGGING

Pretty much every application will have log files associated with it. Most applications will want to access those logs in a persistent way (especially in production), so some kind of centralized logging service is usually required. Because containers are ephemeral (where VMs and dedicated servers are usually not), such logging data can be lost if the container dies and logs are stored on its filesystem. Moving to the containerized world might bring the logging challenge more to the fore for these reasons.

Because logging is such a core and common piece of application functionality, it often makes sense to centralize and standardize it. Containers can provide an opportunity to do just that.

MONITORING

Most applications will need to be monitored to a greater or lesser extent, and there is a bewildering array of vendors and products related to container monitoring. This is still an emerging area.

One product that has a great deal of traction in the Docker space is Prometheus. Originally developed by SoundCloud, it has grown in popularity over time, particularly since it became part of the Cloud Native Computing Foundation.

Because containers aren't the same as VMs or physical machines, traditional monitoring tools won't necessarily work well inside containers, as sidecars, or on the host if they're not container-aware.

Having said that, if you're running a cluster of hosts and need to maintain them, traditional, established, mature monitoring tools will come in handy. Likely, they'll be relied on heavily as you try to squeeze the maximum performance out of your cluster for the end users. That's assuming the platform is a success. Our experience suggests that demand often far exceeds supply.

13.3 Vendors, organizations, and products

There's no shortage of companies and organizations looking to make money from Docker. Here we'll look at the biggest and most significant players as of 2018, and we'll attempt to describe where their efforts are focused and how their products might work for you.

13.3.1 The Cloud Native Computing Foundation (CNCF)

The first of these organizations is different in that it's not a company, but it's probably the most influential player in this space. The CNCF was set up in 2015 to promote common standards in container technology. Founding members included

- Google
- IBM
- Twitter
- Docker
- Intel
- VMWare
- Cisco

Its creation coincided with the release of Kubernetes 1.0, which was donated by Google to the CNCF (although it had already been open sourced by Google some time before).

The CNCF's role in the container space is really that of kingmaker. Because the collective might of the various players involved is so great, when the CNCF gets behind a technology, you know two things about it: it's going to have investment and support behind it, and it's unlikely that one vendor will be favored over another. The latter factor is particularly important to the Docker platform consumer, as it means that your technology choice is unlikely to be obsolete in the foreseeable future.

There's a long (and growing) list of technologies that the CNCF has endorsed. We'll look at some of the most significant ones:

- Kubernetes
- Envoy
- CNI
- Notary
- Containerd
- Prometheus

KUBERNETES

Kubernetes was the founding and most significant technology that's part of the CNCF. It was donated by Google to the community, first as open source, and then to the CNCF.

Although it's open source, its donation to the community is part of Google's strategy to commodify cloud technologies and make it easier for consumers to move away from other cloud providers, the most dominant of which is AWS.

Kubernetes is the foundation technology of most Docker platforms, most notably OpenShift, but also Rancher, and even Docker Inc.'s own Docker Datacenter, because they support Kubernetes in addition to Swarm.

CNI

CNI stands for Container Network Interface. This project provides a standard interface for managing network interfaces for containers. As you saw in chapter 10, networking can be a complex area for container management, and this project is an attempt to help simplify its management.

Here's a (very) simple example that defines a loopback interface:

```
{
    "cniVersion": "0.2.0",
    "type": "loopback"
}
```

This file might be placed into /etc/cni/net.d/99-loopback.conf and used to configure the loopback network interface.

More complex examples are available at the Git repository here: https://github.com/containernetworking/cni.

CONTAINERD

Containerd is the community version of the Docker daemon. It manages containers' life cycles. Runc is its sister project, which is the runtime responsible for running the container itself.

ENVOY

Originally built at Lyft to move their architecture away from a monolithic to a microservices architecture, Envoy is a high-performance open source edge and service proxy that makes the network transparent to applications.

It allows straightforward management of key networking and integration challenges such as load balancing, proxying, and distributed tracing.

NOTARY

Notary is the tool originally designed and built by Docker Inc. to sign and verify the integrity of container images. (Please refer to page 317, "Image integrity.")

PROMETHEUS

Prometheus is a monitoring tool that operates nicely with containers. It's gaining currency in the community, with (for example) Red Hat switching from Hawkular to Prometheus in their OpenShift platform.

13.3.2 Docker, Inc.

Docker, Inc. is the commercial entity that seeks to profit from the open source Docker project.

> **NOTE** The open source Docker project has been renamed Moby by Docker Inc. in an attempt to reserve the Docker name for profit-making purposes. So far this name hasn't caught on, so you won't see much mention of Moby in this book.

Docker Inc. was an early leader in the Docker product space, as you might expect. They put together several of their products into a monolithic product called Docker Datacenter. This included support, integration, and features for Notary, the registry, Swarm, and several other projects that Docker had open sourced. Latterly, Kubernetes support has been forthcoming.

Because Docker was early to the party and its technical reputation was strong in the early days of Docker, their product was very compelling on the "getting to production quickly" metric. Over time Docker's product has lost ground as others have caught up. Docker's business model has been difficult to sell internally due to its "take it all or leave it" strategy, and to its cost-per server model, which opens up customers to a strong dependency on one vendor that could hold them to ransom for their entire Docker platform.

13.3.3 Google

Kubernetes was created by Google in 2014 after Docker blew up in popularity. It was intended to bring the principles behind Google's internal container platform (Borg) to a wider audience.

At around the same time, the Google Cloud service came into being. The promotion of Kubernetes was part of their cloud strategy. (Please refer to page 327, "Kubernetes.")

Google has a paid service for managing Kubernetes clusters called Google Kubernetes Engine (GKE), similar to AWS's EKS.

Google's cloud offering is a key business priority for them, and Kubernetes support and encouragement is a central part of that strategy.

13.3.4 Microsoft

Microsoft has been involved with Docker on several fronts, all with a view to expanding its Azure cloud offering.

First, Microsoft has implemented the Docker API to containers natively on the Windows platform from Windows 10 onward. This allows Windows containers to be built and run. Kubernetes support for Windows nodes is planned, but at the time of writing it's still in the early stages.

Second, Microsoft has worked on an offering of its .NET platform, called Dotnet Core (or .NET Core if you prefer), that provides support for .NET codebases on Linux. Not all .NET libraries are supported, so moving your Windows application is far from trivial (so far), but many organizations will be interested in the possibility of running their Windows code on a Linux platform, and even in the possibility of building from the ground up to run on either platform.

Third, an Azure offering exists for Kubernetes (AKS), also similar to AWS's EKS and Google Cloud's GKE.

All these efforts can be seen as designed to encourage users to move to the Azure cloud. The ability to run similar workloads on Windows or Linux (or even the same on both) is attractive to many organizations. This is especially true if the data already

sits on their data centers. In addition, Microsoft is in a good position to offer attractive license bundles to organizations already heavily invested in Microsoft technologies looking to go to the cloud.

13.3.5 Amazon

Amazon now has several container offerings but arguably was somewhat late to the party. Its first offering was the Elastic Container Service (ECS) which used Mesos under the hood to manage the deployment of containers and their hosts.

This had some initial traction but was soon overtaken in the industry by the popularity of Kubernetes. Amazon responded in late 2017 by announcing the Elastic Kubernetes Service (EKS), which (like the GKE and AKS services mentioned previously) is a curated Kubernetes service. ECS is still supported, but it seems only natural to think that EKS will be the more strategic service for them. Also announced in late 2017 was Fargate, a service that runs containers natively without the need to manage any EC2 instances.

All of these services offer tight integration with other AWS services, which is very convenient if you see AWS as a long-term platform for your software. Obviously, AWS's commercial aim is to ensure you want to continue to pay for their services, but their broad support for the Kubernetes API can give consumers some comfort that the ties to the AWS platform can be looser than with other services.

13.3.6 Red Hat

Red Hat's commercial strategy is to curate, support, and manage core software for their customers, the so-called "open source sommelier" strategy. Red Hat is different from the other commercial players in that they don't have a generic cloud service to offer consumers (though OpenShift online can be viewed as a cloud offering because it's an externally hosted service).

Red Hat's container focus is in two areas. The first is OpenShift, which is a product wrapping around Kubernetes that can be run and supported in multiple environments, such as on-prem with the cloud providers mentioned here (as well as some others), and as a service with Red Hat's OpenShift Online service.

OpenShift development has introduced various enterprise features (such as RBAC, built-in image storage, and pod deployment triggers), which have found their way into core Kubernetes.

Summary

- Some of the major determining factors that inform your choice of Docker platform might include your "buy" versus "build" stance, your security stance, your cloud strategy, and whether your organization tends to solve technical challenges with "monolithic" or "piecemeal" products.

- These factors can in turn be affected by the technical drivers of your software, time-to-market demands, the level of consumer independence, your open source strategy, and your organizational structure.
- In a larger organization, a multiplatform approach can make sense, but care might need to be taken to ensure consistency of approach across these platforms to reduce later organizational inefficiencies.
- The major functional areas that might be considered when implementing a Docker platform include how images will be built, image scanning and integrity, secrets management, image registries, and the underlying OS.
- The significant players in the Docker platform space include Docker Inc., the three big cloud providers (AWS, Google Cloud Platform, and Microsoft Azure), and the Cloud Native Computing Foundation (CNCF).
- The CNCF is a highly influential organization that incubates and supports the key open source technical components of Docker platforms. Full acceptance by the CNCF is a signal that the technology will be sustainable.